Dipartimento di Informatica e
Scienze dell'Informazione

JTN: a Visual Notation for the Specification and Development of Java Correct Programs

by

Eva Coscia

Theses Series DISI-TH-1999-XX

DISI, Università di Genova
v. Dodecaneso 35, 16146 Genova, Italy http://www.disi.unige.it/

Università degli Studi di Genova

Dipartimento di Informatica e
Scienze dell'Informazione

Dottorato di Ricerca in Informatica

Ph.D. Thesis in Computer Science

JTN: a Visual Notation for the Specification and Development of Java Correct Programs

by

Eva Coscia

December, 1999

Dottorato di Ricerca in Informatica
Dipartimento di Informatica e Scienze dell'Informazione
Università degli Studi di Genova

DISI, Univ. di Genova
via Dodecaneso 35
I-16146 Genova, Italy
`http://www.disi.unige.it/`

Ph.D. Thesis in Computer Science

Submitted by Eva Coscia
DISI, Univ. of Genova
`coscia@disi.unige.it`

Date of submission: December 1999

Title: JTN: a Visual Notation for the Specification and Development of Java Correct Programs

Advisor: Prof. Gianna Reggio
DISI, Univ. of Genova
`reggio@disi.unige.it`

Ext. Reviewers: Prof. Christine Choppy
18 rue de Valois
91940 Les Ulis, France

Prof. Paolo Ciancarini
Universitá di Bologna
ciancarini@cs.unibo.it

Abstract

In this dissertation we present a visual notation, called JTN, that can be used to produce formal description of systems that can be automatically translated into Java running programs.

In particular, we focus on the benefits of a notation that combines the rigor of a formal specification language (there is a counterpart, given in term of algebraic specifications, for each JTN specification) with the immediacy of comprehension and use of a graphic notation, where the specification is given only by icons and diagrams. The JTN notation includes some object-oriented concepts (and also some restrictions) that makes possible to define a translation function that return a correct running Java program for each JTN specification.

In the first part we present a sort of user manual, informally describing the notation and the use of each kind of diagram. We give a static semantics used to check the well-formedness of the specification diagrams. Then we give the formal semantics of a well-formed JTN specification that is an lts (labelled transition system). Such lts is described by some specifications, written in the algebraic specification language CASL. Finally, we define the translation function and prove its correctness, by showing that the produced program and the CASL specification associated with the same JTN specification are equivalent wrt an observational semantics. For such purpose, we have defined a semantics of Java programs in terms of lts, again.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

The importance and relevance of the formal methods (just f.m. from now on) has been widely emphasized and demonstrated in the academia. Now, the challenge is to gain acceptance in the industrial world and make f.m. recognized as a viable technology for system development process.

There are some reasons, perhaps mistakes that make f.m. refused (see [U.S94]) for practical use:

- most of them are **completely formal**; it is not taken into account the environment (the kind of user) by which they should be used: perhaps programmers have little training in mathematics and, for example, it is a good practice that formal specifications are combined with their description in natural language;
- formalisms cannot be used by their own, they must be tightly coupled with **methods** that clarify the result to obtain and the way of achieving it;
- system developers are concerned with error elimination, so the need of rigorous development makes the application of techniques based on formal methods necessary. But their relevance in industrial application can also be measured by the **automatization degree** they introduce in the development process. They must be equipped with **tools** that help in every phase of the system development and fully (or, at least, partially) automatize code generation;
- most f.m. focuses just on a part of the development process, (as, for example, the requirement description or the specification of the system design) but industry requires methods that covers **all the software life cycle**, from requirements to code testing. A "partial" methodology cannot be well accepted, since developers seems to be willing to spend time for an initial training period and to change their approach to software

development, if the new methodology helps them to actually implements the system. See the success of the use of UML: a unified notation used for describing every aspects of the software life-cycle.

However it is not realistic to work out a completely formal method for the whole development process: it is necessary to integrate f.m. with existing well-accepted development practices, but it is not possible to equate f.m. and software engineering. So, we want to understand how to use f.ms. in the context of development of software systems that is a very complex activity.

By development process we mean the process that, starting from an informal description of a system that a customer gives to an implementor, performs some kinds of activity that transform this description till a codification in a programming language is produced. Inside the development process we can distinguish three phases: the first one consisting of the requirement elicitation and specification, the second one in which some design choices are formalized by less abstract specifications, the third one in which these specifications are implemented by some programming language. Notice that each phase can be further decomposed into several steps: during the design phase the developers can produce several design specifications at different levels of details.

This development process requires a formal framework (i.e., formal models and formal languages) and a method, that describe how to elicit the requirements of the system, how to produce the corresponding formal specification and how to refine these *requirement* specification, moving to more detailed (*design*) specifications and, finally, to the encoding phase.

It is beyond our intentions to produce a formal specification framework from scratch; the starting points are some pre-exisiting works on this topic developed by our group at University of Genova during the past years.

Such works enlighten the importance of both the formalism and the method in the development process (see [AR97]).

The SMoLCS method, presented in [AR87b, AR87a], integrates specifications formalism to specify a system at different levels of abstraction, from the requirement specification phase to the detailed design one. The development process is supported by methodological guidelines: in [G.R94a] and [Reg98] it is presented a method for capture formal requirements of a system. The theoretical and technical aspects has been described in [AR96, GG87].

SMoLCS especially addresses the specifications of reactive/concurrent/distributed systems, in general *dynamic systems*, that change their state along the time and interact with the external environment; such systems can also be concurrent, if the sub-components can, in turn, be modeled as dynamic systems.

The proposed languages for the requirement and the design phases (see [F.P94, GG87, EG95]) follow an algebraic approach, where system requirements are expressed by first-order many-sorted formulae, possibly with temporal operators too, while the design refinements are specified by positive conditional first-order axioms.

The SMoLCS method also gives a notion of implementation between dynamic specifications produced at different steps of the development process.

The formal models of the specifications are isomorphism classes of first-order structure over many-sorted (dynamic) signatures. It is relevant to note that these models can be seen as labeled transition systems, with an associated tree presentation, where the states of the model are the nodes of the tree, and the transition capabilities are the arcs.

Examples of application of the SMoLCS methodology are in [G.R94b] and in [EA96].

One of the most recent works related to the SMoLCS methodology ([ML97]) supplies a *visual* presentation for the design specifications, that is definitely a must to make the formalism competitive with the commonly accepted informal methods: visual notations have big advantages like readability, compactness, understandability. A specification consists of a sequence of diagrams, depicting different aspects of the system. It is completely formal, even if the user does not need to concern him/herself with algebraic formulae.

In this work we try to overcome one of the main weak points we individuated in many formal methods: the lack of an adequate support for the final steps of the development process; a competitive specification framework should cover all the phases, from the (formal) requirement specification, to the code generation.

In order to fill the gap between a formal specification and its implementation, we considered how to make the specification language more targeted to the programming language used in the implementation phase, without loosing the rigor of a formal language.

We chose Java as the implementation language since it is OO, widely accepted for its simplicity and, at the same time, for its richness. It is considered a language for the net, for its portability, but also a language for concurrency and distribution. Moreover, there exists a precise, even if informal, reference [J.G96] for the semantics of the core language.

Formality allow us to check the correctness of the refinement steps (and, thus, of the translation of the design specifications into Java code).

The notation supplied for the formalism that we obtained by the Java-targeting process is completely visual, for the advantages we have described above above.

We started from the existing work presented in [ML97], where it is given a general method for formal visual design specifications of systems. Unfortunately such specifications are too abstract (in the following we refer them as *abstract design* specifications) to be transformed into Java code in just one step, or at least, the transformation is really hard and complex and requires some form of further interaction with the system developer. For example, following [ML97] you can specify systems with n-ary synchronous communications, where the components can exhibit any kind of non-determinism and can be coordinated by complex scheduling policies, which cannot have a direct implementation into Java.

Moreover, the complexity of this transformation into Java does not allow to check the correctness of the generated code, and there is no way to automatize it. Finally, [ML97] does not take into account the relevant, good characteristics of Java, as the OO features.

Our idea is to introduce an intermediate step in the development process that transforms an abstract design specification into a Java-targeted one, whose transformation into a Java program is really easy.

Step (2) can be automatized and guaranteed correct whereas step (1) cannot be automatized, but it can be supported giving a rich set of guidelines for helping the user in such task.

Furthermore, we want to investigate how the Java-targeting affects the whole development process and if it is possible and useful to transform, modify or substitute the formalisms and the method for the requirement and the abstract design specifications with Java-targeted ones; in such case, we should not give guidelines for step (1), but we directly express more abstract properties (perhaps on the behaviour of the system components rather then on the behaviour of the whole system), having in mind the OO concepts and the features and limitations coming from Java.

In this thesis we present `JTN` (see also [CR99]), a visual formal notation for the Java targeted design specifications, obtained by targeting [ML97] to Java, that is by modifying the specification language to take into account the features and the limitations of Java.

`JTN` is visual because every aspect of the system (e.g., components, global architecture and behaviour) is described only by diagrams. But it is formal, because the diagrams composing the specifications are just an alternative notation for logic specifications as in [ML97].

Its characteristic of being Java-targeted makes easy (almost automatic) to translate the `JTN` specifications into Java programs.

`JTN` is meant to be supported by a software environment, called `Gaggia`[1], that can be used from the phase of development of the `JTN` specifications, till the final encoding phase (using

[1]The name `Gaggia`. borrowed from the famous coffee machine, should evoke the idea of a toolset for preparing very good Java product.

Java); it consists of formal tools and techniques, and of a set of user guidelines that describe in detail how to perform the various tasks.

We think that `JTN`, with the associated method and software support tools, could help the development of reliable systems implemented in Java:

- we can describe the system design **visually**, and that is helpful to grasp the system characteristics; however, the `JTN` visual specifications are structured and that avoids one of the possible drawbacks of visual notations: to handle very large diagrams that could be not understood
- the level of the `JTN` descriptions is **not too low**; the designer avoids to specify too much details and the drawings are simple enough; for example, in `JTN` there are user defined datatypes and abstract communication mechanisms, as synchronous and asynchronous channels
- `JTN` specifications are **formal**, with the usual advantages to use formal methods without bothering the specifiers with too much formalities; it is possible to check the consistency of the descriptions of different aspects of the system
- the **automatizable translation** to Java reduces the time to a working system and also gives a prototyper for such specifications

Let us point out that it also possible to use `JTN` directly, that is not in the context of the whole development process. It can be a useful formal and visual notation for describing and reasoning over Java programs.

1.0.1 Specification languages and method: the starting point

Algebraic specification languages, labeled transition systems (*ltss*, see Sect. D), methodological approaches and ideas for requirements and design specifications, as developed over the past years at the University of Genova, are the starting point for our work.

We follow the idea that a reactive or concurrent system can be modeled by a labeled transition tree, whose nodes are decorated by states and whose arcs are decorated by labels. States represent intermediate interesting situations of the system, arcs in the tree represent transitions, that is capabilities of passing from a state to another, where the label of the arcs describe the interactions with the external environment that determine (or are consequence of) the transition.

Many aspects of this approach seem to be adequate for a method that leads to a codification of a system into Java code. By decomposition of specifications, complex systems are decomposed into more simple subcomponents, furthermore there is a distinction among "active" and "passive" components. The active subcomponents can be modeled by labeled transitions trees too. The representation of communication among components by means of interactions seems to be more adequate then other techniques (like streams and so on).

For what concerns specification languages, since conditional first-order axioms are not powerful enough to express properties of concurrent-reactive systems, it is used a temporal branching-time logic (see [E.A99] and a brief presentation in Sect. D) over many-sorted signatures equipped with dynamic sorts and predicates over them, used to describe the states and the transition capabilities of the systems and of their dynamic sub-components. The model of a specification is an (isomorphism class of) of first-order structures over this many-sorted signature.

In the following we briefly introduce the abstract design specification technique of [ML97]; then we describe how it has been targeted to Java.

The specification technique of [ML97] distinguishes among the datatypes, the passive and the active components of a system, because these components have a different conceptual nature and play a different role within the systems.

The *datatypes* are static structures used by the other components, with no idea of an internal state changing over time. The *passive* components have an internal state that can be modified by the actions of other active components. The *active* components have an internal state, but they are also able to act on their own (possibly by interacting among them and with the passive components).

We can completely describe a datatype by giving its values and the operations over them, whereas we describe the passive components by giving their states and the actions that can be performed over them, which obviously can change such states. We want to remark this difference. Data-types define stateless elements (essentially, values) that are used by (active and passive) components. Instead, the passive components are actual components of the system, having an internal state that can be updated by the active components. Finally we describe the active components by giving the relevant intermediate states of their lives and their behaviours, which are the possible transitions leading from one state to another one. Every transition is decorated by a label abstractly describing the information exchanged with the external, w.r.t. to the component, world. Notice that many transitions can have the same source, and that allows to handle nondeterminism.

Let us consider, for simplicity, a system having a database inside. The database is a passive component whose internal state is modified by the operations it supplies outside; the data managed and exchanged by the database are datatypes; and the processes using the database are the active components of the system.

The activity of a system results by describing how its components *cooperate*, i.e., to say which transitions of the active components and which operations over the passive components have to be performed together, and which is the exchange of information with the external (w.r.t. the whole system) world.

A system, in turn, can be seen as an active component of another system, and so we can specify systems with a multi-level architecture.

The underling formal model for an active component (and thus for a system) is a labeled transition system (*lts*), that is triple consisting of a set S of elements (intermediate states), a set L of the labels (interactions with the external world) and a ternary transition relation, a subset of $S \times L \times S$ (transitions). The passive components and the datatypes are modeled by first-order structures.

The visual specifications of [ML97] follow the system structure; thus they consist of diagrams for the datatypes, for the components (the behaviour of the active components is represented as a kind of finite automata) and for the cooperation among them.

This notation associates with a specification several *views* corresponding to activities required by guidelines of the specification method. For example, it is possible to give a drawing for the view that represents static data types, another one for hierarchical relations among basic data, or for visualize intermediate states and interactions of a component.

This notation has big advantages as readability, compactness, understandability and is very useful to have an informal check for the textual presentation. By examining drawings, it is possible to detect mistakes and omission made in the specification.

Formally, these diagrams correspond to an algebraic/logic specification having *ltss* as models. Note that the specification language of [ML97] has neither concepts nor mechanisms related to OO, nor features of some particular programming language, as handshake communications and asynchronous channels; instead it allows the specifier to directly define any feature of the system by describing the corresponding behaviours and cooperation.

1.0.2 Java-Targeting

The choice of the target language deeply influences the whole development process; we cannot simply take an existing development process and introduce a coding phase that uses Java, but the whole method is affected by this new target (e.g. if the target language is OO, it is sensible to use an OO specification style).

Also, the kind of applications for which the method is tailored determines the kind of properties that have to formalized and the design to describe; this has an immediate impact on various phases and on the definition of the specification language.

The choice of the implementation language, on the contrary, heavily impacts on the design specification language, that must be adequate to (abstractly) describe implementation choices that must be coded into Java.

We designed `JTN` by adapting the technique presented in Sect. 1.0.1 to the features and to the limitations of Java: the system architecture is decomposed into subcomponents, that are specified independently and then the whole system activity can be specified in terms of these subcomponents' activities, using the concept of *cooperations*. But we have to eliminate any local non-determinism as well as n-ary cooperations, that are not supported by Java mechanism. In [EG94, EA96, Reg98, ML97], the cooperations can be described at a very high level: we can specify which components are involved and which is their role, that is which is the activity that each one performs in order to have a global system activity. The problem is that there is no explicit ordering, thus we do not know which is the component that starts the activity, which are (if any) the cause-effect relations among the activities of different components neither if, for example, a component that has performed its activity is blocked until the overall cooperation has finished. Obviously, a direct implementation in Java is not possible: the cooperation must be further refined and transformed in an ordered sequence of one-to-one simple cooperations.

We have also to establish which components are purely concurrent and, thus, will be implemented by Java threads.

We want to keep distinct, at this more concrete level too, the concepts of datatypes, passive and active components of a system. In our opinion, it is useful to have this distinction to avoid confusion and to make the specification more readable.

Then, we introduce the new OO concepts of class and instance and provide an explicit representation of the relevant relationships among them, as inheritance and use; but in `JTN` we have three kinds of classes, one for each kind of entities that we consider: datatypes, passive and active components.

The datatypes are described at an abstract level by giving their constructors and by defining the associated operations, without considering an OO perspective; however it is easy to transform them into Java classes.

The passive components are seen as objects, whose state is given by a set of typed fields and the operations to modify them are methods. The transformation of such object specifications into Java classes is immediate.

Passive components are shared among the active ones, that can modify them by method calls. In order to take under control the effects of such calls over the whole system, a passive components cannot control, that is neither access nor modify, other (sub) components. Thus the object classes in `JTN` lacks the concepts of components (that is sub-objects).

The active components are seen as processes, with a state, an independent activity and communication channels to interact with other active components and with the external world. In this case, the natural implementation is given by Java threads. We chose to use communication among processes via channels, rather than via method calls. In this way, a process does not offer methods outside and we do not have to manage method calls while the process is performing its activity (in Java, there is no built-in mechanism to disable method calls).

As Java objects and threads communicate by method calls and streams, the main typologies of cooperation are: between a process and an object, by means of a call to an object method, and among a set of processes by communication along asynchronous and synchronous channels. The asynchronous channels are rendered by streams, and the synchronous ones are implemented by particular additional objects.

Java supports only system architectures of at most two levels. The first level corresponds to multi-threaded Java programs, and the second corresponds to distributed Java applications consisting of programs possibly running on different machines.

The second level has not been included in `JTN`, yet; but it is one of the most interesting and promising directions of our future work.

1.0.3 The translation into Java: the correctness check and the Java Semantics

The other main topic of our work concerns the translation of the `JTN` specifications into Java.

Shortly we present the main ideas underlying the translation: every `JTN` class is translated into a Java class; the constructors of the datatype values and of the process states are translated into fields; the process classes becomes thread classes in Java; the asynchronous channels are implemented by streams, the synchronous ones are realized by particular classes, using the Java synchronization mechanisms to suspend and resume threads.

Obviously such translation must be checked to be correct w.r.t. the `JTN` specification, by suppling convincing arguments and an equivalence relation among the models of the Java programs and the models of the specifications.

Thus we need a Java semantics that is simple and clear: this makes easy to understand what a program do and, thus, if it really implements the functionality described by the specifications. But we also require that it supplies information about the observations that can be done over the models: these observation must be similar to the ones over the models of the specifications.

The main directions of our work are:

- to establish a Java sublanguage powerful enough for the kind of applications we are interested in;
- to look for aspects of the language that are not relevant (in the sense described above), consequently, which the semantics can abstract from;
- to give it a formal operational (abstract) semantics and find out a reasonable notion of equality between programs, that makes possible to establish when two programs are equal up to some non-relevant implementation details.

We have chosen a sublanguage that is expressive enough to cover every kind of architectural implementation choice or behavioral property we can specify in the design phases. So, the choice of the sublanguage is heavily related to the class of systems for which the method has been developed.

In order to combine simplicity and clearness with the need of explicit information about the external visibility of a program execution, we have followed an operational style (an SOS style) and the Java programs are modeled by a transition system, in which every transition represent the execution of some activity by one or more concurrent thread inside the program, in turn threads are modeled by other transition systems. The labeling of the transitions determine the interactions of a Java program with the external environment and, thus, the observation over a program execution.

Let us stress that our semantics is both operational and denotational, as it completely describes a program execution but it also associates a value (i.e., a transition tree) to each Java program. So, we are able to reason about possible equivalence relations on such values.

From the investigation on the Java semantics, we noticed that it is possible to give an abstract semantics of a wide subclass of Java programs, containing all the programs produced by the translation of the `JTN` specifications.

The abstraction mainly concerns the specification of some low-level details about the use of the shared memory by concurrent threads. Simply introducing minimal restrictions on the form of the programs (essentially, by forcing synchronized access to the shared objects), we can banish from the subclass of interest some programs having unpredictable behaviours and, thus, programs that do not implement anyone of the systems on which we aim to use the `JTN` notation. Some other aspects we abstract from (as they are not observable on the program behaviour) are the priorities assigned to threads and the real-time aspects of the Java language.

This simplification greatly improves the understandability of the models, and, thus, of the correctness check, because a program is modeled by a transition system with simpler intermediate states and fewer transition capabilities.

We have developed both the concrete (that is the one that can be applied to all the Java programs, as it models also these low-level details) and the abstract versions of the semantics. Then we have proved ([ER99]) that they are equivalent if applied to the previously described sub-class of programs. In this thesis we just report the abstract version, that is the one that we use to check the translation correctness.

Some formal semantics has been given by other authors to Java, but they do not suit very well our needs, because in most of the cases they are not intuitive and simple or do not offer information about the observability of the program behaviour; thus it is hard to find a sensible notion of equality among Java programs and between Java programs and specifications models.

One of the first work on the Java semantics has been published by Drossopolou and Eisenbach ([S.D97]); it mainly concerns the Java type system, even if it is also given a run-time semantics for a very small sublanguage. From their work we borrowed the idea of annotating the Java terms with the type information deduced in the static semantics: such information will be used for the *dynamic* semantics. The one we propose differs from theirs, especially because [S.D97] lacks some important features, including: class variables and methods, local variables, `super` and concurrency that are included in the Java sublanguage we use for the translation of the `JTN` specifications. Also the information that we use to annotate terms are more complex that the ones used in [S.D97].

The semantics in [PBM97, PRM98] proposes a different approach. It is mainly focused on the concurrent aspects of Java and, in particular, on the modeling of the shared memory by using event structures. Such approach avoid to explicitly describe the details about the execution order of certain operations, because it uses well-formedness conditions expressing which are the correct ordering of the events that model interactions between the working memories of concurrent threads and the shared main memory.

However, we think that such approach it is not detailed enough, as it does not describe the small steps of the program executions, and, more important, it does not supply any information that can be used to define an observational equivalence on the class of the models: the semantics in [PBM97, PRM98] is not denotational, thus it is very hard to understand which could be a sensible equivalence notion among Java programs.

The semantics given by using Abstract State Machines in [EB98, BE98] covers a great part of Java language, also focusing on the possibility to distinguish a subset of Java programs whose semantics could abstract from the detailed description of memory interaction. However, again, in [EB98, BE98] the models proposed for the Java programs do not help in finding an equality notion.

1.1 Overview

The thesis has been divided into two parts. The first one presents the JTN notation, giving the informal descriptions of its diagrams, the correspondence among the visual presentation and the textual one (precisely described by a BNF grammar), the system for type checking the specifications, the formal semantics of the diagram, described by using the algebraic specification language CASL.

The second one is focused on the Java language and its relation with JTN. We present the translation of the JTN specifications into Java programs, then we provide the abstract version of our Java semantics, in order to have the modelization of a Java program. Such models are used to check the correctness of the translation of a JTN specification, in the last part of the thesis.

1.1.0.0.1 Part 1

Chapter 2 presents a manual for the use of the JTN notation. For each kind of diagram supplied by JTN, it describes its informal meaning and its use, as well as some alternative notations. Some constructs and shortcuts presented in the user manual, that do not significantly increase the expressive power of the language, are ignored in the following, essentially for a question of time; however it is trivial to extend the work of the thesis to include them.

The transformation of a visual presentation into the corresponding textual one, that will be used in the other chapters, and the BNF grammar describing the textual JTN syntax are described in the Appendix A and B.

Chapter 3 defines the static semantics of JTN, by providing a proof system that allows to give a type to the well-formed fragments of a JTN specification. It is defined over the textual presentation of the specification, that is equivalent to the visual one.

Chapter 4 is focused on the formal semantics of the JTN terms. Again, we supply a proof system by which it is possible to associate a formal specification to each well-formed (w.r.t. the static semantics) JTN specification.

We have chosen CASL (see [Mos97]) as the language for describing the formal specifications: it has been developed in collaboration with a group of the University of Genova and seems a good compromise between simplicity and expressive power. It also supplies an extension (CASL-LTL , see [E.A99]) for the specification of dynamic systems in which it is possible to express temporal properties of the systems.

A brief presentation of the CASL language as well as some technical notions related to algebraic specifications and observational semantics have been grouped together in Appendix

D.

1.1.0.0.2 Part 2

Chapter 5.1 precisely defines the translation of the JTN specifications into Java code. Such translation is given by means of a function over the JTN textual presentations that returns, for each specification, a Java program.

Chapter 6 contains the formal semantics of the Java language, given using a SOS style. The model of the Java programs, like the one of a CASL specification, is a labeled transition system. Thus it is easier to compare them in order to check the correctness of the translation phase.

Chapter 7 defines an observational equivalence relation among Java programs and CASL specifications. By showing some arguments and observations, we check that this equivalence holds for each CASLspecification and the Java program obtained by the JTN specification.

Finally, Chapter 8 addresses the future work.

Appendix C contains an example of JTN usage: it contains the visual JTN specification of a desktop calculator.

Note for the reader

Chapters 3,4, 5.1 can be read independently.

Chapter 4 requires a certain familiarity with algebraic specifications, while chapters 5.1, 6 assume a little knowledge of the Java language.

Chapter 2

JTN: the user manual

2.1 Overall structure of the JTN specifications

In the following we present some intuitive ideas and explanations about the JTN diagrams, about their possible alternative forms or shortcuts, plus some little guidelines for their use. This is also rendered by referncing parts of the calculator example that is completly reported in App.C.

We factorize a system specification into several diagrams showing different aspects or parts of the system. Thus the diagrams are not too large and complicated, and so really useful. For example, some diagrams focus on the behaviour of the components and other focus on the architectural structure of the system.

There are three different kinds of diagrams:

- the *class diagram*
- the *architecture diagram*
- the *sequence diagram*

2.1.0.0.3 Class Diagram The *class diagram* captures the classes of the system components and their relations.

All the information about a class is given by two complementary diagrams: *interface* and *body*. The first one describes which are the services (different for each kind of class) that the class offers outside. The latter defines such services and can be given apart from the class diagram. The forms of the two diagrams depend on the class kind and are described in more details in the following sections.

`JTN` defines the operations, the methods and the process behaviours inside the body diagrams by ordered lists of conditional rules, with a uniform presentation, just a general form of "guarded commands". The alternatives of a command are evaluated in order and the first one having a satisfied guard is chosen. The guards are partly realized by pattern matching (as in ML [HMM86]) over the parameters of the operation, of the method call and over the state of the process respectively and partly by a boolean condition.

Several boolean conditions can be associated with the same pattern expression: again, they are evaluated in the order they are listed. The use of pattern matching is useful to make shorter and more readable the definitions. Let us remark that we avoid problems with overlapping patterns and conditions by explicitly ordering the guards.

In an OO perspective, stand-alone classes are not so meaningful: most of them are related to accomplish more complex functionalities. Thus we complete the class descriptions with the relevant relations among them, that are:

Inheritance relates classes of the same kind, it states that a class is a specialization or an extension of another one.

Usage relates a datatype with another class (of any kind); it states that a class uses the data defined by a datatype.

Clientship relates an object and a process class; it states that the process class assumes the existence of the object class, as it can use its methods.

All the relations are graphically rendered by lines of different types that connect the diagrams of the two related classes.

2.1.0.0.4 Architecture Diagram The architecture diagram describes the kind and the number of the system components, and how they interact (by which communication channels and by which method calls). It is represented by an hyper-graph whose nodes are the class instances that models the system components, while the hyper-arcs give a complete description of the way the instances cooperate.

2.1.0.0.5 Sequence Diagram A sequence diagram is a particular form of message sequence chart (see [ITU93]) that describes a sequence of actions occurring in a (possibly partial) execution of the system and involving some components. The represented actions are communications over channels (also with the external world) and method calls.

The class diagram (with the possibly separated body diagrams) and the architecture diagram fully describe a system. The sequence diagrams are an additional way to present information on the system that is very intuitive and easy to be understood: they are used

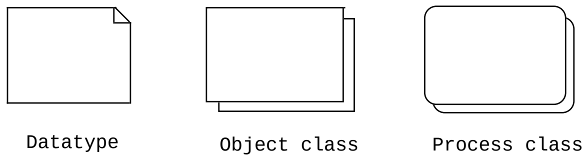


Figure 2.1: The class icons

in the most widely accepted specification techniques in the field of Software Engineering, such as UML.

2.2 Class Diagram

The class diagram describes all the classes of the specified system.

We consider three different kinds of classes: *datatypes*, *object* (for passive components) and *process classes* (for active components), graphically represented by different icons (see fig. 2.1).

The data-types are described at an abstract level by giving their constructors and by defining the associated operations, without considering an OO perspective; however it is easy to transform them into Java classes.

The passive components are seen as objects, whose state is given by a set of typed fields and the operations to modify them are methods. The transformation of such object specifications into Java classes is immediate.

The active components are seen as processes, with a state, an independent activity and communication channels to interact with other active components and with the external world. In this case, the natural implementation is given by Java threads. Communications among processes could be via channels, rather than via method calls. In this way, a process does not offer methods outside and the user does not have to specify how to manage method calls while the process is performing its activity (in Java, there is no built-in mechanism to disable method calls).

Sometimes it is useful to split the description of a class into two diagrams (the *interface* and the *body* diagrams), in order to keep separate the representation of the services supplied outside from their effective definition and from the description of the internal structure of the class instances. All the interface diagrams are collected in the class diagram, together with the relations. The body diagrams are given apart, in an auxiliary diagram that completes the class diagram.

The generic form of both the interface and the body diagrams for datatypes, objects and processes is a sort of rectangle, divided into several slots. Each slot contains a different kind of information, i.e. the class name, the description of the internal state of its instances, the declaration of the services offered outside etc.. The slots are divided by horizontal continuous lines. Inside a slot, elements of the same kind can be separated by a dashed line. The first slot, independently from the kind of the class icon, always contains the capitalized name of the class.

The relations among different classes are depicted by lines (preferably) starting from and arriving to the two slots containing the class name in the diagrams of the two classes.

2.2.0.1 Inheritance

The inheritance relation is represented by the icon $\Longrightarrow$ that is by a double arrow from the parent class to the heir one. We admit only single inheritance, restricted, of course, to classes of the same kind. Furthermore, as we will see later, the meaning of such relation varies with the kind of classes, thus it is differently constrained.

2.2.0.2 Usage

The usage relation is represented by the icon $\longrightarrow$ that is an arrow connecting the icon of the used class with the icon of the using one. Only datatype classes can be used by other classes. The relations cannot have cycles.

A class can use values and public operations of a certain datatype only if it is in use relation with it. For simplicity, we assume the inheritance and the transitivity of the usage relations: a class A that inherits from or uses another class B, implicitly uses all the classes used by B.

2.2.0.3 Clientship

The clientship relation is represented by the icon $\dashrightarrow$ that is an arrow connecting the icon of an object class with the icon of a process class. A process class is client of an object one if it calls its public methods. Again, if a process class inherits from another one, it also inherits the clientship relations, that, for simplicity, are not explicitly depicted.

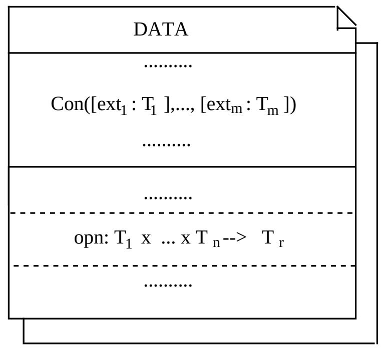


Figure 2.2: The interface diagram of a datatype

2.2.1 Datatype class

Datatypes values in a system are managed by objects and processes and exchanged by the processes with the external world. If a value is used by an object or process, it cannot be referred or modified by any other class; it can only be exchanged during a communication or returned by a method call. It is univocally determined by the way it is constructed and it lacks an internal state, thus it is completely different w.r.t. an object.

In `JTN` we specify a datatype by giving its constructors and its operations, that are described by using conditional rules with pattern matching a la ML (see [HMM86]).

The interface diagram describes what is visible to a class that uses such datatype. It contains the list of its visible constructors and operations, whereas the body diagram defines them together with the private (non visible) constructors and operations.

To give a more abstract description for a datatype, we can completely or partially hide the constructors (describing them as private, that is not visible outside the class) and omit the precise definition of the operations, that are only characterized by first-order axioms. Even if such a datatype needs to be refined by subsequent steps of the development method, it can be useful in order to give just a description of the nature of a data, omitting, for the moment, the design choices for the implementation of its operations. Thus, `JTN` supports the specification of abstract data types a la ML, even if they are not treated in the context of this thesis.

2.2.1.1 Interface diagram

The interface diagram (see fig.2.2) contains the *name* of the class, the *constructors* and the *operations* of the datatype, together with their functionalities.

The icon contains three slots, separated by continuous lines: the first one for the name of the class, the second one for the declaration of the (public) constructors and the last one for the (public) operations. Each operation inside a slot is separated by a dashed line (the separation of the constructors is optional).

If a datatype lacks constructors or no public operations, the corresponding slot is empty. It is reasonable if it inherits them from some other class. On converse, even if the datatype inherits from some other class, it must define some new value (public constructors) or some new services (public constructors): thus, the public constructor and the public operation slots cannot be both empty. For example, the `APPLY_LIST` class in the `Calculator` example does not define any new constructor but only a new public operation.

2.2.1.1.1 Constructor slot

All the possible values for a datatype are described by means of a set of constructors, each one with zero or more arguments. A datatype value is represented by a term of the form $Cons(\mathrm{tt}_1, \ldots, \mathrm{tt}_n)$, where $Cons$ is the name of the constructor and $\mathrm{tt}_1, \ldots \mathrm{tt}_n$ are terms describing the arguments. The declaration of the constructor is: $Cons(T_1, \ldots, T_n)$, where $T_1, \ldots, T_n$ are the types of the arguments. The name of a constructor has the first letter capitalized.

A constructor describes a kind of value, giving it an internal structure, that will be reflected in the translation. The arguments of the constructor describe the internal structure of a value of that kind. For example, we distinguish the empty list, specified by a 0-ary-constructor, and the non-empty ones, specified by a constructor whose arguments describe the first element and the rest of the list.

The operations used to extract components from a data constructor are declared inside the constructor itself. If we need an extractor for a component, the type of the components (say T_i) in the constructor declaration is replaced by a couple $[ext_i{:}\, T_i]$ enclosed into squared braces, where ext_i is the extractor name and T_i is the type of the corresponding extracted component. For example, $Cons([ext_1{:}\ T_1], \ldots, [ext_n{:}\ T_n])$ is just a shortcut for the constructor declaration $Cons{:}\ T_1, \ldots, T_n$ plus the declarations of the auxiliary operations (that should be placed in the slot for the operations):

$ext_1{:}\ T \rightarrow T_1$
$\ldots$
$ext_n{:}\ T \rightarrow T_n$

that return the correspondent arguments and are defined by:

$ext_i(Cons(exp_1,...,exp_n)) = exp_i$ (for $i = 1, \ldots, n$).

In the sect.2.2.1.4 we report some other equivalent, more compact notations for the constructors.

2.2.1.1.2 Operation slot

The operation slot lists the public operations defined on the datatype, that is the operations that can be used by any class that uses such datatype (that is, as we will see in the following, by each class whose icons is connected to the datatype icon by the `use` relation).

A datatype can have zero or more operations. The declaration of an operation has the generic form:

$$opn : T_1 \times \cdots \times T_n \rightarrow \ T_r$$

where *opn* is the operation name, T_i (for $i = 1, \ldots, n$) is the type of the i-th argument of the operation and T_r is the type of the returned value.

The operation name should be a string that describes the action performed on the data value when the operation is invoked. To follow a Java convention, the operation name may contain several words; every word, except the first one, has the first letter capitalized (for example, `maxQuantity` is a good name for an operation that returns the maximum among two quantities of goods).

We have chosen, for the moment, not to allow operation overloading. Thus, if the datatype class inherits an operation *op* from another one, it cannot use the same name for one of its operations.

2.2.1.2 Body diagram

The *body diagram* of a datatype (see fig.2.3) declares only the private constructors and defines all the operations of the datatype.

The declarations of the public operations are repeated w.r.t. the interface diagram; the declarations for the private ones are given in the body diagram for the first time.

The icon is divided into three slots: the first one with the datatype name, the second one with the private constructors and the third one with the definition of all (public and private) operations. The second and the third slot are divided into many parts, separated by dashed lines (optional for the constructors), containing each one the definition of a constructor or of an operation, given by conditional rules.

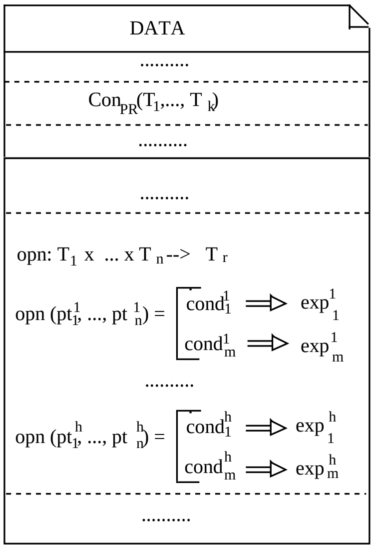


Figure 2.3: The body diagram of a data type

The functionalities of both public and private operations are given explicitly (even if, for public operations, the information is repeated w.r.t. the interface diagram): this makes easier to recognize the types of the variables representing the operation arguments.

2.2.1.2.1 Operation Definition Once we have given, by means of the constructors, a description for the values of the datatype, we can describe how to manipulate them. The operations are described by explicitly listing the interesting cases of the argument values. That is, given an operation, we consider the constructors for the types (be $T_1, \ldots, T_n$) of its arguments and we give a clause for each pattern that describes meaningful values.

The operation names follows the Java convention that they can be formed by several words and every word, except the first one, has the first letter capitalized.

We adopt the convention (rather useful to give the automatic translation of the operations into Java, as we will see in the following) that if an operation for the datatype T has at least one argument of such type, it must appears as the first element in the list of the arguments.

The definitions of the public operations precede the private ones. In the slot it is reported the operation functionality and a list of conditional rules:

opn: $T_1 \times \ldots \times T_n \rightarrow T_r$
$condrule_1$

$\dots$
$condrule_k$

each conditional rule has the form:

$$condrule_i \equiv opn(pt_1, \dots, pt_n) = \left[\begin{array}{l} cond_1 \Rightarrow exp_1 \\ \dots \\ cond_m \Rightarrow exp_m \end{array} \right.$$

where:

- $T_1, \dots, T_n$, T_r are datatypes used by class T, or T itself;
- $pt_1, \dots, pt_n$ are patterns of the corresponding sorts;

  a pattern for the datatype T_i may have forms:

  - simply a variable x of type T_i;
  - a term $con(pt_1{}', \dots, pt_h{}')$, with con a constructor for the datatype T_i whose arguments have types $T_1{}', \dots T'_h$ and each $pt_j{}'$ is a pattern of type T'_j;
- $cond_i$ is a boolean expression, that is an expression of boolean type constructed using:
  - terms of the form $bopn(exp_1{}', \dots, exp_n{}')$ with $bopn$ an operation returning a boolean value; $exp_i{}'$ is an expression constructed over the variables in the pattern expressions $pt_1, \dots, pt_n$;
  - the logic connectors `and, or, not`;
  - boolean operations over datatypes that we assume declared in `JTN`, such as natural numbers and lists (see 2.2.1.2.1 below);
- exp_j ($j = 1, \dots, n$) are expressions of type T_r constructed over: the variables in the pattern expressions $pt_1, \dots, pt_n$, the public operations defined in the interface diagrams of the other used datatypes, and the public and private operations of T (including the inherited ones).

It is not possible to use repeated variables, that is, there cannot be two or more occurrences of the same variable in the same pattern expression, because this makes more difficult to automatically check the matching of a pattern.

We write $opn(pt_1, \dots, pt_n)$ [$cond \Rightarrow exp$ if there is only one conditional rule defining an operation pattern and if there is no condition in a rule, we simply write exp instead of $true \Rightarrow exp$.

The pattern matching should be used as much as possible, avoiding to give constraint on the structure of the argument in the boolean conditions; for example, we do not write

$$opn(x)\begin{bmatrix} x{=}\mathrm{A} \Rightarrow exp_1 \\ x{=}\mathrm{B} \Rightarrow exp_2 \end{bmatrix} \qquad \text{but} \quad \begin{matrix} opn(\mathrm{A})\ [\ exp_1 \\ opn(\mathrm{B})\ [\ exp_2 \end{matrix}$$

The interpretation of a list of conditional rules is as follows. When we have to calculate the value of $opn(exp_1, \ldots ,exp_n)$, we evaluate $exp_1, \ldots , exp_n$ and then we consider, in order, the pattern expression $opn(pt_1, \ldots , pt_n)$ of each rule. When we find the first one that matches, we consider, in order again, every condition associated with such rule, until we find the first one (say $cond_i$) that is true. Then, the result is described by the expression exp_i associated with $cond_i$. If no condition is true, then the next pattern expression is examined.

Let us point out that the `JTN` operations are partial. If an operation application does not match any conditional rule, then it is undefined. In `JTN` there is no mechanism for treating the propagation of this kind of error.

It is obvious that, for the description of an operation, the order of the rules is of a paramount importance. It is possible that some rule cannot be ever used, because, for example, no term can match the patterns of the arguments, or a boolean condition can not be satisfied; the specification can also be incompleted, if there is no pattern that can match a term. If no matching pattern is found, the value of the operation application is undefined.

In the last rule defining an operation, we can use the special element `else` instead of the pattern: `else` [*exp* captures every argument that has no match with the previous patterns; the pattern has no condition associated (thus the $\Rightarrow$ symbol can be omitted) but only an expression *exp* that ensures that the value of the operation is always defined; `else` is just a shortcut for the most general pattern $opn(x_1, \ldots ,x_n)$.

Notice that we can eliminate a boolean expression if it results from the negation of the other expressions that precede it in the ordering.

A future extension of `JTN`(see Chap.8) should enrich the notation in order to give more abstract definitions for operations over (abstract) datatypes; in such cases, the list of the constructors, thus the structure of the values assumed by the arguments, is not explicitly given, and it is not possible to define the operations using pattern matching; the operation can be defined by first-order axioms given with an intuitive syntax.

Declared datatypes

`JTN` supplies some predefined datatypes, in order to avoid that the users have to give the description of well-known and widely used datatypes, such as integers or lists. Thus, the most common ones, either basic (as INT) or parametric (as LIST), are predefined and implicitly used by the other classes, so we do not report them in the class diagram, or we just put a datatype icon with the slot for the name. This allows a great simplification of the class diagram.

A user can also define a datatype as a combination of some other predefined ones, possibly with some renaming. Thus JTN allows to describe a datatype by an expression with one of the following forms:

- $T(T_1, \ldots, T_n)$: parametric datatype instantiation.
  A data type can be obtained by instantiating a parametric predefined datatype with a list of pre-defined or user-defined datatypes. An instantiated parametric datatype is simply represented by an expression $T = T'(T_1, \ldots, T_n)$ where T is the new name of the instantiated type, T' is the name of the parametric class and $T_1, \ldots, T_n$ are the parameter types.

- $T_1 + T_2$: union of two datatypes.
  A datatype can be defined as the union of other predefined ones: $T = T_1 + T_2$, where T is the name of the datatype whose values are the values of type T_1 plus the value of type T_2; the operations over values of type T_1 (T_2) are the ones defined in the diagram of T_1 (T_2), There are two implicitly defined operations:

  IsT_1: $T \rightarrow BOOL$

  IsT_2: $T \rightarrow BOOL$

  that recognize if a T value is a value of type T_1 or T_2, respectively (see, for example, the *IsDigit* operation used by the keyboard in the behaviour diagram in C.3).

- *PD*: a predefined JTN datatype that can be renamed by a new one; some predefined data are the primitive Java datatypes (int, char, and bool), the other ones are some commonly used data types: Set, List, Ordered_Lists, Stack, Map...

  A renamed datatype can just use a new name to denote a predefined datatype; thus its definition does not require to introduce new constructors or operations and can be given by using just one icon (eliminating the distinction among interface and body diagrams). The only icon for the renamed datatype contains the name of the data and the expression describing it, separated by the = symbol. See, for example, the definition of the APPLY_LIST datatype in the example, that is declared to be a new name for the lists of digit, where the parametric type *LIST(T)* is a predefined type of JTN.

From now on, we call them *declared* datatypes.

If a declared datatype does not add any new constructor or operation, it is represented by one diagram with just one slot containing the datatype name and its defining expression.

Otherwise, if the declared datatype adds new constructors and operations, the interface diagram contains a slot for the datatype name and the defining expression, plus a slot for the new public constructors and a slot for the new public operations.

The body diagram has three slots again: one for the datatype name and the expression, one for the new private constructors and the last one for the declaration and the definition of both the public and private new operations.

2.2.1.3 Use and Inheritance relations on datatypes

2.2.1.3.1 Inheritance relation The inheritance relation between two datatypes can be used to add constructors and/or new operations to an already defined datatype.

If a datatype B inherits from A, all the constructors of class A becomes also constructors of class B. All the operations in A that manipulates values of class A becomes operations with the same name and definition, but operating over B values. Notice that the inheritance always introduces a new type of data. It is not possible to simply add some operations over an existing type. For example, if B inherits from A and does not define new constructors but only an operation:

$$Dummy : A \rightarrow B$$

we have a new type B of data, whose values are the ones of A. If an object or a process class wants to use the *Dummy* operation, it must use both class B (to have *Dummy* and B) and A (to have the values of type A).

The overloading is not admitted, thus we cannot declare a new operation in the heir diagram, with the same name of an inherited operation.

To maintain the `JTN` specification tightly connected to the Java language and to make the translation more natural, multiple inheritance is not admitted. Inheritance of B from A describes just an enrichment of class A by the constructors and the operations described in class B.

Notice that, if we add new constructors and, thus, enrich the range of possible values for a datatype, the inherited operations, defined by cases on the "old" values, could be undefined. There is no problem every time a class B inherits from A and does not change the values but only define new operations. Otherwise, if B defines new constructors, there are two possible cases w.r.t. the definitions of an inherited operation *opn*: in the definition of *opn* there is an `else` clause or a clause with a generic pattern that can be applied also to the term obtained by the new constructor; otherwise, there is no pattern in the old definition that matches the terms describing the new values, thus the operation is undefined on them.

2.2.1.3.2 Use relation A datatype B can use another datatype A, that is in the constructors and in the operations of class B can appear arguments of type A. The arrow

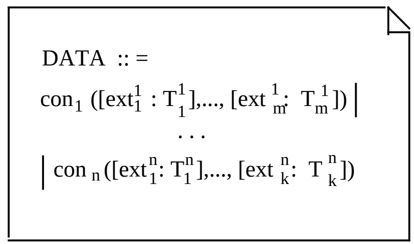


Figure 2.4: The interface diagram for a datatype with no operation

describing the usage relation connects the slot containing the name of class B to the corresponding slot of class A.

2.2.1.4 Presentation Options

2.2.1.4.1 Elimination of the body diagram When a datatype has a simple specification, it could be better to summarize the interface and the body diagrams into just one diagram, where we assume that the appearing constructors and operations are public.

In a datatype specification we can distinguish which are the the private constructors by observing which constructors appears only in the body diagram and not in the interface. Analogously, the private operations are the ones appearing only in the body diagram. Thus, if a datatype has only public constructors and operations, we can just give the body diagram, eliminating the redundant information supplied by the interface diagram.

If the datatype lacks public and private operations (i.e., an enumeration datatype), the interface diagram consists of a single slot, in which we give the datatype name and the constructors. The name is followed by the ::= symbol and by the list of the constructors, each one separated by the | symbol (see fig. 2.2).

An enumeration datatype is just a particular case of datatype with only 0-ary constructors.

Otherwise, if the datatype has also some private constructors or operations, we can still use one single diagram if it is small and readable, by adding them at the bottom of the diagram, in dotted rectangle appended to the interface diagram, as in fig. 2.5. See, for example, the `APPLY_LIST` datatype, that defines only one public operation. The auxiliary operation, not used outside the class, are declared and defined at the bottom of the diagram).

2.2.1.4.2 Groups of declared datatypes

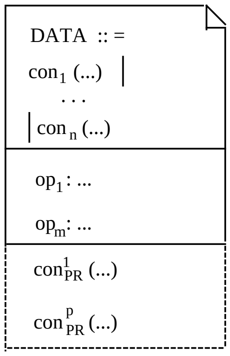


Figure 2.5: Another shortcut for the datatype diagrams

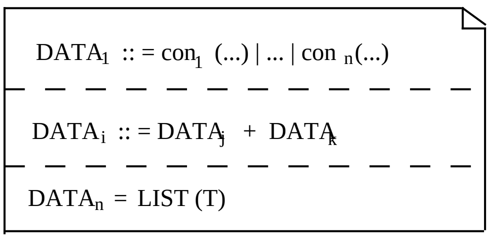


Figure 2.6: The diagram of a data group

Many different declared datatypes can be grouped together if they are used together. The unique diagram (collective diagram) is a shortcut for several diagrams related by the `use` relation.

If the resulting diagram is still space-consuming, we can give it a name (a collective name) and in the class diagram we just put a diagram containing such name. The collective diagram is given separately.

See, for example, the icon for basic datatypes in the `Calculator` example. The diagram use a single name to denote a set of simple datatypes, i.e. datatypes resulting by the renaming of other ones, or enumerations of values or simply a list of constructors. In most of the cases, there are not neither private constructors nor operations to be defined, thus the body diagram is not given.

The only (interface) diagram is constructed by putting together the datatypes declaration, separated, if necessary for the clarity, by continuous lines.

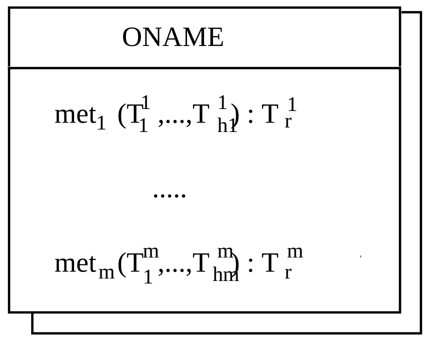


Figure 2.7: The interface diagram of an object class

2.2.1.4.3 Elimination of the usage relation If a datatype or a group of datatypes is used by every object and process class of the system, we can implicitly assume the usage relations, without depicting them.

2.2.2 Object classes

The information about an object class is usually also split into two diagrams: the *interface* and the *body* diagrams.

We consider "encapsulated" objects, that offer services outside but whose internal structure cannot be freely accessed. Thus the object state is described by fields that appear in the body diagram but not in the interface one. The value of the fields can be managed outside the object class only by public methods.

2.2.2.1 Interface Diagram

The object interface diagram is divided into two slots, separated by continuous lines (see fig.2.7). The first one contains the class name, in capitals. The second one contains the declarations of the public methods of the class, separated by dashed lines. For convention, the names of the methods have the first letter capitalized.

A method declaration has the form:

$$met(T_1, \ldots, T_n) : T_r$$

where *met* is the name of the method, $T_1, \ldots T_n$ are the types of the arguments and T_r is the type of the returned value. If the method does not return any value, T_r is *void* and it

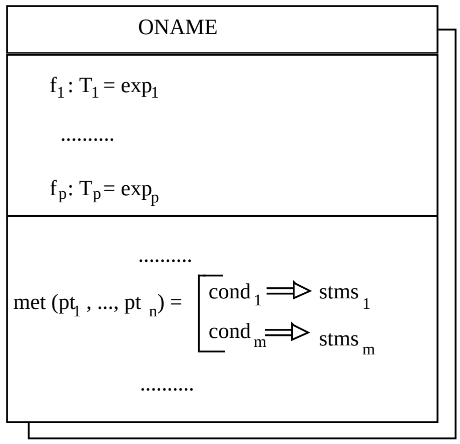


Figure 2.8: The body diagram of an object class

is skipped, thus the declaration has the form:

$$met(T_1, \ldots, T_n)$$

Notice that the list of the argument types can also be empty.

The method name should be a string that describes the effect or the use of the method when invoked on the object. It follows the same convention of the operation names: it may contain several words and every word, except the first one, has the first letter capitalized.

Again, we do not allow method overloading.

2.2.2.2 Body Diagram

The body diagram of an object class declares the object fields and defines all the methods, both private and public. The declarations of the public methods are repeated w.r.t. the interface diagram.

The icon of the object body diagram (see fig. 2.8) is a simple rectangle divided into three slots, separated by continuous lines: the first one cannot be empty and contains the name of the class; the second one can be empty and contains the list of the field declarations; the last one contains the definition of both the private and the public methods. The public methods precede the private ones and each definition is separated by a dashed line.

`JTN` notation does not support *class methods* and *class fields*.

An object class by itself describes instances having an internal state (thus, fields) and methods to manipulate them, thus it is meaningful that the second or the third slots are empty only in a class that inherits from another one.

2.2.2.2.1 Field slot A field declaration has the form:

$$f\colon T = exp$$

where T is the type of the field named f, and exp is an (optional) expression whose value denotes the initial value of the field. Let us recall that, for the moment, `JTN` does not support sub-objects, that is objects created and used by other objects. Thus, the type of any field must be a datatype, in usage relation with the object class.

The initialization expression denotes a datatype value, thus it is formed by using the datatype constructors and operations: it cannot contain neither method calls nor variables.

An example of field declaration is the field `Cont` of type `APPLY_LIST` in the class `DISPLAY`, depicted in fig.C.2.

A field declaration introduces a private field, that is a field that cannot be accessed (that is read or written) by any other class. If we want to allow another class to modify its value, we must explicitly provide a public method that manipulates the field content (see methods `Read` and `Write` in class `DISPLAY`).

The order of the fields is not meaningful, whereas their names must be different.

2.2.2.2.2 Method slot The slot for the methods in the body diagram is divided into sub-slots, containing each one a method definition. The order of the definitions is not meaningful, even if, for clarity, it should be better not to mix together the definitions of the private and the public methods. As for the operations in the datatype body diagram, a method is defined by reporting its declaration (according to the interface diagram) plus a list of conditional rules.

The signature of the method reported in the body diagram is enriched by explicitly attaching the names of the formal parameters (used in the definitions) to their sorts, separated by colons. For example, we write:

$$met(x_1 : T_1, \ldots, x_n : T_n) : T_r$$

instead of

$$met(T_1, \ldots, T_n) : T_r$$

if x_i is the variable of type T_i used in the definition of met.

Notice that, inside a method definition, all the appearing variables are not explicitly declared. Besides, each one appears in at least one of the pattern expression of the method arguments, thus their types can be easily deduced: declaring them in the method signature avoids the effort due to a type check them in the definition.

We explicitly list the different relevant cases (called *method cases*), expressed by patterns. For each sequence of patterns describing the argument passed in a single method invocation, there is a list of couples, formed by a condition and an action list, called a *conditional statement*: the condition and the statements are separated by an $\Rightarrow$.

Any method case for the method $met(T_1, \ldots, T_n)$: T_r has the form:

$$met(pt_1, \ldots, pt_n) = \left[\begin{array}{l} cond_1 \Rightarrow stms_1 \\ \ldots \\ cond_m \Rightarrow stms_m \end{array} \right.$$

Again, the $\Rightarrow$ is omitted if there is no condition associated with an action list. We write $met(pt_1, \ldots, pt_n) = cond \Rightarrow stms$ if there is just one conditional statement for the method pattern expression.

The conditions are boolean expressions as defined in sect.2.2.1, using also field names and methods of the class, returning a boolean value.

The possible forms of the statements are:

- $f = exp$;
- `res`$= exp$;
- $x = exp$;
- $met(exp_1, \ldots exp_n)$;

where x is the name of a formal parameters in the method definition, f is the name of a field of the class, `res` is a special variable whose value is the value returned by the method invocation, met is the name of a method defined in the same class, exp, exp_1, $\ldots$ exp_n are expressions constructed over the variables in $pt_1, \ldots, pt_n$, the field names, the operations of the used datatypes and the public and private methods of the object class.

Let us point out that also methods can be partial in `JTN`.

If the return type of the method is not *void*, the last action of the list must be $= exp$. No other assignment in the list can have such form. If the `res` assignment is omitted, the method call is undefined.

The order of the rules is very important. It is possible, as for the operation definitions, that some rule cannot be ever used, because no term can match the patterns of the arguments, or a boolean condition cannot be satisfied; the specification can also be incompleted, if there is no pattern for a certain term $met(exp_1, \ldots, exp_n)$: the method call result undefined. Again, we can use the special element `else` instead of the pattern: we write `else`= *stms* if the method definition of every method call that does not match with the previous patterns is given by *stms*; the rule has no condition associated, thus the $\Rightarrow$ symbol can be omitted: `else` is just a shortcut for the most general pattern $met(x_1, \ldots, x_n)$.

2.2.2.2.3 Relations

The interface diagram of an object class may be connected to the diagrams of the other classes by some arrows, representing relations:

- `use`: we can put icons for the usage relations that start from the interface diagrams of some datatype and end in the object interface diagram. All the datatypes that appear in the field or in the method declarations of the object class must be related to the object class by one of these relation icons.

- `inheritance`: the inheritance relation over object classes follows the well-accepted notion of inheritance in OO languages: if object class B inherits from object class A (graphically described by an icon starting from the interface diagram for B and ending on the interface diagram for A), the Java program defines a new class B that has all the fields and the public and private operations of A, plus the fields and operations explicitly introduced by the body and the interface diagrams of B.

  Neither method overloading nor field hiding are admitted: thus it is important to avoid name clash when describing the fields or the methods of a class that inherits from another one.

- `clientship`: if a `clientship` relation connects a process with an object class, this means that the process class uses some services supplied by the object class or use the value of some fields (it the object class supplies the corresponding public methods to access them).

2.2.2.3 Presentation options

In some cases, if it is not too space-consuming and if the class diagram is still compact and readable, we can give the description of the class by a single diagram that describes the state of the class instances and the services it provides to the outside world.

In such case the only diagram is the body diagram. The private methods are reported in a dotted slot at the bottom of the diagram (as for the datatype, see section above), in order to distinguish them from the public ones.

2.2.2.4 Parameterized object class

If all the instances of an object class O needs a particular instance (p) of the datatype *PAR_DATA*, that is an instance that is neither used nor modified by other objects or processes, and if every instance of O has its own instance of *PAR_DATA*, we can parameterize such instance on the value of *PAR_DATA*.

The value of the parameter is treated as a constant in the object instance and we can refer it in the whole behaviour diagram with the name (p) used in the interface diagram.

A datatype class *PAR_DATA* that is used to define a parameter of the instances of O is called a *parameter* class and its interface diagram is put next to the one of O. The datatype diagram contains one slot with the name of the instance and the datatype name (e.g.: p:*PAR_DATA*), separated by colons. In the other two slots, as for the usual datatype interface diagram, we declare constructors and operations, with possibly some relations.

2.2.3 Process Class

In `JTN`, processes are different from objects and their descriptions cannot be given in the same way. First, note that processes are *active* components that behave independently and do not offer methods outside. Their behaviours are not sequential, instead they run concurrently, call object methods and cooperate by message exchange with the other processes and the external world. So, the process interface diagram does not contain methods, but the names of the objects and of the communication channels, synchronous and asynchronous, used by the class.

The body diagram describes the behaviour of the process, by presenting its interesting intermediate states (each one characterized by a constructor name and typed parameters), and its transitions.

Notice that the constructors arguments are datatype values, not objects; we said above that values can be created or received during communications and are maintained into the process state: no other process can use them. An object, instead, is an entity with a state; it remains "external" w.r.t. the process, that can modify it only by calling one of its methods: thus an object cannot be part of the process state.

The behaviour is represented by a labeled graph, that gives an intuitive presentation of the alternatives that a process has in every state, that is a description of the ways it can change

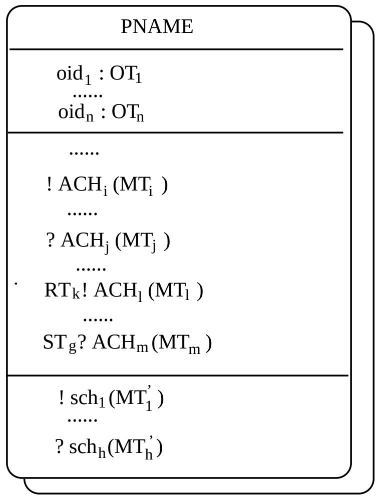


Figure 2.9: The interface diagram of a process class

its state by performing some interactions with the external world.

Precisely, from every intermediate state, some conditional rules define all the states it can reach by interacting with the external. These interactions, which are just described by labels, are communications performed by means of object method calls or by communication channels.

We give a graphic presentation of the behaviour that naturally depicts what a process does, by showing all the possible transitions starting from any state.

2.2.3.1 Interface diagram

The interface diagram (see fig.2.9) is divided into four slots: the first one for the class name, the second one for the names and the types of the object instances whose methods are called by the process, the third and the fourth ones for the declaration of the channels used to exchange messages with the external environment or with other processes. Precisely, `JTN` processes use two kinds of channels, *asynchronous* and *synchronous*, that are declared into two separate, contiguous slots: in the third slot there are the synchronous channels, in the fourth one the asynchronous ones.

2.2.3.1.1 Object reference slot

The syntax of the object reference declaration is: *oid*: *O* where *oid* is the object name and

O is its class. Let us point out that *oid* is just a *local* name, not the name of a precise instance of class O. The mapping between local names and object instances is described in the architecture diagram (see Sect. 2.3). All the names appearing in this section must be different. The object classes must be in clientship relation with the process class and there should be at least one declaration for each object class in `clientship` relation with the process class. The object declaration slot can be empty.

When a slot contains just one instance of an object class, the instance name is the class name (see the `DISPLAY`instance in the architecture diagram of fig.C.7). The clientship relation gives the information that the process class uses such object; in the process behaviour description it is used the object class name to refer to such instance. See, for example, the processes in the `Calculator` example that use the unique `DISPLAY` instance.

2.2.3.1.2 Asynchronous channel declaration slot

Every channel declaration contains: the direction of the communication over the channel (input or output), the name of the channel and the type of the exchanged messages. Thus the forms are:

! $ach\_id(T)$ and ? $ach\_id(T)$

where ach_id is a name for an *asynchronous* channel, ! and ? denote, respectively, output and input channels, T is the type of the messages, that can be omitted in the synchronous channels used to exchange signals. The datatype classes of the messages must be in `use` relation with the process class.

The names of the asynchronous channels are in capitals.

The asynchronous channel could also be used for a *broadcast communication*: the name of a broadcast asynchronous channel has an capital B as the first letter of the name.

2.2.3.1.3 Synchronous channel declaration slot

Every channel declaration contains: the direction of the communication over the channel (input or output), the name of the channel and the type of the exchanged messages: if it is omitted, the channel is used just to synchronize by means of signals. Thus the forms are:

! $sch\_id(T)$ and ? $sch\_id(T)$

where sch_id is the name for a *synchronous* one; the ! and ? have the same meaning of the slot above, T is the type of the messages (that can be omitted). The datatype classes of the messages must be in `use` relation with the process class.

The names of the synchronous channels are in lower case.

In the interface diagram of the `DISPLAY_DRIVER` there is an asynchronous channel (OUT) used to send digits outside the system and another (REFRESH) to send a refresh signal to the physical display outside. The synchronous channels `Show` and `Ok` are used, respectively, to receive a signal from the `COMPUTING_UNIT` and to send a re-start signal to the `COMPUTING_UNIT`or to the `KEYBOARD_DRIVER`.

2.2.3.1.4 Named communication

`JTN` offers another kind of asynchronous communication: by using named channels. The underlying idea is simple: every communication activity contains the message, the name of the channel and the name of the other process that is the sender or the receiver of the message. Thus, we can explicitly indicate which is the instance of a given process class we want to communicate with.

The channels declaration in the interface diagram is slightly modified by adding the process class of the receivers or of the senders (depending on the nature of the declared channel); the new form is : $T_{rec}!\ ach\_id(T)$ and $T_{snd}?\ ach\_id(T)$.

A channel for named communication must be attached to another "matching" channel for named communications. That is, if the process class A declares an output channel OUT_B for named communications with process of class B, in the interface diagram for class B there must be a input channel IN_A for named communication with process of class A, and these two channels are connected in the architecture diagram.

The mechanism of the communication is as follows: if pid_A and pid_B are instances of class A and B, pid_A can send a message $pid_B!$ OUT_B(msg) into the OUT_B channel that will be sent to the message receiver, that is pid_B.

When using a named input communication in the behaviour diagram, it is also possible to explicitly indicate the name of the sender of the message: if in the behaviour diagram of class B there is a transition labeled by $pid_A?$ IN_A(m), pid_B can only perform the transition if it receives a message from pid_A. Instead, if we use a label, x ? IN_A (msg), we describe a transition capability to receive a message sent by any process of class A.

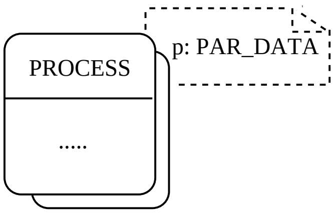
All the names in the interface (both in the object and in the channel declaration slots) must be different and cannot override the name of some inherited objects or channels.

2.2.3.2 Parameterized process class

Process classes can be parameterized over datatype values, in a way similar to the object classes (see above). For example, if a process of class P uses an internal database that cannot be modified by any other process but it, it is not meaningful to declare such database in the same way of the other datatype that can be used by several classes. Furthermore, we want to stress that every single instance of P has its own database instance.

Again, the value of the parameter is treated as a constant in the P instance and it is referred in the whole behaviour diagram of P with the name (p) used in the interface diagram.

The interface diagram of the parameter class is put aside to the one of P.



Obviously, there cannot be usage relations connecting the *PAR_DATA* diagram to other process classes, but *PAR_DATA* can use other datatypes and/or inherit from other datatypes. The clientship relation with the parametric process class is obvious and omitted.

The body diagram is the usual one for the datatype.

2.2.3.3 Behaviour Diagram

The specification of a process is given by means of the description of its intermediate states and of the way it can move from one state to another. This transitions can be merely internal otherwise can be caused by (or can have effect over) the external environment, thus are equipped with the descriptions of these interactions.

The visual presentation of a transition is an arrow connecting the initial and the final state. Such arrow is decorated by the (possible) interaction with the external; the internal transitions have a special label (τ) that is usually omitted.

The behaviour diagram is divided into three slots: the first one contains the name of the process class; the second one contains the description of the behaviour of the process instances (given by a sort of graph, as we will see in the following) and cannot be empty; the last one contains the (optional) declarations of the variable used in the graph and can be empty.

2.2.3.3.1 Graph structure

The *behaviour* of a process of a class is described by giving all the possible alternatives that it has in every intermediate state.

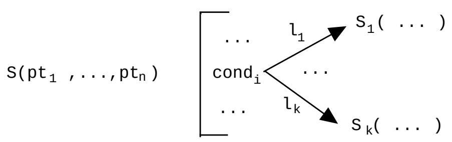
It is rendered by a graph, whose arcs represent "generic" labeled transitions, whose form is: $S(pt_1, \dots, pt_n) \xrightarrow{l} S'(exp_1, \dots, exp_m)$

where S, S' are state constructors, pt_1, ... ,pt_n are patterns for the state parameters, l is a pattern for the external interaction and exp_1, ... ,exp_m are expressions over the variables in l and pt_1, ... ,pt_n.

All the transition sources and targets that represent state patterns of the same kind, that is with the same state constructor, are grouped together into a node of the behaviour graph (called a *state case*), avoiding repetitions.

A *state case* is a sequence of *transition cases*, that is of pattern expressions each one with an associated list of conditioned transitions. A state case describes the transition capabilities of the process when its internal state matches a certain pattern.

Thus each state pattern is written once, and the conditions are listed aside the corresponding state pattern to form an *transition case*, as below



The arrows leaving a condition are ordered. Thus, inside a node of the graph we have an ordered list of transition cases plus state patterns that are only targets of transitions (called *arriving cases*).

To increase the graph readability, it is better to keep separate, inside the node, the two kinds of state patterns: the transition cases are order, thus they should be presented in sequence and they should not be mixed with the terms describing arriving cases of the transitions.

Conditions are boolean expressions over the variables in the pattern expressions. The boolean expressions are the ones described before for the definition of the operations over datatypes (methods are not admitted).

Moreover, in case of processes parameterized over a datatype value, the patterns, the conditions and the labels of the alternatives can contain the name of the parameter, as declared in the class diagram.

The interpretation of the behaviour graph is as follows. When a process is in a state $S(exp_1$, ... ,$exp_n)$, we consider, in order, all the alternatives inside the node for the states constructed with S, until we find one whose pattern matches exp_1, ... ,exp_n. Then, inside the chosen alternative, we look for the first true condition. Finally, we consider the labels on the arrows leaving the condition, trying to determine the first one that can be executed (recall they are ordered). If no matching pattern with a true condition is found, then the process is definitively stopped.

2.2.3.3.2 State constructors The meaningful intermediate cases of a process life are described by terms over constructors. The constructor names should be evocative and give a little flavor of the internal state or of the future activity of the system. Like the datatype constructors, the names have the first letter capitalized.

We can omit the name of a state every time it is not relevant (we talk about *anonymous* states of the diagram); in such a case, the icon is empty, or just contains the list of the arguments (examples of anonymous states are in the behaviour graph of the `PRINTER_DRIVER`, in figC.6).

Every process class has an initial state, described by a term in the node of the `initial` constructor (`initial` is a reserved word that cannot be used for an intermediate state of the process behaviour). Notice that in such node there usually are several terms. The one describing the process starting state is individuated by an incoming arrow with neither label nor starting state (see the upper left arrow in the node of the `initial` state in the behaviour diagram of the ComputingUnit, for example).

We can improve the readability of the behaviour diagram, by disposing the patterns with exiting arrows separated (by a dotted line) from the pattern with entering arrows. This allows to individuate which are the transition capabilities of the system in each state.

Sometimes, on the contrary, the same pattern has incoming and outcoming arrows: obviously we do not replicate the term in the node (see the Cancel state in the behaviour graph of the `DISPLAY_DRIVER`)

In `JTN` it is not required to explicitly declare the state constructors; however in many cases (for example if the graph is splitted into several diagrams)it is not easy to deduce the type of a variable from just a fragment of the behaviour graph, thus it is useful to declare, in the third slot of the behaviour diagram, the type of the used variables.

The label l can have six different forms, depending on the kind of interaction that is performed with the external world:

1. $?ach\_id(x)$, $!ach\_id(exp)$: input from/output to an asynchronous channel;
2. $exp_{snd}?ach\_id(x)$, $exp_{rec}!ach\_id(exp)$: input from/output to an asynchronous channel, with sender/receiver denoted by the expressions exp_{snd} and exp_{rec};
3. $x = oid.met(x_1, \ldots, x_n)$ (or $oid.met(x_1, \ldots, x_n)$ if met has no return type): object method invocation;
4. $?sch\_id(x)$,$!sch\_id(exp)$: input from/output to a synchronous channel;

5. $?sch\_id()$,$!sch\_id()$: input from/output of a signal on a synchronous channel;

6. τ: internal activity, usually omitted.

In cases 1 and 2, the types of the variable x and of the expression *exp* must be the same type of the message declared for *ach_id* in the interface diagram. Furthermore, in case 2, exp_{snd} and exp_{rec} must be expressions of type P, if *ach_id* has been declared to be channel for named communication with processes of class P.

In case 3, *oid* must be the name of an object declared in the interface diagram (or the name of an object class if there is just one instance of such class in the system), *met* the name of one of its methods x_i are variables whose type is according to the type of the corresponding argument.

In case 4, the type of the variable x and of the expression *exp* must be the type of the message declared for *sch_id* in the interface diagram.

In case 5, *sch_id* has to be a synchronous channel, with no message type.

Admissible label alternatives Not all choices of labels (l_1,..,l_h) are meaningful; the admissible cases are described below, and for each of them we explain how to select the one to execute:

1. h = 1 and l_1 is a *method call* or an *output* operation on an *asynchronous channel* or an *internal transition*; the corresponding transition can be executed.

2. h $\geq$ 1 and for each i (1$\leq$ i $\leq$ h) l_i = $!sch\_id_i(\ \dots)$ or l_i = $?sch\_id_i(\ \dots)$ are *communications* over *synchronous channels*. All the channels are checked in the order: if the communication on sch_id_i cannot be executed, then the following one is checked: if no communication can be executed, then the process is suspended until the last communication completes.

3. h $\geq$ 1 and for each i (1 $\leq$ i$\leq$ h) l_i = $?ach\_id_i(\ \dots)$: are *simple input operations* on the *asynchronous channels* ach_id_i that are continuously tested in the order, until an available message is found.

4. h $\geq$ 1 and for each i (1 $\leq$ i$\leq$ h) l_i = $x?ach\_id_i(\ \dots)$: are *named input operations* on the *asynchronous channels* ach_id_i that are continuously tested in the order, until an available message is found.

In cases 2, 3 and 4 we can add an arrow labeled with `else` with the meaning that, every time no other transition can be executed, such escape will be performed as an internal action, leading to another state from which the activity continues. An else label can be used in case 2 when no communication is immediately available, to return to the same state and start again the polling procedure. See the state *Init* in fig. C.6 of the `Calculator` example, where the `PRINTER_DRIVER` can perform a synchronous communication on channel `Show` otherwise on `NEW_LINE`: if none can be immediately completed, it can return to the same state by performing the else transition, thus the synchronous communications are non-blocking and it can start again checking the two channels in the order.

Obviously, the `else` transition is not used to return to the same state in case of multiple choices over asynchronous channels: if all the channels are not ready to complete the communication, when the last one has been tested, the polling procedure starts again from the first channel, with no need to returns back to the first state using the `else` alternative. `else` is used only if we want to perform the polling just one time and then, if no communication succeed, pass into another state.

2.2.3.4 Inheritance of process classes

Inheritance over process classes assumes a different meaning w.r.t. objects. Since processes do not offer services (that is operation or methods) neither outside nor, obviously, to an heir class, we think that a class of processes can extend another one by extending the behaviour: it can introduce new transitions and, thus, new states, new object references and new interactions used for the new interactions.

In our example, the `PRINTER_DRIVER` class inherits from `DISPLAY_DRIVER`: its interface diagram is the one of `DISPLAY_DRIVER` with a new synchronous channel `Newline` and the asynchronous one `NEW_LINE`; the behaviour graph of class `PRINTER_DRIVER`depicts only the new transitions (see fig. C.6), implicitly assuming those described in the behaviour graph of `DISPLAY_DRIVER`. The `PRINTER_DRIVER` behaves like the `DISPLAY_DRIVER`when it receives a message on the `Show` channel (that is, when it is required to show an the display the `DISPLAY` content). But it also has the additional capability to receive a new-line command on the NewLine channel. Notice that the inherited `OUT` channel is renamed into `OUT_P` and it is connected to the *PRINTER* external component.

Thus, if a process class extends another one, it keeps all the capabilities of the first one and adds some new ones. The new behaviour graph extends the inherited one with some transitions starting from old states (that is states described also in the behaviour graph of the inherited class). These transition does not substitute the old ones, but are interpreted as other alternatives, to be considered after the old ones.

The arriving states of the transition, on the contrary, can also be new states. Thus we can extend the graph with some completely new parts, just anchoring it to an old state. See, for example, the READY_TO_PRINT state that is reached by a transition starting from the `initial` state. Obviously, all the channels and object invoked in the new transitions have to be declared in the interface diagram of the new class.

More precisely, the interface diagram of the heir class contains, in the first slot, the class name, in the second slot the new object references, in the third and fourth ones the new channels. Obviously the names of the objects and of the channels are new and must not cause any clash with the inherited ones.

The first slot of the behaviour diagram contains the name of the new class.

The second one contains just the new part of the behaviour graph. The behaviour of the class is described by a graph obtained by adding these new capabilities to the old one, thus it can be constructed as follows (in the following the "old" states and transitions are the inherited ones in the new class): the transitions starting from states of the same kind are merged together; the new state cases for an old state, as well as the new transitions associated with an existing condition, are added at the end of the list, as they represent alternatives to be considered after the existing ones (we are currently studying more suitable mechanisms to describe how to re-order these alternatives).

The third slot of the diagram may contain the variable used in the second slot.

2.2.3.4.1 Presentation Options

The graph describing the behaviour of a complex process is often very large and tangled. We can keep it more readable if we split it into several ones, each one reporting the class name in the first slot, a part of the graph in the second one and, preferably, the variable declarations in the third one. These declarations becomes very easy when we cannot see all the graph at a glance: sometimes we have to analyse all the diagrams to deduce the type of a variable; by putting them at the bottom of the diagram increase the overall readability of the process specification.

We give some criteria to split the behaviour graph into sub-graphs:

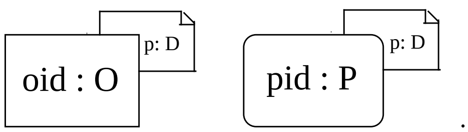
- every sub-graph completely describes the activity starting form some states (there cannot be two sub-graphs containing alternatives for the same state);
- if an activity ends in a state that is not described in the graph, the corresponding arrow end in a node, represented by a double circle, containing just the term describing the final state of the transition; these nodes cannot have outcoming arrows

2.3 Architecture Diagram

The architecture diagram describes the structure of the system by showing its components and how they cooperate.

It is represented by an hyper-graph whose nodes are the class instances modeling the system components and whose hyper-arcs represent how they cooperate.

The icons for the process and the object instances are slightly different from the corresponding ones for the classes; they are single boxes or single boxes with rounded corners:



An instance icon contains only the instance identifier with the name of the corresponding class, separated by colons; the identifier is omitted if there is only one instance for such class.

For the parametric process classes, we also indicate the value of the parameter for each instance. $pid{:}PT(p)$ denotes the instance of the parametric process of class PT whose parameter value is p.

Let us remark that, for the moment, we consider neither creation/deletion of components nor architectures having a generic number of components.

We can distinguish three kinds of hyper-arcs, representing:

- the cooperation between an object and a process that uses one or more of its methods (object-process cooperation);
- the cooperation of several processes that communicate by means of connected asynchronous channels (asynchronous cooperation);
- the cooperation of several processes that communicate by means of connected synchronous channels (synchronous cooperation);

2.3.0.4.2 Object-process cooperations

An object and a process cooperate by method calls: the process, during its activity, calls one or several public methods supplied by the object. The icons of the two instances are linked by an arrow decorated with the names of the methods and with the local name of

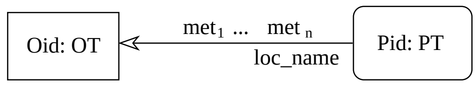


Figure 2.10: The object-process cooperation

the object, that is the name (for an object of that type) appearing in the interface diagram of the process; the arrow is oriented from the caller process to the called object (see fig. 2.10).

Let us recall that if the object instance is the only one of its class, the icon contains only the class name, with no identifier. The process interface has no reference for the object (the information id deduced by the clientship relation between the two classes), thus there is no local name in the cooperation. Thus, given a process class P with a clientship relation with object class O, we explicitly declare, for each instance *pid* of P and each instance *oid* of class O, which methods of *oid* are called by *pid*. The keyword *ALL* means that the process uses all the methods supplied by the class of the object.

2.3.0.4.3 Asynchronous cooperations

`JTN` supplies three kinds of asynchronous communication channels, involving processes as well as external entities: simple, broadcast and named channels. The architecture diagram describes how these channels are connected (by means of *connectors*), with the obvious underling constraint that a communication happens only through connected channels of the same type.

A channel is represented by a double arrow starting from or ending into a connector, labeled by the name of the channel as it appears in the interface diagram of the corresponding process. The ! or ? symbol is explicitly reported in the label, while the type of the exchanged message (that is the same for all the channels) is omitted.

Fig. 2.11 describes the generic schema of connections of some asynchronous channels: OC_i are output channels and IC_i are input channels of the processes attached to the hyper-arc. Moreover, the channel types and versus must be in accord with the interfaces of the classes of the connected processes.

The external environment is represented by a double rectangle with the name of the entity (that can also be omitted) inside.

Now let us examine, in more details, the various forms of a cooperation, distinguishing among connectors for *simple*, *broadcast* and *named* cooperations, that are graphically rendered in figure 2.11: every connector may have a name, that can be used in the sequence diagrams.

In case of connector for *simple* communications (just *simple* connector for brevity), we can distinguish some cases:

- if $n > 1$,$m = 0$ (see case 1a in figure 2.11) or $n = 0$,$m > 1$ (case 1b), then the icon connects several channels of processes that communicate with the external world (e.g., the `OUT` and the `REFRESH` channels associated with the `DISPLAY_DRIVER`or the in the `Calculator`);
- if $n > 0$,$m > 0$ (case 1c), a message sent on a generic OC_i will be consumed by one of the processes connected to the IC_1,..,IC_m channels;
- if $n = 1, m = 0$ or $n = 0, m = 1$ the connector is used to rename a channel as we can see in fig. C.7 of the `Calculator`, where channel `OUT` of `PRINTER_DRIVER`is renamed as `OUT_P` to avoid name clashes with the same channel of `DISPLAY_DRIVER`.

In case of *broadcast* connector we have two meaningful cases:

- if $m > 0, n > 1$ (case 2a) and the input channels IC_i are connected to the channels of other processes (and not to the external), then every message *msg* sent on a generic OC_i will be replied on all the IC_1,..,IC_m and every process connected to IC_i can read its own copy of the message;
- if $m = 0$ and $n > 1$, every message coming from the external word is broadcasted to all the processes using the input channels IC_i.

Finally, in case of *named* connector the communication is meaningful only among processes, thus cannot be used to communicate with the external environment. The only case we can represent is 3a) in figure: a message sent on a generic OC_i comes together with the identity of one of the processes connected to the IC_i channels and will be consumed only by that process. On the contrary, if a process *pid* is connected to one of the IC_i channels and it performs a read operation, it receives the first message sent to *pid*, together with the identity of the sender. It can also specify the identity (say pid_S) of the sender: in such case it will receive the first available message sent on OC_i by pid_S.

Let us stress that all the $OC_1, \ldots, OC_n$ and IC_1,..,IC_m channels have to be declared channels for named communications in the interface diagrams of the connected processes. Furthermore, the directions and types of the messages in the declarations must match.

2.3.0.4.4 Synchronous communication The channels are represented by single arrows, labeled, as for the asynchronous case, by the name of the channel as it appears in the

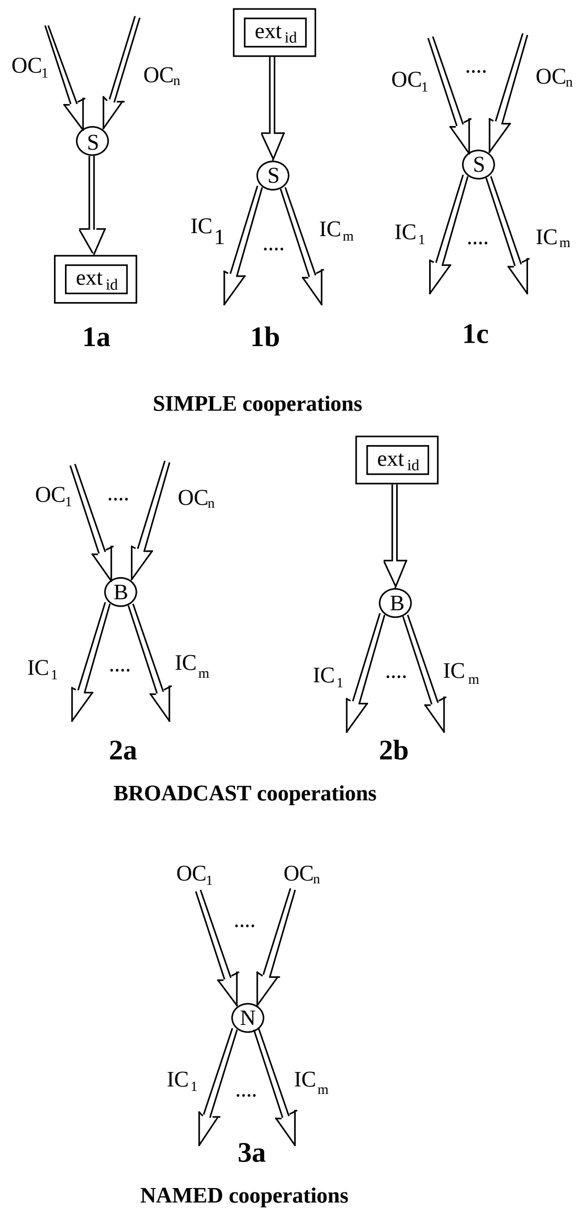


Figure 2.11: The asynchronous cooperations

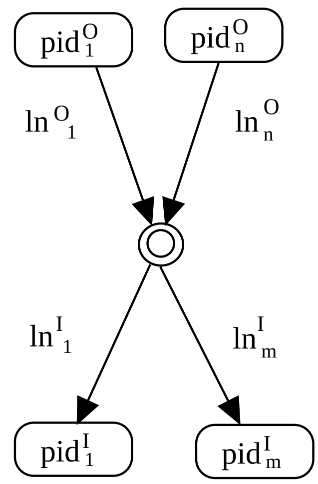


Figure 2.12: A synchronous connector

interface diagram of the corresponding process. Again, the ! or ? symbols are explicitly reported in the label, while the type of the exchanged message (that is the same for all the channels) is omitted.

The icon of the connector for the synchronous channels is a double circle.

Fig. 2.12 describes the connection of some synchronous channels. We always have $n > 0, m > 0$, because synchronous channels cannot be used for the communication with the external world. Again, the channel types and versus must be in accord with the interfaces of the classes of the connected processes.

The synchronous communication always involves two processes at a time: one process connected on a generic $\ln^O_i$ acting as a sender, and one connected on a generic $\ln^I_i$, acting as a receiver. Thus, the drawing can be interpreted as a shortcut for a set of channels connecting pairwise all sending to all receiving processes.

Notice that there is neither a mechanism to synchronize n $(n > 2)$ processes, nor to specify the identity of the partner of the synchronization.

An example is the channel Ok in fig. C.7 that connects `KEYBOARD_DRIVER`, `COMPUTING_UNIT`(acting as receivers) and the `DISPLAY_DRIVER`and the `PRINTER_DRIVER`(senders).

2.3.1 Generic architecture diagram

When specifying a generic system, that is when we give the specification of a set (possibly infinite) of systems differing only for the number of components, we need a notation for the cooperations among "generic" instances of object and process classes.

If the generic system has n instances of the same process (or object) class C, we can represent them by drawing two process (object) icons, separated by dots, with identity pid_1 (oid_1) and pid_n (oid_n). We assume that all the other $n-2$ instances have exactly the same structure and behaviour, that is, they use the same connections or the same objects of the depicted pid_1 and pid_n instances: this is a sensible assumption as they are instances of the same class.

If there is an instance of class C that differs in the way it cooperates with the other processes or with the external world, it must be explicitly drawn in the diagram. For example, if in a generic architecture we have n processes that shares the same database (be db), while another process uses another database (be db′), the process icon must appear in the architecture diagram where it is depicted its cooperation with db′.

2.3.2 Presentation options

The connector icon can be omitted in case of channels used by one process to communicate with the external. The cooperation is represented by a double arrow, labeled by the channel local name, that connects the process instances with the icon of the external entity (see, for example, the channel IN connecting the `KEYBOARD_DRIVER`with the external in C.7).

The use of channels for named communications can lead to the creation of very tangled architecture diagrams: if a process *pid* declares a channel for the communication with processes of a class P, in the architecture diagram we have to explicitly depict a channel connecting *pid* with each instance of the P class. Some shortcuts or more concise notations could be very helpful to keep the diagram readable.

For example, only the generic i-th instance of each class is represented. This notation allow to depict a connection between a channel ICH from a generic instance pid_i of class $PT$and a channel OCH to a generic instance oid_j of class OT as a shortcut for the more complex connection in which every ICH channel of an instance of a class PT is connected with the channel OCH of each instance of class PT.

2.4 Sequence Diagram

A sequence[1] diagram is a kind of Message Sequence Chart [ITU93] that gives a (possibly partial) description of a (possibly partial) execution of the system. Sequence diagrams are of particular interest because they introduce in a specification formalism a technique that is used in the most widely accepted methods and notations in the Software Engineering field (such as UML).

A sequence diagram graphically represents a partial system execution: that is some of the components taking part into it and the ordered sequence of interactions among such components as well as with the external world.

The class and the architecture diagrams supply complementary information about the system, whereas the sequence diagrams are just a different way to visualize information that has been already specified by the other diagrams. We are able to construct the sequence diagrams, once we are provided with the class and the architecture diagrams. Thus, why and when are they valuable, during a system specification development?

The answer is that sequence diagrams are not valuable for their information content (because it is already present in the other diagrams) but mainly from a methodological point of view: several sequence diagrams may be presented for the same specification e.g., to cover the description of some interesting use cases of the system. They can be used for different purposes, for instance:

- to give a more natural and clear representation of the developed system to a client (e.g., to show how the calculator performs the addition);
- to show that the behaviour of the system specified by the class and the architecture diagrams is, in particular circumstances, the expected one. In the `Calculator` example, a sequence diagram could be used to show that if the user does not digit an "=" at the end of the operation, then the calculator does not return the result. From our experience, the construction of sequence diagrams, also if "by hand", helps to control the quality of the proposed design and allows to detect errors and omissions in the specification.

In `JTN` the components that can interact are of kind object and process. Objects can interact only with processes, by receiving method calls. Processes can also interact with other processes or with the external world, by communications over channels.

Thus, the graphic presentation of a sequence diagram supplies three kinds of information:

[1]In this case we use the same terminology of UML, since our sequences and the UML ones are rather similar; we can analogously define a form of collaboration diagram.

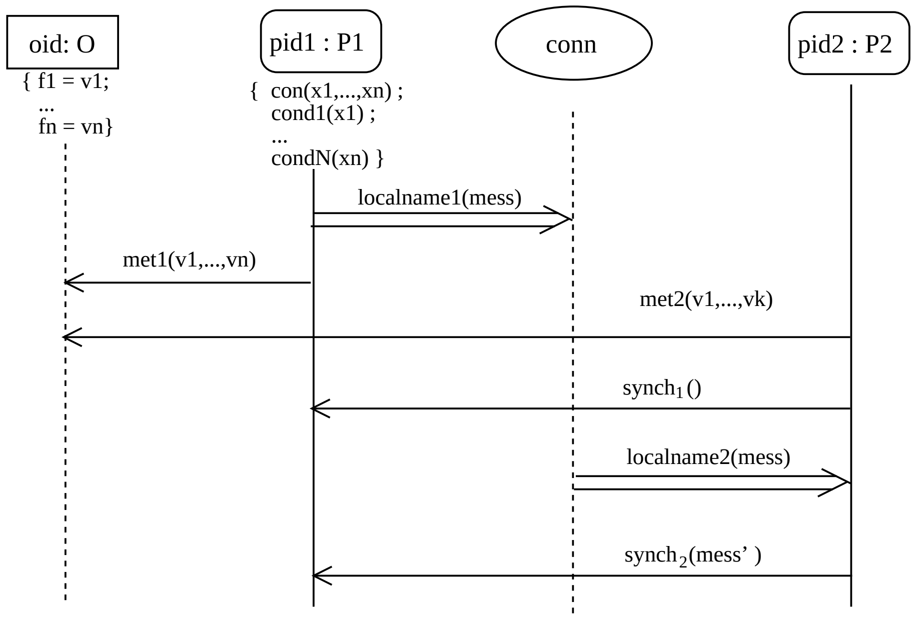


Figure 2.13: A generic sequence

- the kind, the identities and (optionally) the internal states of the components involved;
- the way they interact;
- the temporal ordering among the interaction occurrences.

Moreover, the diagram can be annotated with information about the state of the components, so it is possible to represent effects or conditions of action occurrences on single components.

2.4.1 The graphical presentation

Every sequence diagram describes the interactions among a finite number of processes and objects. The diagram contains *vertical lines* representing the lives of the components involved in the execution fragment. We use a dashed line to represent objects and a continuous line to represent processes (see fig.2.13).

At the top of each line there is an instance icon containing the name and the type (separated by colons) of the instance. Optionally, it is possible to add a description of the state of the component at the begin of the partial execution described by the diagram. The initial state can be unambiguously determined, otherwise it is just described by giving some conditions it has to satisfy.

The choice (among a complete or a partial description) mainly depends on the use of the sequence diagram: with regard to the two possible uses described in the paragraph above, in the first case we can show that, under certain circumstances, that is under certain conditions on the states of the components, the expected behaviour of the system is the one described by the sequence diagram. Thus we want to give a graphic description of many possible cases, not one in particular, just to clarify some aspects of the system behaviour.

On the other side (the second case above), if we want to use the sequence diagram to describe a precise sequence of actions/ interactions, we presumably starts with the components in a precise initial state.

The initial state of an object is described by the content of (some of) its field; or we could give a condition on such fields. For a process, the initial state can be simply a term (Initial, for example) or a conditional expression over the arguments of a state constructor (see, again, the description of the initial states in C.8).

The *horizontal lines* describe the interactions occurring among the components or with the external world (i.e., communications on a channel, and method calls). Lines are put from top to bottom with respect to the temporal ordering of happening.

We use different icons for asynchronous communications (a double arrow), synchronous communications (a single arrow) and method calls (a simple line) as we can see in fig. C.8.

2.4.1.1 Method calls

A method call is a line connecting the calling process and the called object. It is decorated by a label whose form is: $met(exp_1, \ldots, exp_n)$ or $exp = met(exp_1, \ldots, exp_n)$, where met is the method name, $exp_1, \ldots, exp_n$ are the method call parameters, and exp is the returned value.

2.4.1.2 Asynchronous communications

The asynchronous communications over a given channel need an additional dashed vertical line representing the connector because writing and reading operation of a message are not performed at the same time, thus cannot be represented by arrow connecting the sender and the receiving processes; furthermore, the two arrows describing the two separate phases of an asynchronous communication may be interleaved by other interactions.

At the top of the dashed line there is the icon representing the connector, thus a circle containing the connector kind (that can be S, B or N). In case of a connector for several process channels (each one, possibly, with a different local name) the icon contains also the connector name, that must be the same appearing in the architecture diagram. The name in the architecture diagram can be omitted if the channels is used by only one process to communicate with the external; in the sequence diagram it is used the local name of the unique input or output process channel.

Finally, if the asynchronous communication does not involve any connector but only a channel used by processes to communicate with the external environment, the icon contains the local name of the channel: see the channel OUT in the sequence diagram C.8).

For simplicity we assume that the connector always starts from a state in which there is no message inside.

An asynchronous communication with the external world is just an incoming or outcoming double arrow, labeled by the channel name and by the exchanged data. The dashed vertical line can be omitted.

An asynchronous communication between two processes is represented by two broken arrows. The part representing the start of the communication (send) is over the other one and is represented by a double arrow from the sending process to the dashed line representing the channel. The receiving of the message is represented by a double arrow from the channel to the receiving process.

The label has two forms:

- a couple (*ach_id*, *msg*) where *ach_id* is the local name of the channel, used by the sender or receiver process and *msg* is the message; the sender identity and the direction (input/output) are deduced by the arrow;
- a triple (*pid*, *ach_id*, *msg*) where *pid* is the identity of the receiver or of the sender of the message, in case of named communications;

The local names of the channels labeling the incoming and outcoming interactions of the connector must be consistent with the cooperation described in the architecture diagram.

Other actions may occur between the two phases of the asynchronous communications.

2.4.1.3 Synchronous communications

A synchronous communication is a simple arrow connecting the sending to the receiving process and it is decorated by as label whose form is: *sch_id*(m) where *sch_id* is the name of the channel (that does not appear explicitly as a component in the diagram) and m is the exchanged message. Notice that there is no message if the communication is used just as a synchronization among the processes, thus the label is just *sch_id*.

To increase the readability of the interactions, it is possible to put annotations at the bottom of the diagram. For example, if the parameters or the returned value of a method call or the term describing a message are too long expressions, they are replaced by a short term and completely described by an equation at the end of the diagram.

Let us recall that the focus is put on the interactions and on the data flow among the components and/or with the external environment. Thus, a sequence diagram does not report the internal activities and the state transformations of the components. Furthermore, not all the components are depicted in the diagram: it is possible that, between two depicted interactions, a component performs also other activities, involving components not represented in the diagram. Thus what we see in the diagram generally is not the "complete" behaviour of a component.

Note that the elimination of a component and of its interactions returns another sequence diagram.

If we drop `DISPLAY_DRIVER`in fig. C.8, then we have a sequence diagram concerning only the updating of `DISPLAY`.

2.4.1.3.1 Consistency Check As sequence diagrams can erroneously depict executions that are not coherent with the rest of the specification, we must define when a sequence diagram is consistent with the information supplied by the class and the architecture diagrams.

The consistency check can be divided into two phases. The first one is a *global* consistency check: we control if the interactions between two components are consistent with the cooperations depicted in the architecture diagram. The second one is a *local* consistency check: we control that the component state is able to perform the interaction.

2.4.1.3.2 Global consistency check

For each depicted interaction:

- if it is a *method call*, we check that the object-process interaction, by means of the particular method, is depicted in the architecture diagram;
- if it is a *synchronous* communication, labeled by *sch_id*(), we check the presence of the corresponding communication in the architecture diagram, where appears the output channel with local name *sch_id* connected to the sending process.

- if it is an *asynchronous* communication, labeled by $ach\_id()$ and involving the connector *conn*, we check the presence of *conn* in the architecture diagram, where ach_id is the local name of one of the channels used for input or output communication, depending on the direction of the arrow depicting the interaction.

2.4.1.3.3 Local consistency check

The local consistency check is aimed to control if an interaction can take place in the starting state of a component, as it results from the initial state description and after the execution of the previous interactions. Once we fixed the starting state of each instance, we can easily trace out how the system evolves, proceeding as follows.

For each object component, on the top of the corresponding line there is an assignment expression for each one of its fields. Each process component has a (ground) term describing its initial state. Each asynchronous connector has the list of the contained messages.

Now we have to calculate, for each interaction, if it can happen in the initial state and which are the changes in the state caused by the interaction execution.

The object body diagrams describe how a method execution changes the state of an object; remind that a method execution modifies only the called object.

The behaviour graphs describe which communications or which method calls a process can perform from a given state and which is the corresponding new state.

The architecture diagram presents the topology of both the external as well as the inter-process communications. Thus, when we know which are the values arriving on the input asynchronous channels from the external world, we can find which actions the connected process components can perform and consequently, we can trace out the change of state of all the depicted components.

So, given a sequence diagram, we can determine the starting state of the system and then whether the depicted interactions can happen in the depicted order.

This consistency idea is precisely defined (see Chap. 4), remembering that `JTN` is a formal specification language (the semantics of the class plus the architecture diagrams is an *lts*) and that each sequence diagram corresponds to a formula in a branching time temporal logic saying that, from the starting state there exists a sequence of transitions where the depicted communications happen in the depicted order.

Let us point out that from now on (in the next chapters also) we consider the bare core of the `JTN` notation, forgetting about some shortcuts and alternative notations presented in the Chap.2.

More precisely, we will give the static semantics, the formal semantics of Java and the translation of the JTN specifications leaving aside:

- declared datatypes;
- groups of datatypes;
- parametric process classes and parameter datatype classes.

Their exclusion is due only to simplification considerations, as their treatment do not add any significant modification to the formalization and translation of JTN.

2.5 Preprocessing

We assume that there is a preprocessing phase that, given the textual presentation of a JTN specification, returns another one (equivalent w.r.t. the semantics of JTN) with some simplifications that facilitate the static checking, the semantics definition and the translation phase. Essentially it performs some re-arrangement of the class specifications:

- ordering the datatypes in a sequence, according to the usage relations, so that a datatype specification appears after the specifications of the used datatypes.
- ordering all the datatype an object classes so that the specification of an heir class appears after the specification of the inherited one;
- removing the `inheritance` relation over the process classes; recall that its semantics in JTN is different from the one over datatypes and objects and that can only add transition capabilities to processes and, consequently, channel and object declarations, to the activity of the inherited class, without enriching the set of its constructors. In the term resulting from the preprocessing phase, if T inherits from T', the term corresponding to T is obtained by adding the new activities to the term describing T'.

  Thus, the preprocessing takes the textual presentation of T and enrich it by adding the new channels and objects names to the interface description. The term describing the T behaviour diagram is substituted by another one constructed as follows: the list of constructor declarations is the one of T' enriched with the new ones (if any); the initial state is the one in T, if any, otherwise is the one of T'; finally, the term representing the graph is the corresponding one of T', enriched by the new state cases and by the new transition cases, added at the end of the list of the corresponding state case, after the transition cases defining in the graph of T'.

Instead, inheritance over datatypes and objects adds, respectively, constructors and operations or fields and methods to an already existing class, thus it can be exactly translated by using the inheritance mechanism of Java.

Chapter 3

JTN static semantics

3.1 Introduction

The static semantics gives a formal way to statically check the correctness of the textual presentation of a JTN specification (as defined by the BNF grammar of Appendix A).

The value of this tool is that it helps in finding many errors when writing a JTN specification; furthermore, we can assume that a JTN specification is statically correct to simplify both the formal semantics (see Sect. 4) and the translation into Java (see Sect. 5.1), that become shorter and more efficient.

We define the static semantics by giving an inference system whose rules allow to assign a type to the correct (well formed) fragments of JTN specifications. Thus, we say that a fragment is correct if and only if we can assign it a type, using such inference system.

Clearly the correctness of a JTN construct depends on its context; thus the rules of the system use also additional informations describing such contexts. Precisely using the system we may prove judgments of the form

$\mathrm{I} \vdash t : T$

with the meaning that the fragment t is statically correct and has type T (for some T) in the context described by I, where I is some information about the context of t, whose precise structure depends on the nature of t.

In the system we have also rules to prove type assertions plus some new context information:

$\mathrm{I} \vdash t : T, \mathrm{I}'$.

The checking of the whole JTN specification starts with a null context information, which will be enriched as long as the other parts of the specification are examined. The semantics of a class diagram collects the names of the classes, the constructors, the operations and the methods, plus other information about the examined classes. The architecture diagram uses an information structure containing the identities of the system components and the declarations of the cooperations. This information will be passed to the semantics of the sequence diagrams.

Some constructs have an obvious type (for example, expressions and variables), whereas others do not; in such case, we assign them the type *void* (for example, whole diagrams, if correct, have the type *void*).

In `JTN` every kind of class (with some restrictions for process classes) has an inheritance relation. Inheritance is always single and involves only classes of the same kind (datatypes, objects or processes). It allows to define a new class as a specialization of an existing one. The new class can inherit constructors, operations, methods and channels and, on its turn, defines new ones. Following a well-accepted O.O. concept, the new class describes a subtype of more specialized instances, that can be used in every place where an instance of the inherited (less specialized) class is expected. Thus, if class T inherits from T', we can use an instance of T where an instance of T' is required. For example, an assignment $x = t$ is correct if the type of t is a subtype of the type of x. This observation explains the need of a subtype relation, easily deduced from the inheritance relation, to check the correctness of the assignment expressions or of the argument passing in a method call.

From now on, in this chapter we use the term semantics instead of static semantics, for simplicity, and we name "environment" the context information.

3.2 Class Diagram

3.2.1 Class diagram environment

The *class environments* (the information on the context needed to check a class diagram) are n-uplas of sets (containing class names, relations among classes, definitions of constructors, operations and so on) of previously examined classes, which we manage by means of operations that add and retrieve pieces of information. For example, we define operations to add instances of the use relations, to add new datatype names, but also to retrieve the names of already defined datatypes, the functionality of constructors and operations and so on. Other functions perform more articulated computations over the environment: for example, NoCycles tests the absence of cycles inside a relation.

CEnv is the set of such environments; and it is ranged over by η, with possibly indexes and/or primes.

3.2.1.0.4 To retrieve information

- Use $(\mathsf{Inheritance}, \mathsf{Clientship})$: $\mathrm{CEnv} \to \mathit{Set}(\mathsf{Type}_{\mathsf{id}} \times \mathsf{Type}_{\mathsf{id}})$

  returns the usage (inheritance, clientship) relation

- $\mathsf{DataNames}$ $(\mathsf{ObjectNames}, \mathsf{ProcessNames})$: $\mathrm{CEnv} \to {\mathsf{Type}_{\mathsf{id}}}^*$

  returns the names of the declared datatypes (object, process classes)

- $\mathsf{AllNames}$: $\mathrm{CEnv} \to {\mathsf{Type}_{\mathsf{id}}}^*$

  returns all the declared names (of the datatypes, the object and the process classes)

- Field: $\mathrm{CEnv} \times \mathsf{Field}_{\mathsf{id}} \times \mathsf{Type}_{\mathsf{id}} \to \mathsf{Type}_{\mathsf{id}} \times \mathsf{Type}_{\mathsf{id}}$

  returns the declaration for a field with its type and the name of the class in which the declaration appears

- $\mathsf{ObjFields}$: $\mathrm{CEnv} \times \mathsf{Type}_{\mathsf{id}} \to (\mathsf{Field}_{\mathsf{id}} \times \mathsf{Type}_{\mathsf{id}})^*$

  returns the names and types of the fields of a given object class

- $\mathsf{Constructor}$: $\mathrm{CEnv} \times \mathsf{Con}_{\mathsf{id}} \times \mathsf{Type}_{\mathsf{id}} \to {\mathsf{Type}_{\mathsf{id}}}^* \times \mathsf{Type}_{\mathsf{id}} \times \mathit{Modifiers} \cup \{\mathsf{undef}\}$

  given a constructor name and a class, returns the declaration of the constructor with such name that is accessible from such class, the name of the class inside which it is declared and the access permission

  Modifiers= {pr, pu} is the set of the access restrictions that may be associated with constructors, operations and methods; it is ranged over by the *acc* metavariable.

  Notice that the presence of the inheritance relations in `JTN` requires to look for constructor (or operation or method) declarations following the superclass hierarchy.

- $\mathsf{Operation}$: $\mathrm{CEnv} \times \mathsf{Opn}_{\mathsf{id}} \times \mathsf{Type}_{\mathsf{id}} \to ({\mathsf{Type}_{\mathsf{id}}}^* \times \mathsf{Type}_{\mathsf{id}} \times \mathsf{Type}_{\mathsf{id}} \times \mathit{Modifiers}) \cup \{\mathsf{undef}\}$

  given an operation name and a class, returns the declaration for an operation with such name that is visible from such class, with the name of the class inside which it is declared and the access permission

- Method: $\mathrm{CEnv} \times \mathsf{Meth}_{\mathsf{id}} \times \mathsf{Type}_{\mathsf{id}} \to ({\mathsf{Type}_{\mathsf{id}}}^* \times \mathsf{Type}_{\mathsf{id}} \times \mathsf{Type}_{\mathsf{id}} \times \mathit{Modifiers}) \cup \{\mathsf{undef}\}$

  given a method name and a class, returns the declaration of the method with such name that is accessible from such class, the name of the class inside which it is declared and the access permission

- $\mathsf{CurrentType}$: $\mathrm{CEnv} \to \mathsf{Type}_{\mathsf{id}}$

  returns the name of the currently examined class (the last one added)

- $\mathsf{ObjectInt}$: $\mathrm{CEnv} \times \mathsf{Type}_{\mathsf{id}} \to \mathit{Set}(\mathsf{Comp}_{\mathsf{id}} \times \mathsf{Type}_{\mathsf{id}})$

  returns the set of object identifiers declared in the interface of a given process class

- $\mathsf{AsynchInt}(\mathsf{SynchInt})$: $\mathrm{CEnv} \times \mathsf{Type}_{\mathsf{id}} \to \mathit{Set}(\mathsf{Type}_{\mathsf{id}} \times \mathsf{IOSym} \times \mathsf{Chan}_{\mathsf{id}} \times \mathsf{Type}_{\mathsf{id}})$

  returns the set of the asynchronous (synchronous) channels declared in the interface of a given process class

- $\mathsf{UndefProcess}$: $\mathrm{CEnv} \to \mathsf{Type}_{\mathsf{id}}{}^*$

  returns the list of process class names used (in some names channel declarations) before being declared

- $\mathsf{CurrentOperation}$: $\mathrm{CEnv} \to \mathsf{Opn}_{\mathsf{id}} \times \mathsf{Type}_{\mathsf{id}} \times \mathsf{Type}_{\mathsf{id}}{}^* \times \mathsf{Modifier}$

  returns the name, the functionality and the access modifier of the current operation (the last operation added to the environment)

- $\mathsf{CurrentMethod}$: $\mathrm{CEnv} \to \mathsf{Meth}_{\mathsf{id}} \times \mathsf{Type}_{\mathsf{id}} \times \mathsf{Type}_{\mathsf{id}}{}^* \times \mathsf{Modifier}$

  returns the name, the functionality and the access modifier of the current method (the last method added to the environment)

3.2.1.0.5 To add or update information

The functions below simply add pieces of information to the environment. The meaning of each one of them is obvious.

- $\mathsf{AddUse}(\mathsf{AddInherit}, \mathsf{AddClient})$: $\mathrm{CEnv} \times \mathsf{Type}_{\mathsf{id}} \times \mathsf{Type}_{\mathsf{id}}{}^* \to \mathrm{CEnv}$

  adds a list of use (inheritance, clientship) relations

- $\mathsf{AddData}(\mathsf{AddObject}, \mathsf{AddProcess})$: $\mathrm{CEnv} \times \mathsf{Type}_{\mathsf{id}} \to \mathrm{CEnv}$

  adds a datatype (object, process) name, that becomes the current type

- AddCon: $\mathrm{CEnv} \times \mathsf{Modifier} \times \mathsf{Con}_{\mathsf{id}} \times \mathsf{Arg}^* \to \mathrm{CEnv}$

  adds a constructor declaration to the current class

- $\mathsf{AddOperation}$: $\mathrm{CEnv} \times \mathsf{Modifier} \times \mathsf{Opn}_{\mathsf{id}} \times \mathsf{Type}_{\mathsf{id}}{}^* \times \mathsf{Type}_{\mathsf{id}} \to \mathrm{CEnv}$

  analogous

- $\mathsf{AddMethod}$: $\mathrm{CEnv} \times \mathsf{Modifier} \times \mathsf{Meth}_{\mathsf{id}} \times \mathsf{Type}_{\mathsf{id}}{}^* \times \mathsf{Type}_{\mathsf{id}} \to \mathrm{CEnv}$

  adds a method declaration (that becomes the current method) to the current class

- $\mathsf{AddField}$: $\mathrm{CEnv} \times \mathsf{Field_{id}} \times \mathsf{Type_{id}} \to \mathrm{CEnv}$

  analogous

- $\mathsf{AddInterface}$: $\mathrm{CEnv} \times \mathsf{Comp_{id}} \times \mathsf{Type_{id}}^* \to \mathrm{CEnv}$

  analogous

- $\mathsf{AddAsynchInt}$: $\mathrm{CEnv} \times \mathsf{Type_{id}} \times \mathsf{Type_{id}} \times \{\,!\,,\ ?\,\}\mathsf{AsynCh} \to \mathrm{CEnv}$

  adds a list of asynchronous (synchronous) channel declarations to a process class interface

- $\mathsf{AddSynchInt}$: $\mathrm{CEnv} \times \mathsf{Type_{id}} \times \{\,!\,,\ ?\,\} \times \mathsf{SynCh} \to \mathrm{CEnv}$

  adds a list of asynchronous (synchronous) channel declarations to a process class interface

- $\mathsf{AddForw}$: $\mathrm{CEnv} \times \mathsf{Type_{id}} \to \mathrm{CEnv}$

  adds a process class name that has been used before being declared

- $\mathsf{SetCurrentType}$: $\mathrm{CEnv} \times \mathsf{Type_{id}} \to \mathrm{CEnv}$

  modifies the current type

3.2.1.0.6 Derived Predicates

- $\sqsubseteq$: $\mathrm{CEnv} \times \mathsf{Type_{id}} \times \mathsf{Type_{id}}$

  (Subtype relation definition) is a relation over the type names (but the `void` type is excluded), deduced from the inheritance relations in the (class) environment;

  $\sqsubseteq(\eta, T, T')$, written $T \sqsubseteq_\eta T'$ means that T is a subtype of T', in the environment η.

  - $T \sqsubseteq_\eta T$
  - if $T_1 \sqsubseteq_\eta T_2$ and $T_2 \sqsubseteq_\eta T_3$ then $T_1 \sqsubseteq_\eta T_3$
  - if $(T_1, T_2) \in \mathsf{Inheritance}(\eta)$ then $T_1 \sqsubseteq_\eta T_2$

- $\mathsf{DeclaredData(DeclaredObjects, DeclaredProcesses)}$: $\mathrm{CEnv} \times \mathsf{Type_{id}}^*$

  checks if a list of datatype (object, process) names have been declared in the environment

  $\mathsf{DeclaredData}(T; TL, \eta) \equiv T \in \mathsf{DataNames}(\eta) \wedge \mathsf{DeclaredData}(TL, \eta)$

  $\mathsf{DeclaredData}(\Lambda, \eta)$

- $\mathsf{NewName}$: CEnv $\times$ $\mathsf{Comp_{id}}$ $\times$ $\mathsf{Type_{id}}$

  checks if an object or channel name does not clash with another inherited one:

  $\mathsf{NewName}(\eta,\ id,\ T)$ iff $\forall\, T', \mathrm{T}''.\ T \sqsubseteq_\eta T' \Rightarrow$
  $(id,\mathrm{T}'') \notin \mathsf{ObjectInt}(T',\eta)$ and $(id,\mathrm{T}'') \notin \mathsf{AsynchInt}(T',\eta) \cup \mathsf{SynchInt}(T',\eta))$.

- $\mathsf{Visible}$: CEnv $\times$ $\mathsf{Type_{id}}$ $\times$ $\mathsf{Type_{id}}$

  checks whether a type is visible to another one, i.e., if the two types are in the closure of the `use` relations augmented with the inheritance relations (a type can use also the type it inherits from)

  - if $(T_1, T_2) \in \mathsf{Use}(\eta)$ then $\mathsf{Visible}(\eta, T_1, T_2)$
  - if $\mathsf{Visible}(\eta, T_1, T_2)$ and $\mathsf{Visible}(\eta, T_2, T_3)$ then $\mathsf{Visible}(\eta, T_1, T_3)$
  - if $\mathsf{Visible}(\eta, T_1, T)$ and $(T_1, T_2) \in \mathsf{Inheritance}(\eta)$ then $\mathsf{Visible}(\eta, T_2, T)$

3.2.2 Subtype relation

In the inference system we have two rules stating that if we can assign a type to a term, then we can assign it every other super type too.

$$\frac{\eta \ \vdash \ t : T}{\eta \ \vdash \ t : T'} \quad \textbf{if} \ \ T \sqsubseteq_\eta T'$$

$$\frac{\eta \ \vdash \ t : T, \eta'}{\eta \ \vdash \ t : T', \eta'} \quad \textbf{if} \ \ T \sqsubseteq_\eta T'$$

3.2.3 Class diagram

The main constraints for the correctness of a class diagram are:

- each class is correct;
- the class names are unique;
- the `use` relation relates only a datatype with another datatype, an object or a process class and has no cycles;
- the `inheritance` relation is single, relates classes of the same kind, has no cycles and does not introduce overloading of constructors, operations, methods or channel names;

- the `clientship` relation relates an object and a process class.

In the following we give the rules performing the checking of the constraints above.

Class diagram
A class diagram is correct if the data, the object and the process diagrams are correct.

$$\frac{\begin{array}{l}\eta_\phi \vdash \mathit{datatypes}\colon \mathit{void},\ \eta_1\\ \eta_1 \vdash \mathit{objdias}\colon \mathit{void},\ \eta_2\\ \eta_2 \vdash \mathit{pdias}\colon \mathit{void},\ \eta_3\end{array}}{\eta_\phi\ \vdash\ (\mathit{datatypes}, \mathit{objdias}, \mathit{pdias}) : \mathit{void}, \eta_3}$$

3.2.4 Datatype

Datatype Diagrams
A list of datatype diagrams is correct iff every datatype diagram is correct.

$$\frac{\eta_i\ \vdash\ \mathit{datatype}_i : \mathit{void}, \eta_{i+1}\quad i = 1, \ldots, n}{\eta_1\ \vdash\ \mathit{datatype}_1; \ldots; \mathit{datatype}_n : \mathit{void}, \eta_{n+1}}$$

Datatype Diagram
A datatype diagram is correct iff its name is new (it is added to the list of the datatype names and becomes the current type), and the constructors and the operations are correct. The operation check needs also the variable declarations.

$$\frac{\begin{array}{l}\mathsf{AddData}(T,\eta)\ \vdash \mathit{uses}\colon \mathit{void},\ \eta_1\\ \eta_1 \vdash \mathit{inherits}\colon \mathit{void},\ \eta_2\\ \eta_2,\mathrm{pr} \vdash \mathit{Pr\_cons}\colon \mathit{void},\ \eta_3\\ \eta_3,\mathrm{pu} \vdash \mathit{Pu\_cons}\colon \mathit{void},\ \eta_4\\ \eta_4,\mathrm{pr},\mathit{vardecs} \vdash \mathit{Pr\_opndefs}\colon \mathit{void},\ \eta_5\\ \eta_5,\mathrm{pu},\mathit{vardecs} \vdash \mathit{Pu\_opndefs}\colon \mathit{void},\ \eta_6\end{array}}{\eta\ \vdash\ (T, \mathit{uses}, \mathit{inherits}, \mathit{vardecs}, \mathit{Pr\_cons}, \mathit{Pu\_cons}, \mathit{Pr\_opndefs}, \mathit{Pu\_opndefs}) : \mathit{void},\ \eta_6}$$

if $T \notin \mathsf{DataNames}(\eta)$

Constructor Declarations
A list of constructor declarations is correct iff every declaration is correct and the names

are distinct.

$$\frac{\eta_i, acc \vdash con_i : void, \eta_{i+1} \quad i = 1, \ldots, n}{\eta_1, acc \vdash con_1; \ldots ; con_n : void, \eta_{n+1}}$$

if $\mathsf{DistinctNames}(\mathsf{CName}(con_1) \ ; \ \ldots \ ; \ \mathsf{CName}(con_n))$

where CName returns the name of the constructor in a constructor declaration:

$\mathsf{CName}(con(arg_1 \ ; \ \ldots \ ; \ arg_n)) = con.$

Constructor Declaration

A constructor declaration is correct iff it does not overload any inherited constructor name, every its argument is a datatype used by the current class and the names of its extractors are distinct. The resulting new environment contains the constructor declaration.

$$\frac{}{\eta, acc \vdash con((ext_1 : T_1); \ldots ; (ext_n : T_n)) : void, \mathsf{AddCon}(\eta, acc, con, T_1; \ldots ; T_n)}$$

if $T = \mathsf{CurrentType}(\eta)$
$\mathsf{Constructor}(\eta, \ con, \ T) = \mathsf{undef}$
$\mathsf{Visible}(\eta, T, T_i)$ for $i = 1, \ldots, n$
$\mathsf{DistinctNames}(ext_1; \ \ldots ; ext_n)$

Operation definitions

A list of operation definitions is correct iff every operation definition is correct.

$$\frac{\eta_i \vdash opndef_i : void, \eta_{i+1} \ i = 1 \ldots , n}{\eta_1 \vdash opndef_1; \ldots ; opndef_n : void, \eta_{n+1}}$$

Operation definition

An operation definitions correct iff it does not overload any inherited operation name, the types of its arguments are visible, its body is correct and its name is not used by another operation visible from the current class. The new environment contains the operation declaration.

$$\frac{\eta', vardecs \vdash opnbody : void}{\eta, vardecs, acc \vdash (opn, T_1; \ldots ; T_n, T_r, opnbody) : void, \eta'}$$

if $T = \mathsf{CurrentType}(\eta)$

$\mathsf{Operation}(\eta,\ opn,\ T) = \mathsf{undef}$
$\mathsf{Visible}(\eta, T, T_i)$ for $i = 1, \ldots, n$
$T_r \neq \texttt{void} \Rightarrow \mathsf{Visible}(\eta, T,\ T_r)$
$\eta' = \mathsf{AddOperation}(\eta,\ acc,\ opn,\ T_1; \ldots ; T_n,\ T_r)$

Operation body

An operation body is correct iff it consists of correct conditional rules.

$$\frac{\eta, vardecs \vdash condrule_i : void \quad i = 1, \ldots, n}{\eta, vardecs \vdash condrule_1; \ldots; condrule_n : void}$$

Conditional rule

A conditional rule is correct iff it uses the currently defined operation and every pattern has the type of the corresponding argument, and the conditional expressions have the operation result type.

$$\frac{\begin{array}{l}\eta, vardecs \vdash pt_i\!:\ T_i \quad i = 1, \ldots, n \\ \eta,\ vardecs,\ T \vdash condexp_j\!:\ T_r \quad j = 1, \ldots, k\end{array}}{\eta, vardecs \vdash opn(pt_1; \ldots; pt_n), condexp_1; \ldots; condexp_k : void}$$

if $\mathsf{CurrentOperation}(\eta) = opn, T_1; \ldots ; T_n, T_r, acc$
$\mathsf{Vars\_Of}(condexp_j) \subseteq \mathsf{Vars\_Of}(opn(pt_1; \ldots; pt_n)) \quad j = 1, \ldots, k$

Conditional expression

A conditional expression is correct with type T if the condition is a correct boolean expression without method calls and the expression has type T.

$$\frac{\begin{array}{l}\eta,\ vardecs \vdash bexp\!:\ BOOL \\ \eta,\ vardecs \vdash exp\!:\ T\end{array}}{\eta, vardecs \vdash (bexp, exp) : T}$$

if $\mathsf{NoMethodCall}(bexp, \eta)$, $\mathsf{NoMethodCall}(exp, \eta)$

where $\mathsf{NoMethodCall}(exp, \eta)$ checks if exp contains any call to a method declared in η.

Pattern1

A pattern built by a constructor declared within the current class T or is a public constructor of any other visible class $\hat{T}$, is correct iff the patterns have the same type of the *corresponding constructor arguments, and the variables are not repeated.*

$$\frac{\eta, vardecs \vdash pt_i : T_i \quad i = 1, \ldots, n}{\eta, vardecs \vdash con(pt_1; \ldots ; pt_n) : \hat{T}}$$

if $T=$ CurrentType(η)
Constructor(η, con, T) $=$ ($T_1; \ldots ; T_n, \hat{T}$,pu)

Constructor returns the declaration and the class of the constructor that is visible from the current class. If the constructor has been declared inside another class, it must be public or it has been inherited.

Pattern2
A pattern consisting of a variable is correct such variable has been declared in the current class.

$$\frac{}{\eta, (x, T); ll \vdash x : T}$$

$$\frac{\eta, ll \vdash x : T}{\eta, (x', T'); ll \vdash x : T} \quad \textbf{if } x' \neq x$$

3.2.4.1 Expression

Constructor Application
An expression built by a constructor is correct iff the expressions representing the arguments have the correct types.

$$\frac{\eta, vardecs \vdash exp_i : T_i \quad i = 1, \ldots, n}{\eta, vardecs \vdash con(exp_1; \ldots ; exp_n) : T'}$$

if CurrentType(η) $= T$
Constructor(η, con, T) $=$ ($T_1; \ldots ; T_n, T', acc$)

Operation Call
An expression built by an operation is correct iff the expressions representing the parameters have the types of the corresponding formal ones.

$$\frac{\eta, vardecs \ \vdash \ exp_i : T_i \quad i = 1 \ldots n}{\eta, vardecs \ \vdash \ opn(exp_1; \ldots; exp_n) : T_r}$$

if $\mathsf{CurrentType}(\eta) = T$
$\mathsf{Operation}(\eta, opn, T) = (T_1; \ldots; T_n, \ T_r, \ T', acc)$

Operation returns the declaration and the class of the operation that is visible from the current class.

Method call
An expression built by a method is correct iff the actual parameter expressions have the types of the corresponding method arguments.

$$\frac{\eta, vardecs \ \vdash \ exp_i : T_i \quad i = 1, \ldots, n}{\eta, vardecs \ \vdash \ met(exp_1; \ldots; exp_n) : T_r}$$

if $\mathsf{CurrentType}(\eta) = T$
$\mathsf{Method}(\eta, con, T) = (T_1; \ldots; T_n, \ T_r, \ T', acc), \ T_r \neq$ `void`

Method returns the declaration and the class of the method that is visible from the current class.

3.2.5 Relationships

Inheritance1
An inheritance relation is correct iff it involves an already defined class of the same type and does not introduce cycles.

$$\frac{}{\eta \ \vdash \ \texttt{inherit} \ T' : void, \mathsf{AddInherit}(\eta, (T, T'))}$$

if $\mathsf{CurrentType}(\eta) = T$
$\mathsf{NoCycles}(\mathsf{Inheritance}(\mathsf{AddInherit}(\eta, (T, T'))))$
$T, T' \in \mathsf{DataNames}(\eta)$ or $T, T' \in \mathsf{ObjectNames}(\eta)$ or $T, T' \in \mathsf{ProcessNames}(\eta)$

Inheritance2
The empty relation inheritance is obviously correct.

$$\frac{}{\eta \vdash \texttt{inherit} : void, \eta}$$

Use1

A use relations is correct iff it involves already declared datatypes and does not introduce cycles.

$$\frac{}{\eta \vdash \texttt{use}\ (T_1; \ldots; T_n) : void, \mathsf{AddUse}(\eta, (T, T_1); \ldots; (T, T_n))}$$

if $\mathsf{CurrentType}(\eta) = T$
$\mathsf{DeclaredData}(T_1; \ldots; T_n, \eta)$
$\mathsf{DistinctNames}(T_1; \ldots; T_n)$
$\mathsf{NoCycles}(\mathsf{Use}(\mathsf{AddUse}(\eta, (T, T_1); \ldots; (T, T_n))))$

DistinctNames is the operation that checks if the elements in a list are distinct.

Use2

The empty use relation is obviously correct.

$$\frac{}{\eta \vdash \texttt{use} : void, \eta}$$

Clientship1

A clientship relation when described in the object diagram is correct iff it involves distinct process class names.

$$\frac{}{\eta \vdash \texttt{client}\ T_1; \ldots; T_n : void, \mathsf{AddClient}((T, T_1); \ldots; (T, T_n), \eta)}$$

if $T = \mathsf{CurrentType}(\eta)$
$T \in \mathsf{ObjectNames}(\eta)$
$\mathsf{DistinctNames}(T_1; \ldots; T_n)$

Clientship2

A clientship relation when described in the process diagram is correct iff it involves distinct already declared object classes, and it has already been declared in the object classes.

$$\frac{}{\eta \ \vdash \ \texttt{client} \ T_1; \ldots; T_n : void, \eta}$$

if T= $\mathsf{CurrentType}(\eta)$
$T \in$ $\mathsf{ProcessNames}(\eta)$
$\mathsf{DistinctNames}(T_1; \ldots ; T_n)$
$\mathsf{DeclaredObjects}(T_1; \ldots ; T_n, \eta)$
$(T_i, T) \in \mathsf{Clientship}(\eta) \ i = 1 \ldots n$

Relations
A relation part is correct iff all its three components, describing clientship, inheritance and use relations, are correct.

$$\frac{\begin{array}{l}\eta \vdash clientships\text{: } void,\ \eta_1 \\ \eta_1 \vdash uses\text{: } void,\ \eta_2 \\ \eta_2 \vdash inherits\text{: } void,\ \eta_3\end{array}}{\eta \ \vdash \ (clientships, uses, inherits) : void, \eta_3}$$

3.2.6 Object class

Object Classes
A list of object class diagrams is correct iff every object class diagram is correct.

$$\frac{\eta_i \ \vdash \ objdia_i : void, \eta_{i+1} \quad i = 1, \ldots, n}{\eta_1 \ \vdash \ objdia_1; \ldots; objdia_n : void, \eta_{n+1}}$$

Object Class
An object class diagram is correct iff the class name is new (it is added to the list of the object class names and becomes the current type), the relations, the field declarations and the method definitions are correct.

$$\frac{\begin{array}{l}\mathsf{AddObject}(\eta, T) \ \vdash relations\text{: } void,\ \eta_1 \\ \eta_1 \vdash fields\text{: } void,\ \eta_2 \\ \eta_2,\ vardecs,\ \mathrm{pr} \vdash mets_1\text{: } void,\ \eta_3 \\ \eta_3,\ vardecs,\ \mathrm{pu} \vdash mets_2\text{: } void,\ \eta_4\end{array}}{\eta \ \vdash \ (T, relations, vardecs, fields, mets_1, mets_2) : void, \eta_4}$$

if $T \notin$ $\mathsf{AllNames}(\eta)$

Field Declarations

A list of field declarations is correct iff each field declaration is correct and the field names are distinct.

$$\frac{\eta_i \ \vdash \ (f_i, T_i, exp_i) : void, \eta_{i+1} \quad i = 1, \ldots, n}{\eta_1 \ \vdash \ (f_1, T_1, exp_1); \ldots; (f_n, T_n, exp_n) : void, \eta_{n+1}}$$

if DistinctNames$(f_1; \ \ldots \ ; f_n)$

Field Declaration

A field declaration is correct iff the field type is visible, the initialization expression is correct and does not contain neither method calls nor variables (it is evaluated in the context with an empty list of variable declarations). The new environment contains the field declaration.

$$\frac{\eta, \Lambda \ \vdash \ exp : T}{\eta \ \vdash \ (f, T, exp) : void, \mathsf{AddField}(\eta, f, T)}$$

if T'= CurrentType(η)
Visible(η, T', T)
NoMethodCall(exp,η)

Method Definitions

A list of method definitions is correct if each method definition is correct.

$$\frac{\eta_i, vardecs, acc \ \vdash \ metdef_i : void, \eta_{i+1} i = 1, \ldots, n}{\eta, vardecs, acc \ \vdash \ metdef_1; \ldots; metdef_n : void, \eta_{n+1}}$$

Method Definition

A method definition is correct iff it does not overload another one, the types of the arguments are visible, the returned type (if not `void`*) is visible too and the method body is correct; the environment is increased by such method definition.*

$$\frac{\eta', vardecs \ \vdash \ mbody : void}{\eta, vardecs, acc \ \vdash \ (met, T_1; \ldots; T_n, T_r, mbody) : void, \eta'}$$

if Method$(met$,CurrentType$(\eta),\eta)$ = undef
Visible$(\eta$,CurrentType$(\eta), T_i)$ $i = 1, \ldots, n$
$T_r \neq$ `void` $\Rightarrow$ Visible$(\eta$,CurrentType$(\eta), T_r)$

$\eta'=$ $\mathsf{AddMethod}(\eta,\ acc, met, T_1; \ldots; T_n,\ T_r)$

Method body

A method body is correct iff every method case is correct.

$$\frac{\eta, vardecs \ \vdash \ metcase_i : void \quad i = 1, \ldots, n}{\eta, vardecs \ \vdash \ metcase_1; \ldots; metcase_n : void}$$

Method case

A method case is correct iff the patterns have the types of the method arguments and the conditional statements have the method return type.

$$\frac{\begin{array}{l}\eta, vardecs \vdash pt_i\!: \ T_i \quad i = 1, \ldots, n \\ \eta, vardecs \vdash cstm_j\!: \ T_r \quad j = 1, \ldots, m\end{array}}{\eta, vardecs \ \vdash \ (met(pt_1; \ldots; pt_n), cstm_1; \ldots; cstm_m) : void}$$

if $\mathsf{CurrentMethod}(\eta) = (met, T_1; \ldots; T_n,\ T_r,\ acc)$

$\mathsf{Vars\_Of}(cstm_j) \subseteq \mathsf{Vars\_Of}(met(pt_1; \ldots; pt_n)) \quad j = 1, \ldots, k$

Conditional statement

A conditional statement is correct iff the conditional expression has type bool and the statements are correct; the list of statements has the return type of the method.

$$\frac{\begin{array}{l}\eta, vardecs \vdash bexp\!: \ BOOL \\ \eta, vardecs \vdash stms\!: \ T\end{array}}{\eta, vardecs \ \vdash \ (bexp, stms) : T}$$

Statements 1

A list of statements is correct if all the statements are correct, the last has type T and the others void.

$$\frac{\begin{array}{l}\eta, vardecs \vdash stm_i\!: \ void \ \ \text{i=1, \ldots ,n-1} \\ \eta, vardecs \vdash stm_n\!: \ T\end{array}}{\eta, vardecs \ \vdash \ stm_1; \ldots; stm_n : T}$$

Statements2

$$\frac{\eta, vardecs \ \vdash \ stm : T}{\eta, vardecs \ \vdash \ stm; \Lambda : T}$$

Method call

A method call is correct iff the parameters are correct expressions of the same type of the corresponding method argument, and the method is visible in the current class.

$$\frac{\eta, vardecs \ \vdash \ exp_i : T_i \quad i = 1, \ldots, n}{\eta, vardecs \ \vdash \ met(exp_1; \ldots; exp_n) : void}$$

if T= CurrentType(η)
Method(η,met,T) = (met, T_1; $\ldots$; T_n, `void`, T, acc)

Field assignment

A field assignment is correct iff the expression is correct and has the same type of the field.

$$\frac{\eta, vardecs \ \vdash \ exp : T_r}{\eta, vardecs \ \vdash \ f = exp : void}$$

if T= CurrentType(η)
Field(η,f,T)= (T_r, T')

Variable assignment

A variable assignment is correct if the expression is correct and has the type of the variable.

$$\frac{\begin{array}{l}\eta, vardecs \vdash x{:}\ T \\ \eta, vardecs \ \vdash exp{:}\ T\end{array}}{\eta, vardecs \ \vdash \ x = exp : void}$$

Result assignment

An assignment to the return variable is correct and has type T if the expression is correct and has type T.

$$\frac{\eta, vardecs \ \vdash \ exp : T}{\eta, vardecs \ \vdash \ \texttt{res} \ = exp : T}$$

3.2.7 Process class

We can separate the correctness of the interface from the correctness of the body of a process class. In the interface, the objects references belong to classes of objects that are in clientship relation with the process class. The channels (synchronous and asynchronous) must have different names. The type of the messages of each channel must be a datatype, in usage relation with the process class.

In the body, the constructors must have different names, the types of the arguments must be visible datatypes; in the graph, the same state constructor cannot appear in different nodes, the pattern expressions and the transitions are correct (this will be explained in more details later).

Process Classes
A list of process class diagrams is correct iff every process class diagram is correct and the forward declarations are correct, that is the process class names used inside other process classes before being declared, will be declared in a following process class diagram.

$$\frac{\eta_i \vdash pdia_i\!: \ void,\, \eta_{i+1} \quad i = 1, \ldots, n}{\eta_1 \ \vdash \ pdia_1; \ldots; pdia_n : void, \eta_{n+1}}$$

if $\mathsf{UndefProcess}(\eta_{n+1}) \subseteq \mathsf{ProcessNames}(\eta_{n+1})$

Process Class
A process class is correct iff the class name is new and the relations, the interface and the body are correct.

$$\frac{\begin{array}{l}\mathsf{AddProcess}(\eta, T) \ \vdash relations\!: \ void,\, \eta_1 \\ \eta_1 \vdash interface\!: \ void,\, \eta_2 \\ \eta_2 \vdash pbody\!: \ void\end{array}}{\eta \ \vdash \ (T, relations, interface, pbody) : void, \eta_2}$$

if $T \notin \mathsf{AllNames}(\eta)$

Interface
A process class interface is correct if the object and channel references are correct and their names are distinct.

$$\frac{\begin{array}{l}\eta \vdash \textit{obrefs}\colon \textit{void}, \eta_1 \\ \eta_1 \vdash \textit{asynchs}\colon \textit{void}, \eta_2 \\ \eta_2 \vdash \textit{synchs}\colon \textit{void}, \eta_3\end{array}}{\eta \ \vdash \ (\textit{obrefs}, \textit{asynchs}, \textit{synchs}) : \textit{void}, \eta_3}$$

if $\mathsf{DistinctNames}(\mathsf{Names}(\textit{obrefs}) \cup \mathsf{Names}(\textit{asynchs}) \cup \mathsf{Names}(\textit{synchs}))$

$\mathsf{Name}\colon \mathsf{ObjRef} \cup \mathsf{AsynCh} \cup \mathsf{SynCh} \rightarrow \mathsf{Comp}_{\mathsf{id}} \cup \mathsf{Chan}_{\mathsf{id}}$

returns the name contained in an object or channel reference declaration and Names is the obvious extension of Name to lists.

$\mathsf{Name}(\textit{oid}, T) = \textit{oid};$

$\mathsf{Name}(T,\ \textit{io},\ \textit{chan},\ T') = \textit{chan};$

Object references

A list of object references is correct iff each object reference is correct.

$$\frac{\eta_i \ \vdash \ \textit{obref}_i : \textit{void}, \eta_{i+1} \quad i = 1, \ldots, n}{\eta_1 \ \vdash \ \textit{obref}_1; \ldots; \textit{obref}_n : \textit{void}, \eta_{n+1}}$$

Object reference

An object reference is correct iff the name is new, the type is of kind object and in clientship relation with the process class (the relations have been already added to the environment during the check of the object classes). The object identity is added to the interface information about the current process in the environment.

$$\frac{}{\eta \ \vdash \ (\textit{oid},\ T) : \textit{void}, \mathsf{AddInterface}((\textit{oid},\ T), \eta)}$$

if $(T, \mathsf{CurrentType}(\eta)) \in \mathsf{Clientship}(\eta)$
$\mathsf{NewName}(\eta,\ \textit{oid},\ T)$

Asynchronous channel references

A list of asynchronous channel references is correct iff each reference is correct.

$$\frac{\eta_i \ \vdash \ \textit{ach\_ref}_i : \textit{void}, \eta_{i+1} i = 1, \ldots, n}{\eta_1 \ \vdash \ \textit{ach\_ref}_1 \ldots \textit{ach\_ref}_n : \textit{void}, \eta_{n+1}}$$

Asynchronous channel reference

An asynchronous channel declaration is correct iff the name is new, the type of the message is a visible one. A channel supporting named communications connects the process class with another process class. The reference is added to the environment.

$$\frac{}{\eta \ \vdash \ (T, io, ach\_id, T') : void, \eta''}$$

if T= $\mathsf{CurrentType}(\eta)$
$\mathsf{NewName}(\eta,\ ach\_id,\ T)$
$T' \in$ $\mathsf{DataNames}(\eta)$
$\mathsf{Visible}(\eta,\ T,\ T')$ and
η'= $\mathsf{AddAsynchInt}(\eta, T, T, io, ach\_id, T')$

$$\eta'' = \begin{cases} \mathsf{AddForw}(\eta',\ T) \text{ if } T \notin \mathsf{ProcessNames}(\eta) \\ \eta' \text{ otherwise} \end{cases}$$

Synchronous channel references

A list of synchronous channel reference is correct iff each reference is correct.

$$\frac{\eta_i \ \vdash \ sch\_ref_i : void, \eta_{i+1} \quad i = 1, \ldots, n}{\eta_1 \ \vdash \ sch\_ref_1; \ldots; sch\_ref_n : void, \eta_{n+1}}$$

Synchronous channel reference

A synchronous channel reference is correct iff the name is new and the type of the message is a visible one. The reference is added to the environment.

$$\frac{}{\eta \ \vdash \ (io, sch\_id, T') : void, \eta'}$$

if T= $\mathsf{CurrentType}(\eta)$
$T' \in$ $\mathsf{DataNames}(\eta)$
$\mathsf{Visible}(\eta,\ T,\ T')$
$\mathsf{NewName}(\eta,\ sch\_id,\ T')$
η'= $\mathsf{AddSynchInt}(\eta,\ T,\ io, sch\_id,\ T')$

Behaviour diagram

A behaviour diagram is correct iff the constructors are correct, the initial state is a correct expression of the sort of the process and the behaviour graph is correct.

$$\frac{\begin{array}{l}\eta,\mathrm{pr} \vdash cons\text{: } void,\ \eta_1\\ \eta_1, vardecs \vdash con(exp_1;\ \ldots;exp_n)\text{: } T\\ \eta_1, vardecs \vdash bgraph\text{: } void\end{array}}{\eta\ \vdash\ (cons, vardecs, con(exp_1;\ldots;exp_n), bgraph) : void}$$

if T= $\mathsf{CurrentType}(\eta)$

Behaviour graph

A behaviour graph is correct iff each node (called a state case) describes a different state and is correct. It does not change the environment.

$$\frac{\eta, vardecs\ \vdash\ state\_case_i : void\quad i = 1,\ldots,n}{\eta, vardecs\ \vdash\ state\_case_1;\ldots;state\_case_n : void}$$

if $\mathsf{DistinctNames}(\mathsf{Name}(state\_case_1);\ \ldots;\mathsf{Name}(state\_case_n))$

where $\mathsf{Name}((con(pts_1),transcase_1)\ ;\ \ldots\ ;\ (con(pts_n),transcase_n))\ =\ con.$

Statecase

A state case is correct iff the state constructor is the same for all the transition cases, it has been declared to be a constructor of the process class and the transition cases are correct.

$$\frac{\eta\ \vdash\ (con(pts_i), trans_i) : void\quad i = 1,\ldots,n}{\eta\ \vdash\ (con(pts_1), trans_1);\ldots;(con(pts_n), trans_n) : void}$$

if $\exists\ TL.\ \mathsf{Constructor}(\eta,\ con,\ \mathsf{CurrentType}(\eta))\text{= } (TL,\ \mathsf{CurrentType}(\eta),\ \mathrm{pr})$
$\mathsf{Vars\_Of}(trans_i)\subseteq\mathsf{Vars\_Of}(con(pts_1))\quad i = 1,\ldots,n$

Transition case

A transition case is correct iff the pattern and the transition alternatives are correct.

$$\frac{\begin{array}{l}\eta, vardecs \vdash con(pt_1;\ \ldots;pt_n)\text{: } T\\ \eta, vardecs \vdash tran\_alt_i\text{: } void\quad i = 1,\ldots,m\end{array}}{\eta, vardecs\ \vdash\ (con(pt_1;\ldots;pt_n), tran\_alt_1;\ldots;tran\_alt_m) : void}$$

if T= $\mathsf{CurrentType}(\eta)$

Transition alternative

A transition alternative is correct iff the guard is a correct boolean expression without method calls and the label-state alternatives are correct.

$$\frac{\eta,vardecs \vdash bexp\text{: } BOOL \qquad \eta,vardecs \vdash trans\text{: } void}{\eta, vardecs \vdash (bexp, trans) : void}$$

if $T=$ CurrentType(η)
NoMethodCall($bexp$,η)

3.2.7.1 Label-state alternatives

Channel communication
An alternative made by a communication on a channel is correct iff the label and the final state are correct.

$$\frac{\eta,vardecs \vdash comm\text{: } void \qquad \eta,vardecs \vdash st\text{: } void}{\eta, vardecs \vdash (comm, st) : void}$$

Method call
An alternative made by a method call is correct iff the label and the final state are correct.

$$\frac{\eta,vardecs \vdash oid.met(exps)\text{: } void \qquad \eta,vardecs \vdash st\text{: } void}{\eta, vardecs \vdash (oid.met(exps), st) : void}$$

Method call with returned value
Analogous to the previous case. Moreover, the type of the variable is the type of the method call.

$$\frac{\eta,vardecs \vdash x = oid.met(exps)\text{: } void \qquad \eta,vardecs \vdash st\text{: } void}{\eta, vardecs \vdash (x = oid.met(exps), st) : void}$$

Notice that since the above rules are the only one concerning a single alternative, we have that he case of a single label `else` is not correct.

Asynchronous input communications
An alternative made by several asynchronous readings is correct iff every label and every state is correct.

$$\frac{\begin{array}{l}\eta, vardecs \vdash st_i\text{: } void \quad i = 1, \ldots, n \\ \eta, vardecs \vdash\ ?\ ach\_id_i(x_i)\text{: } void \quad i = 1, \ldots, n\end{array}}{\eta,\, vardecs\ \vdash\ (\ ?\ ach\_id_1(x_1), st_1); \ldots; (\ ?\ ach\_id_n(x_n), st_n) : void}$$

if AsynchNames(sch_id_1; ... ; sch_id_n, T,η)

where AsynchNames checks if the identifiers have been declared in the current process interface to be asynchronous channels.

Analogous if the last transition is labelled by `else`.

Synchronous communications
An alternative made by several synchronous readings or writings is correct iff every label and every state is correct.

$$\frac{\begin{array}{l}\eta, vardecs \vdash st_i\text{: } void \quad i = 1, \ldots, n \\ \eta, vardecs \vdash io_i\ sch\_id_i(exp_i)\text{: } void \quad i = \\ 1, \ldots, n\end{array}}{\eta,\, vardecs\ \vdash\ ((io_1\ sch\_id_1(exp_1), st_1); \ldots; (io_n\ sch\_id_n(exp_n), st_n)) : void}$$

if SynchNames(sch_id_1; ... ; sch_id_n, T,η)

where SynchNames checks if the identifiers have been declared in the current process interface to be synchronous channels.

3.2.7.2 Labels

Asynchronous simple output communication
An asynchronous simple output communication is correct iff the expression denoting the sent value has the type conforming to the channel declaration contained in the interface of the process class.

$$\frac{\eta,\, vardecs\ \vdash\ exp : T'}{\eta,\, vardecs\ \vdash\ !\ chan(exp) : void}$$

if T= CurrentType(η)

$(void, !\ , chan, T') \in \mathsf{AsynchInt}(\eta,\ T)$

Asynchronous named output communication

Analogous to the previous case. Moreover, the expression denoting the receiver has the type conforming to the channel declaration contained in the interface of the process class

$$\frac{\begin{array}{l}\eta, vardecs \vdash exp_1\colon\ T' \\ \eta, vardecs \vdash exp_2\colon\ T_m\end{array}}{\eta,\ vardecs\ \vdash\ exp_1\ !\ chan(exp_2) : void}$$

if $T=$ $\mathsf{CurrentType}(\eta)$
$(T',\ !\ , chan, T_m) \in \mathsf{AsynchInt}(\eta,\ T)$

Asynchronous simple input communication

Analogous.

$$\frac{\eta,\ vardecs,\ T\ \vdash\ x : T_m}{\eta,\ vardecs\ \vdash\ ?\ chan(x) : void}$$

if $T=$ $\mathsf{CurrentType}(\eta)$
$(void,\ ?\ , chan, T_m) \in \mathsf{AsynchInt}(\eta,\ T)$

Asynchronous named input communication

Analogous.

$$\frac{\begin{array}{l}\eta, vardecs \vdash y\colon\ T' \\ \eta, vardecs \vdash x\colon\ T_m\end{array}}{\eta,\ vardecs\ \vdash\ y\ ?\ chan(x) : void}$$

if $T=$ $\mathsf{CurrentType}(\eta)$
$(T',\ ?\ , chan, T_m) \in \mathsf{AsynchInt}(\eta,\ T)$

Synchronous output communication

Analogous

$$\frac{\eta,\ vardecs\ \vdash\ exp : T_m}{\eta,\ vardecs\ \vdash\ !\ chan(exp) : void}$$

if T= $\mathsf{CurrentType}(\eta)$
$(\ !\ ,chan, T_m)\in \mathsf{SynchInt}(\eta,\ T)$

Synchronous input communication
Analogous

$$\frac{\eta, vardecs\ \vdash\ \ x : T_m}{\eta, vardecs\ \vdash\ \ ?\ chan(x) : void}$$

if T= $\mathsf{CurrentType}(\eta)$
$(\ ?\ ,chan, T_m)\in \mathsf{SynchInt}(\eta,\ T)$

Synchronous signal communication
A signal communication is correct if the channel has been declared to transmit messages of type void (i.e., signals).

$$\frac{}{\eta, vardecs\ \vdash\ \ !\ chan() : void}$$

if T= $\mathsf{CurrentType}(\eta)$
$(\ !\ ,chan, void)\in \mathsf{SynchInt}(\eta,\ T)$

Synchronous signal communication
Analogous

$$\frac{}{\eta, vardecs\ \vdash\ \ ?\ chan() : void}$$

if T= $\mathsf{CurrentType}(\eta)$
$(\ ?\ ,chan, \texttt{void})\ \in\ \mathsf{SynchInt}(\eta,\ T)$

3.2.7.3 Final States

Final state
A final state is correct iff it is a correct expression of the same type of the current process class.

$$\frac{\eta \vdash st : T}{\eta \vdash st : void}$$

if $T = \mathsf{CurrentType}(\eta)$

3.3 Architecture Diagram

The architecture diagram describes which are the instances of the object and of the process classes that compose the system state and how they cooperate. It is correct if all the components belong to declared object or process classes, if every object cooperation involves an instance of an object class and an instance of a process class and if the communication cooperation involves channels and processes that can be connected, according with the information contained in the process class interface diagrams.

The information about the context that is needed to check the architecture diagram correctness is class environment (thus, an element of the CEnv set) plus another structure, called the architectural environment, containing the components and the cooperation declarations. The set of the architectural environments is denoted by AEnv and it is ranged over by the α, α_i metavariables.

Notice that the class environment is used but never modified by the semantics of the architecture diagram. The rules below have the forms:

$$\eta;\alpha \vdash t{:}T,\ \alpha' \quad \text{or} \quad \eta;\alpha \vdash t{:}T$$

where T is a metavariable ranging over $\mathsf{Type_{id}} \cup \{void\}$.

3.3.1 Architecture environment

We characterize the architecture environment by the operations:

3.3.1.0.1 To retrieve information

- ObjID (PrID): $\mathrm{AEnv} \rightarrow \mathsf{Comp_{id}} \times \mathsf{Type_{id}}^*$

  returns the object (process) instance names that have been declared in the architecture diagram, with their types

- ConnID: AEnv $\rightarrow \mathsf{Conn_{id}}^*$

  returns the names of the asynchronous connectors that have been declared in the architecture diagram (the names are added by the AddAsynchCoops function defined below)

- $\mathsf{ObjCoops}$: AEnv $\rightarrow \mathsf{Comp_{id}} \times \mathsf{Comp_{id}} \times \mathsf{Comp_{id}} \times \mathsf{Meth_{id}}^*)^*$

  returns the list of the process cooperations that have been declared in the architecture diagram, with their types

- $\mathsf{AsynchCoops}$: AEnv $\rightarrow (\mathsf{Conn_{id}} \times \mathsf{Type_{id}} \times \mathsf{ConnKind} \times \mathsf{Comp_{id}}^* \times \mathsf{Comp_{id}}^*)^*$

  returns the list of the asynchronous cooperations.

- $\mathsf{SynchCoops}$: AEnv $\rightarrow (\mathsf{Conn_{id}} \times \mathsf{Type_{id}} \times \mathsf{Comp_{id}}^* \times \mathsf{Comp_{id}}^*)^*$

  returns the list of the synchronous cooperations.

3.3.1.0.2 To add or update information

- $\mathsf{AddObjID}$: AEnv $\times \mathsf{Comp_{id}} \times \mathsf{Type_{id}}^* \rightarrow$ AEnv

  adds a list of object instances

- $\mathsf{AddPrID}$: AEnv $\times \mathsf{Comp_{id}} \times \mathsf{Type_{id}}^* \rightarrow$ AEnv

  adds a list of process instances

- $\mathsf{AddObjCoops}$: AEnv $\times \mathsf{ObjectCoop}^* \rightarrow$ AEnv

  adds a list of object-process cooperation

- $\mathsf{AddAsynchCoops}$: AEnv $\times \mathsf{AsynhcCoop}^* \rightarrow$ AEnv

  adds a list of asynchronous cooperations

- $\mathsf{AddSynchCoops}$: AEnv $\times \mathsf{SynchCoop}^* \rightarrow$ AEnv

  adds a list of synchronous cooperations

Architecture Diagram

An architecture diagram is correct iff all its parts are correct.

$$\frac{\eta \vdash comps\text{: } void,\ \alpha \qquad \eta;\alpha \vdash coops\text{: } void,\ \alpha'}{\eta\ \vdash\ (comps, coops) : void, \alpha'}$$

3.3.2 Components

Components
A list of components is correct iff the object components are instances of declared object classes and the process components are instances of declared process classes and all the components have distinct names.

$$\frac{}{\eta \ \vdash \ \begin{array}{l}(\mathtt{ob}, oid_1, T_1) \ldots (\mathtt{ob}, oid_n, T_n) \\ (\mathtt{pr}, pid_1, T_1{}') \ldots (\mathtt{pr}, pid_k, T_k') \end{array} \quad : void, \alpha'}$$

if $\mathsf{DeclaredObjects}(T_1; \ldots; T_n, \eta)$
$\mathsf{DeclaredProcesses}(T_1{}'; \ldots; T_k', \eta)$
$\mathsf{DistinctNames}(oid_1; \ldots; oid_n; pid_1; \ldots; pid_k)$
$\alpha' = \mathsf{AddPrID}(\mathsf{AddObjID}(\alpha, (oid_1, T_1); \ldots; (oid_n, T_n)), (pid_1, T_1{}'); \ldots; (pid_k, T_k'))$

Component
Every object or process component has a type according to the declaration in the architectural environment.

$$\frac{}{\alpha \ \vdash \ pid : T} \quad \textbf{if } (pid, T) \in \mathsf{PrID}(\alpha)$$

$$\frac{}{\alpha \ \vdash \ oid : T} \quad \textbf{if } (oid, T) \in \mathsf{ObjID}(\alpha)$$

3.3.3 Cooperations

Cooperations
A cooperations part is correct iff all the cooperations are correct.

$$\frac{\begin{array}{l}\eta;\alpha \vdash objcoops{:}void,\ \alpha_1 \\ \eta;\alpha_1 \ \vdash Acoops{:}void,\ \alpha_2 \\ \eta;\alpha_2 \ \vdash scoops{:}void,\ \alpha_3\end{array}}{\eta;\alpha \ \vdash \ (objcoops, Acoops, scoops) : void, \alpha_3}$$

Object Cooperations

A list of object cooperations is correct iff every cooperation is correct and if the same local name, for the same process does not denote two different object instances.

$$\frac{\eta;\alpha_i \ \vdash \ (oid_i, pid_i, ln_i, mets_i) : void, \alpha_{i+1} \quad i = 1, \ldots, n}{\eta;\alpha_1 \ \vdash \ (oid_1, pid_1, ln_1, mets_1); \ldots; (oid_n, pid_n, ln_n, mets_n) : void, \alpha_{n+1}}$$

if $pid_i = pid_j$ and $ln_i = ln_j$ implies $oid_i = oid_j \quad i = 1, \ldots, n$

Object Cooperation

An object cooperation is correct iff the two components have been declared to be, respectively, an instance of an object class and an instance of a process class; the local name of the object appears in the interface of the process class; the methods names are visible from the object class. The cooperation is added to the environment.

$$\frac{}{\eta;\alpha \ \vdash \ (oid, pid, ln, met_1; \ldots; met_n) : void, \mathsf{AddObjCoops}(\alpha, (oid, pid, ln, met_1; \ldots; met_n))}$$

if $(oid,\ T) \in \mathsf{ObjID}(\alpha)$
$(pid,\ T') \in \mathsf{PrID}(\alpha)$
$(ln, T) \in \mathsf{ObjectInt}(\eta,\ T')$
$\mathsf{Method}(\eta, met_i, T) = (TL,\ T_r, T, \mathrm{pu}) \quad i = 1, \ldots, n$

Asynchronous Cooperations

A list of asynchronous cooperations is correct iff every cooperation is correct.

$$\frac{\eta;\alpha_i \ \vdash \ Acoop_i : void, \alpha_{i+1} \quad i = 1, \ldots, n}{\eta;\alpha_1 \ \vdash \ Acoop_1 \ldots Acoop_n : void, \alpha_{n+1}}$$

Asynchronous simple cooperation

An asynchronous simple cooperation is correct iff the process instances have been declared, every process class uses the message datatype, the channels have been declared with the local name and the message type in the process interface diagram. The connector declaration is added to the environment.

$$\frac{\begin{array}{l}\alpha \vdash pidI_i\text{:}\ \ TI_i \quad i = 1, \ldots n \\ \alpha \vdash pidO_j\text{:}\ \ TO_j \quad j = 1, \ldots, m\end{array}}{\eta;\alpha \ \vdash \ \begin{array}{l}(conn, T, \mathsf{s}, (pidI_1, lnI_1); \ldots; (pidI_n, lnI_n), \\ (pidO_1, lnO_1); \ldots; (pidO_m, lnO_m))\end{array} \quad : void, \alpha'}$$

if $(TI_i,\ T) \in \mathsf{Use}(\eta)$, $(void,\ ?\ , lnI_i,\ T) \in \mathsf{AsynchInt}(\eta, TI_i)$ $i = 1, \ldots, n$
$(TO_j, T) \in \mathsf{Use}(\eta)$ $(void,\ !\ , lnO_j,\ T) \in \mathsf{AsynchInt}(\eta, TO_j)$ $j = 1, \ldots, m$
$\alpha' = \mathsf{AddAsynchCoops}(\alpha, (conn, T, \mathsf{s}, (pidI_1, lnI_1); \ldots; (pidI_n, lnI_n), (pidO_1, lnO_1); \ldots; (pidO_m, lnO_m)))$

Analogous if the cooperation involves external entities.

Asynchronous broadcast cooperation
Analogous to the previous case

Asynchronous named cooperation
Analogous to the previous cases, but all the process performing input (output) operations are instances of the same class TI (TO) and the local names of the channel have been declared, in TI to be used for named communication with instances of class TO and vice versa.

$$\frac{\begin{array}{l}\alpha \vdash pidI_i\text{: } TI \quad i = 1, \ldots, n \\ \alpha \vdash pidO_j\text{: } TO \quad j = 1, \ldots, m\end{array}}{\eta;\alpha \ \vdash \begin{array}{l}(conn, T, \mathtt{n}, (pidI_1, lnI_1); \ldots; (pidI_n, lnI_n), \\ (pidO_1, lnO_1); \ldots; (pidO_m, lnO_m))\end{array} \quad : void, \alpha'}$$

if $(T,\ TI) \in \mathsf{Use}(\eta)$, $(T,\ TO) \in \mathsf{Use}(\eta)$,
$(TO,\ ?\ ,\ lnI_i,\ T) \in \mathsf{AsynchInt}(\eta, TI)$ $i = 1, \ldots, n$
$(TI,\ !\ ,\ lnO_j,\ T) \in \mathsf{AsynchInt}(\eta, TO)$ $j = 1, \ldots, m$
$\alpha' = \mathsf{AddAsynchCoops}(\alpha, (conn, T, \mathtt{n}, (pidI_1, lnI_1); \ldots; (pidI_n, lnI_n), (pidO_1, lnO_1); \ldots; (pidO_m, lnO_m)))$

3.3.3.1 Synchronous Cooperations

Synchronous Cooperations
A list of synchronous cooperations is correct iff every cooperation is correct.

$$\frac{\eta; \alpha_i \ \vdash \ scoop_i : void, \alpha_{i+1} \quad i = 1, \ldots, n}{\eta; \alpha_1 \ \vdash \ scoop_1 \ldots scoop_n : void, \alpha_{n+1}}$$

Synchronous Cooperation

A synchronous cooperation is correct iff: the process instances have been declared, every process class uses the message datatype, the channels have been declared synchronous channels with the local name and the message type in the process class interface diagrams. The environment is enriched by the cooperation declaration.

$$\frac{\begin{array}{l} pidI_i\colon\ TI_i \quad i=1,\ldots,n \\ pidO_j\colon\ TO_j \quad j=1,\ldots,m \end{array}}{\eta;\alpha\ \vdash \begin{array}{l}(sch\_id, T,((pidI_1,lnI_1);\ldots;(pidI_n,lnI_n)),\\ ((pidO_1,lnO_1));\ldots;(pidO_m,lnO_m))\end{array} \quad : void, \alpha'}$$

if $(\ ?\ ,lnI_i,\ T) \in \mathsf{SynchInt}(\eta, TI_i)\ i=1,\ldots,n$
$(\ !\ ,\ lnO_j,\ T) \in \mathsf{SynchInt}(\eta, TO_j)\ j=1,\ldots,m$
$(TI_i,\ T) \in \mathsf{Use}(\eta)\ \ i=1,\ldots,n$
$(TO_j,\ T) \in \mathsf{Use}(\eta)\ \ j=1,\ldots,m$
$\alpha' = \mathsf{AddSynchCoops}(\alpha,(sch\_id, T,(pidI_1,lnI_1);\ldots;(pidI_n,lnI_n),\ (pidO_1,lnO_1);\ldots;(pidO_m,lnO_m))$

Synchronous Signal Cooperation

Analogous.

3.4 Sequence Diagram

The static semantics of the sequence diagrams checks the correspondence of the described interactions with the cooperations declared in the architecture diagram. For such reason, the environment contains all the information found in the architecture diagram.

A sequence diagram is correct if:

- all the components (objects, processes and asynchronous connector) have been specified in the architecture diagram.
- the initialization expressions of the components are correct; the initial state of a process is described by an expression built using the constructors declared in the behaviour diagram; the initial state of an object is described by a sequence of field assignments; the initial state of an asynchronous channel is described by an expression without variables.

- the synchronous communications are labelled with the local name of the channel in the interface of the sender process; and the corresponding cooperation has been declared in the architecture diagram;
- the asynchronous communications, are labelled with the local name of the process that reads or writes from the connector, by the exchanged message and, if it is a named communication case, by the sender or the receiver identity; and the cooperation of the process by means of the connector has been declared in the architecture diagram.
- the interactions of an object and a process by a method call have been described by an object cooperation of the architecture diagram;
- the annotations are correct boolean expressions, according with the information collected in the class diagram.

Sequence Diagram
A sequence diagram is correct iff the initialization expression, the cooperations and the annotations (boolean expressions) are correct.

$$\frac{\begin{array}{l}\eta;\alpha,\ vardecs \vdash inits\colon\ void\\ \eta;\alpha,\ vardecs \vdash interacts\colon\ void\\ \eta;\alpha,\ vardecs \vdash anns\colon\ BOOL\end{array}}{\eta;\alpha\ \vdash\ (vardecs, inits, interacts, anns) : void}$$

Initial process state
An initialization expression for a process is correct iff the instance has been declared and the expression has the same type of the process.

$$\frac{\mathsf{SetCurrentType}(\eta, T), vardecs\ \vdash\ con(exps) : T}{\eta;\alpha, vardecs\ \vdash\ (act, con(exps)) : void}$$

if $\exists\ T.(act, T) \in \mathsf{PrID}(\alpha)$

Initial object state
An initialization expression for an object is correct iff the instance has been declared in the architecture diagram and the expressions are assignments to the object fields.

$$\frac{\mathsf{SetCurrentType}(\eta, T), vardecs\ \vdash\ exp_i : TI\ \ i = 1, \ldots, n}{\eta;\alpha, vardecs\ \vdash\ (act, f_1 = exp_1; \ldots; f_n = exp_n) : void}$$

if $\exists\ (act, T) \in \mathsf{ObjID}(\alpha),\ (f_1, T_1; \ldots ; f_n, T_n) \in \mathsf{ObjFields}(\eta, T)$

3.4.1 Interactions

Synchronous interaction with message
A synchronous interaction is correct iff there is a synchronous cooperation where the first actor participates as a sender, with the local name for the channel used in the interaction, and the second actor participates as a receiver.

$$\frac{\begin{array}{l}\alpha \vdash act_1\colon\ T \\ \mathsf{SetCurrentType}(\eta,\ T),\ vardecs \vdash exp\colon\ T'\end{array}}{\eta;\alpha, vardecs\ \vdash\ (act_1,\ !\ ln(exp), act_2) : void}$$

if $\mathsf{OkSynchCoop}(T',\ act_1,\ ln,\ act_2,\ \alpha)$

where

$\mathsf{OkSynchCoop}(T',\ act_1,\ ln_1,\ act_2,\ \alpha) \equiv \exists conn,\ T', ln_2, inl, outl.(conn,\ T', inl, outl) \in \mathsf{SynchCoops}(\alpha)$ and $(act_1, ln_1) \in inl$ and $(act_2, ln_2) \in outl$

The definition is obvious if one of the two components is the external world.

Synchronous interaction with signal
A synchronous interaction is correct if there is a synchronous cooperation where the first actor participates as a sender, with the local name for the channel used in the interaction, and the second actor participates as a receiver.

$$\frac{}{\eta;\alpha, vardecs\ \vdash\ (act_1,\ !\ ln(), act_2) : void}$$

if $\mathsf{OkSynchCoop}(\texttt{void},\ act_1,\ ln,\ act_2,\ \alpha)$

The $\mathsf{OkAsynchCoop}$ operation used below is defined in a similar way:

$\mathsf{OkAsynchCoop}(T_m,\ act_1,\ ln_1,\ conn,\ \alpha) \equiv \exists inl, outl.(conn,\ T_m, inl, outl) \in \mathsf{AsynchCoops}(\alpha)$ and $(act_1, ln_1) \in outl$

Asynchronous output communication
An asynchronous output interaction is correct if there is a asynchronous cooperation with

the connector conn, where the act component participates (as a sender) with the channel named ln.

$$\frac{\alpha \ \vdash \ act_1 : T \quad \eta, vardecs, T \ \vdash \ exp : T_m}{\eta;\alpha, vardecs \ \vdash \ (act_1, !ln(exp), conn) : void}$$

if $\mathsf{OkAsynchCoop}(T_m,\ act_1,\ ln,\ conn,\ \alpha)$

Asynchronous input communication

An asynchronous input interaction is correct if there is a asynchronous cooperation with the connector conn, where the act component participates (as a receiver) with the channel named ln.

$$\frac{\alpha \ \vdash \ act_1 : T \quad \eta, vardecs, T \ \vdash \ exp : T_m}{\eta;\alpha, vardecs \ \vdash \ (act_1, ?ln(exp), conn) : void}$$

if $\mathsf{OkAsynchCoop}(T_m,\ act_1,\ ln,\ conn,\ \alpha)$

Method Call 1

A method call to an object by a process is correct if there exists the corresponding object cooperation in the environment and the method call expression is correct.

$$\frac{\begin{array}{l}\alpha \vdash act_2\!:\ T_2 \\ \eta, vardecs, T_2 \vdash met(exp)\!:\ void\end{array}}{\eta;\alpha, vardecs \ \vdash \ (act_1, met(exp), act_2) : void}$$

if $(act_2, act_1, ln, met) \in \mathsf{ObjCoops}(\alpha)$

Method Call 2

An assignment with a method call is correct if there exists the corresponding object cooperation in the environment and the method call expression is correct.

$$\frac{\begin{array}{l}\alpha \vdash act_2\!:\ T_2 \\ \eta, vardecs, T_2 \vdash x\!:\ T \\ \eta, vardecs, T_2 \vdash met(exp)\!:\ T\end{array}}{\eta;\alpha, vardecs \ \vdash \ (act_1, x = met(exp), act_2) : void}$$

if $(act_2, act_1, ln, met) \in \mathsf{ObjCoops}(\alpha)$

3.4.2 Annotations

An annotation is an equation of the form x= *exp* where *exp* denotes a datatype value or an object instance.

The type of x and *exp* must be the same, when calculated in the class that uses the x variables. This is the reason why the variable declarations in a sequence diagram is equipped with the name of the class of the instance that uses it.

Annotations
A list of annotations arecorrect iff they are boolean equations whose left side is one of the declared variables, while the right side is an expression of the same type denoting a datatype value or an object.

$$\frac{\mathsf{SetCurrentType}(\eta,\, T),\, vardecs\ \vdash\ exp : T}{\eta;\alpha,\, vardecs\ \vdash\ x = exp}\quad \mathsf{User}(x) = act \text{and } \exists\ T.(act, T) \in \mathsf{PrID}(\alpha)$$

where $\mathsf{User}(x,(y,\ act);\ ll) = \mathsf{User}(x,\ ll)$ if $x \neq y$
and $\mathsf{User}(x,\ (x,\ act);\ ll) = act$
returns the actor using the x variable in one of its interactions.

3.5 `JTN` specification

`JTN` specification
A `JTN` specification is correct if the class diagram, the architecture diagram and the sequence diagrams are correct.

$$\frac{\begin{array}{l}\eta_\phi \vdash cldia\text{: } void,\ \eta_1\\ \eta_1 \vdash archdia\text{: } void,\ \alpha\\ \eta_1;\alpha \vdash seqdias\text{: } void\end{array}}{\eta_\phi\ \vdash\ (cldia, archdia, seqdias) : void}$$

η_ϕ is the empty class environment.

Chapter 4

Formal Semantics of JTN

JTN is a visual specification language with a completely formal counterpart. Indeed, we can give the semantics of a JTN specification, by means of an algebraic specification presented using the CASL (see [oLD99]) language. Such algebraic specification, which will be monomorphic[1], describes a generalized transition system (shortly *glts*: see Chap.D) that models the system specified by the JTN diagrams.

The states of that system are n-tuples of states of processes, objects, synchronous and asynchronous channels. The labels describe the I/O operations over files and sockets that are observable from outside of the system. The transitions are defined as the composition of the transitions of the internal active components (the processes).

The processes, on their turn, are described by simple labelled transition systems (shortly *ltss*, see Chap.D), that are *gltss* lacking the information associated with the transitions, and that do not have subcomponents performing activities.

We assume that the JTN specifications to whom we give semantics are statically correct; this is an acceptable assumption, as we precisely describe what is the static semantics of a JTN specification (see 3). This assumption allows us to simplify the description of this formal semantics.

The formal semantics of JTN is precisely defined over its textual presentation (see Appendix A) as it results after a pre-processing phase that performs some simplifications and transformations over the specification. The preprocessing:

- orders the datatypes following the `use` relations;
- eliminates the `inheritance` relations over process classes: if process class B inherits from process class A, the interface diagram for B is enriched by the channel declar-

[1] i.e., such that the class of its models is an isomorphism class.

ations and by the state constructors of A, whereas the behaviour diagram of B is substituted by the behaviour diagram of A enriched with the new transitions of B.

We give the formal semantics to `JTN` by means of an inference system to prove assertions of the form: $JTN \vdash_{\texttt{JTN}} casl$, with the meaning that the *JTN* construct has the formal semantics described by the CASL expression *casl* (that can be a specification or just a part of a specification). The inference system is given following the structure of the specifications, so we have different derivation symbols to give the semantics of different fragments of `JTN`. Again, as for the static semantics, some context information (named *environment*, and generically denoted by ρ) are necessary to give the semantics to particular parts of the specification: thus the derivations usually have the form

$frag, \rho \vdash_{\text{Term}} \texttt{term}, \rho'$,

where *frag* is a generic `JTN` fragment, `term` is its formal semantics, thus a (part of) CASL specification, ρ and $\rho'$are omitted when contain no information (ρ' is the information structure updated with new information get from *term*). $\rho$is omitted when it is not used;$\rho'$is omitted when is equal to ρ.

In the following we denote a part of a `JTN` specification using the *italic* font, while the corresponding semantics is displayed in `true type`. The keywords that indicate structuring operators of CASL are displayed **boldface**.

4.1 Environment

The information on the context needed to give the semantics to a `JTN` construct is called *environment* and it is a list of sets containing: sort names, field names, channel names, component identities and method signatures.

The information content is managed by means of operations that add and retrieve pieces of information. For example, we define operations to add sort names as well as operations to retrieve the names of the already defined sorts.

SEnv is the set of such environments, ranged over by the ρ metavariable (also with indexes and primes).

To add information

- $\mathsf{AddType}$: $\mathsf{Type}_{\mathsf{id}} \times \mathrm{SEnv} \to \mathrm{SEnv}$

  adds the name of a type corresponding to a datatype or to an object class. It becomes the current one.

- $\mathsf{AddField}\colon \mathsf{Field}_{\mathsf{id}} \times \mathrm{SEnv} \to \mathrm{SEnv}$

  adds the name of a field to the current object class

- $\mathsf{AddSchanNames}\colon {\mathsf{Chan}_{\mathsf{id}}}^* \times \mathrm{SEnv} \to \mathrm{SEnv}$

  adds the names of the synchronous channels to the current process class

- $\mathsf{AddAchanNames}\colon {\mathsf{Chan}_{\mathsf{id}}}^* \times \mathrm{SEnv} \to \mathrm{SEnv}$

  adds the names of the asynchronous channels to the current process class

- $\mathsf{SetReturnType}\colon \mathsf{Meth}_{\mathsf{id}} \times \mathsf{Type}_{\mathsf{id}} \times \mathrm{SEnv} \to \mathrm{SEnv}$

  sets the return type of a given method in the current class

- $\mathsf{SetPattAx}\colon \mathtt{String} \times \mathrm{SEnv} \to \mathrm{SEnv}$

  stores in the environment the axiom describing that the previous patterns are not matched

- $\mathsf{AddComponents}\colon \{\mathtt{pr}, \mathtt{ob}\} \times \mathsf{Comp}_{\mathsf{id}} \times \mathsf{Type}_{\mathsf{id}} \times \mathtt{Ident} \to \mathrm{SEnv}$

  stores in the environment a component information, that is 4-tuple containing the kind (object or process), the name, the type and the identity.

To retrieve information

- $\mathsf{IdentComp}\colon \mathsf{Comp}_{\mathsf{id}} \times \mathrm{SEnv} \to \mathtt{Ident}$

  returns the identifier (a term of the CASL sort `Ident`) encoding the identity of a component

- $\mathsf{CurrentType}\colon \mathrm{SEnv} \to \mathsf{Type}_{\mathsf{id}}$

  returns the name of the current type

- $\mathsf{ReturnType}\colon \mathrm{SEnv} \to \mathsf{Type}_{\mathsf{id}}$

  returns the return type of the current method

- $\mathsf{FieldNames}\colon \mathsf{Type}_{\mathsf{id}} \times \mathrm{SEnv} \to {\mathsf{Field}_{\mathsf{id}}}^*$

  returns the names of the fields of an object class

- $\mathsf{SchanNames}$ $(\mathsf{AchanNames}, \mathsf{BchanNames}, \mathsf{NchanNames})\colon \mathsf{Type}_{\mathsf{id}} \times \mathrm{SEnv} \to {\mathsf{Chan}_{\mathsf{id}}}^*$

  returns the names of the synchronous (simple asynchronous, broadcast, named asynchronous) channels of a process class

- Sorts: $\mathrm{SEnv} \to \mathsf{Type_{id}}^*$

  returns the list of the already specified types

- $\mathsf{GetPattAx}$: $\mathrm{SEnv} \to \texttt{String}$

  returns the axiom describing that the previous patterns are not matched

4.2 JTN Specification

JTN specification
The semantics of the class diagram is a sequence of CASL *specifications: one for each class of the diagram; the semantics of the architecture diagram is the specification of the global system (the glts); the semantics of the sequence diagrams is a specification that modifies the system specification by adding some temporal properties to constrain the global behaviour of the system.*

$$\frac{\begin{array}{l} cldia \vdash_{\overline{\text{Class}}} \texttt{SPs} \\ archdia \vdash_{\overline{\text{Arch}}} \texttt{SP} \\ seqdias \vdash_{\overline{\text{Seqs}}} \texttt{SP}' \end{array}}{(cldia, archdia, seqdias) \vdash_{\overline{\text{JTN}}} \texttt{SPs SP SP}'}$$

4.3 Class Diagram

Class diagram
The semantics of a class diagram is the sequence of the CASL *specifications that are the semantics of its datatype, object and process diagrams.*

$$\frac{\begin{array}{l} datatypes \vdash_{\overline{\text{Datas}}} \texttt{SPs}, \rho_1 \\ objdias, \rho_1 \vdash_{\overline{\text{Objs}}} \texttt{SPs}', \rho_2 \\ pdias, \rho_2 \vdash_{\overline{\text{Procs}}} \texttt{SPs}'' \end{array}}{(datatypes, objdias, pdias) \vdash_{\overline{\text{Class}}} \texttt{SPs SPs}' \texttt{ SPs}''}$$

4.3.1 Datatypes

Datatypes

The semantics of some JTN *datatype diagrams is a sequence of* CASL *specifications, one for each diagram.*

$$\frac{datatype_i, \rho_i \vdash_{\text{Data}} \mathtt{SP}_i, \rho_{i+1} \quad i = 1, \ldots, n}{datatype_1; \ldots; datatype_n \vdash_{\text{Datas}} \mathtt{SP}_1 \ldots \mathtt{SP}_n, \rho_{n+1}}$$

if ρ_1 is the empty environment, that is the list of empty sets.

Datatype

The semantics of a datatype is a JTN *specification that extends the specifications corresponding to the used datatypes, renames the sort of the inherited datatype (if any), defines a sort, a set of constructors and a set of operations and hides all the private constructors and operations.*

$$\frac{\begin{array}{l} Pr\_cons \vdash_{\text{Cons}} \mathtt{cons} \\ Pu\_cons \vdash_{\text{Cons}} \mathtt{cons\acute{}} \\ vardecs \vdash_{\text{Vars}} \mathtt{vardecs} \\ Pr\_opndefs, \rho' \vdash_{\text{Opns}} \mathtt{opns} \\ Pu\_opndefs, \rho' \vdash_{\text{Opns}} \mathtt{opns\acute{}} \end{array}}{\begin{array}{l} (T, \mathtt{use}\ T_1; \ldots; T_n, \mathtt{inherit}\, T', vardecs, \\ Pu\_cons, Pr\_cons, Pu\_opndefs, Pr\_opndefs), \rho \end{array} \vdash_{\text{Data}} \begin{array}{l} \textbf{spec}\ T\_\mathtt{SP}\ = \\ \textbf{free}\{ \\ T_1\_\mathtt{SP}\ \ \textbf{and}\ \ldots\ \textbf{and}\ T_n\_\mathtt{SP}\ \ \textbf{and} \\ T'\_\mathtt{SP} \\ \textbf{with}\ T' \mapsto T \\ \textbf{hide}\ cons\ opns \\ \textbf{then} \\ \textbf{type}\ T ::= \mathtt{cons}\ |\mathtt{cons\acute{}} \\ \mathtt{vardecs} \\ \mathtt{opns} \\ \mathtt{opns\acute{}} \\ \} \end{array}, \rho'}$$

if $cons = \mathsf{ConNames}(Pr\_cons)$
$opns = \mathsf{OpnNames}(Pr\_opndefs)$
$\rho' = \mathsf{AddType}(T, \rho)$

$\mathsf{ConNames}$: $\mathsf{ConDec}^* \to \mathsf{Con_{id}}^*$ returns the names of the constructors in a constructor definition list.

$\mathsf{OpnNames}$: $\mathsf{OpnDef}^* \to \mathsf{Opn_{id}}^*$ returns the names of the operations in an operation definition list.

Constructors

$$\frac{con_i \vdash_{\overline{\text{Con}}} \texttt{con}_i \quad i = 1, \ldots, n}{con_1; \ldots; con_n \ \vdash_{\overline{\text{Cons}}} \texttt{con}_1 | \ldots | \texttt{con}_n}$$

Constructor

The semantics of a JTN *constructor (possibly with extractors) is the declaration of a* CASL *constructor.*

$$\frac{}{(con(ext_1, T_1); \ldots; (ext_n, T_n)) \vdash_{\overline{\text{Con}}} con(ext_1 \colon T_1; \ldots; ext_n \colon T_n)}$$

Variable declarations

The semantics of a JTN *variable declaration is a* CASL *variable declaration.*

$$\frac{}{(x_1, T_1); \ldots; (x_n, T_n) \vdash_{\overline{\text{Vars}}} \textbf{vars } x_1 \colon T_1; \ldots; x_n \colon T_n;}$$

Operation Definitions

The semantics of a list of operation definitions is the list of the semantics of each single operation.

$$\frac{opndef_i, \rho \vdash_{\overline{\text{Opn}}} \texttt{opndef}_i \quad i = 1, \ldots, n}{opndef_1; \ldots; opndef_n, \rho \vdash_{\overline{\text{Opns}}} \texttt{opndef}_1; \ldots; \texttt{opndef}_n;}$$

Operation Definition

The semantics of an operation definition is a CASL *operation declaration plus the axioms corresponding to its body.*

$$\frac{opnbody, \rho \vdash_{\overline{\text{OpnBody}}} \texttt{AXs}}{(opn, T_1; \ldots; T_n, T_r, opnbody), \rho \vdash_{\overline{\text{Opn}}} \textbf{op } opn \colon T_1 \times \cdots \times T_n \to T_r; \texttt{AXs}}$$

Operation Body

A conditional rule corresponds to a CASL *axiom. To render the explicit ordering of the conditional rules in the* JTN *diagram, the premises of every axiom (associated with a condition cond) require that none of the previous pattern and conditions can be satisfied (while cond is) by the elements to which the operation is applied. The fragment of axiom stating that the previous patterns are not matched is stored in the environment and it used by the semantics of the the single conditional rule.*

$$\frac{condrule_i, \rho_i \vdash_{\text{CondRule}} \mathtt{AXs}_i, \rho_{i+1} \quad i = 1, \ldots, k}{condrule_1; \ldots; condrule_k, \rho \vdash_{\text{OpnBody}} \begin{array}{l} \textbf{preds} \\ \mathtt{NoMatch}_{[pt_1]}: \ldots; \\ \ldots \\ \mathtt{NoMatch}_{[pt_n]}: \ldots; \\ \textbf{axioms} \\ NoMatchDef_1 \ldots NoMatchDef_n \\ \mathtt{AXs}_1 \ldots \mathtt{AXs}_k \end{array}}$$

if $\rho_1 = \mathsf{SetPattAx}(\mathbf{true}, \rho)$

Given a pattern pt, $\mathtt{NoMatch}_{[pt]}$ is a predicate with an argument of the sort of pt, checking whether an element matches pt. It is defined by the following axioms
$\mathtt{NoMatch}_{[\mathtt{x}]}(pt) \equiv \mathbf{false}$,
$\mathtt{NoMatch}_{[pt]}(pt) \equiv \mathbf{false}$,
$\mathtt{NoMatch}_{[con(pt_1;\ldots;pt_n)]}(con'(pt'_1; \ldots ; pt'_m)) \equiv \mathbf{true}$ if $con \neq con'$
$\mathtt{NoMatch}_{[con(pt_1;\ldots;pt_n)]}(con(pt'_1; \ldots ; pt'_n)) \equiv \mathtt{NoMatch}_{[pt_1]}(pt'_1) \lor \ldots \lor \mathtt{NoMatch}_{[pt_n]}(pt'_n)$

The $\mathtt{NoMatch}_{[\ldots]}$ predicates that are needed may be determined by looking at the various conditional rules appearing in the operation body.

Let us recall, that it is not possible to use repeated variables in the same pattern expression, that is, there cannot be two or more occurrences of the same variable in the same pattern expression.

Conditional Rule

$$\frac{\begin{array}{l} exp_i \vdash_{\text{Exp}} \mathtt{term}_i \\ bexp_i \vdash_{\text{Exp}} \mathtt{bterm}_i \ i = 1, \ldots, m \end{array}}{con(pt_1; \ldots; pt_n), (bexp_1, exp_1); \ldots; (bexp_m, exp_m), \rho \vdash_{\text{CondRule}} \mathtt{AXs}_1 \ldots \mathtt{AXs}_m, \rho'}$$

if for $i = 1, \ldots, m$
$\mathtt{Ax} = \mathsf{GetPattAx}(\rho)$
$\mathtt{AXs}_i \equiv \mathtt{Ax} \land \mathtt{bterm}_1 = \mathbf{false} \land \ldots \land \mathtt{bterm}_{i-1} = \mathbf{false} \land$
$\mathtt{bterm}_i = \mathbf{true} \land \mathtt{x}_1 = pt_1 \land \ldots \land \mathtt{x}_n = pt_n \Rightarrow opn(\mathtt{x}_1, \ldots, \mathtt{x}_n) = \mathtt{term}_i$
$\rho' = \mathsf{SetPattAx}(\rho, \mathtt{Ax} \land \mathtt{NoMatch}_{[pt_1]}(\mathtt{x}_1) \land \ldots \land \mathtt{NoMatch}_{[pt_n]}(\mathtt{x}_n))$

4.3.2 Expressions

Variable

A JTN *variable corresponds to the same* CASL *variable.*

$$\frac{}{x \vdash_{\text{Exp}} x}$$

Constructor

$$\frac{exp_i \vdash_{\text{Exp}} \texttt{term}_i \quad i = 1, \ldots, n}{con(exp_1; \ldots; exp_n) \vdash_{\text{Exp}} con(\texttt{term}_1, \ldots, \texttt{term}_n)}$$

Operation

$$\frac{exp_i \vdash_{\text{Exp}} \texttt{term}_i \quad i = 1, \ldots, n}{opn(exp_1; \ldots; exp_n) \vdash_{\text{Exp}} opn(\texttt{term}_1, \ldots, \texttt{term}_n)}$$

There is no rule for the semantics of method calls, because they are not permitted inside an operation body.

4.3.3 Object Class

The semantics of an object class diagram is a CASL algebraic specification with a sort named as the object class. The constructors for the elements of the sort are the methods defined in the object class plus the special 0-ary operation `initial` that is the constructor of the initial (unique) state. The specification has operations that, given an element of the class, return the values of the fields (named as the fields themselves). The value of the fields for the element denoted by `initial` are those reported in the object body diagram.

The `use` and the `inheritance` relations (if any) are rendered by the **then** and **with** constructs of CASL that allow to extend some specifications with new sorts and operations, renaming some sorts. If class B inherits from class A, the sort A is renamed to B in the specification of class B. Thus, every operation corresponding to a field or to a method in the specification of class A becomes an operation in the specification of class B. Let us recall that, for the moment, we do not admit neither method overloading nor overriding or field hiding.

A method of class T is an operation of the specification that takes the same arguments of the method, plus an additional one of sort T and returns just a value of sort T (if the method has a void return type), or a value of sort T plus a value of the returned sort.

Object Class Diagrams

The semantics of a list of object class diagrams is the specification of the identities of the (object) instances (`Ident_SP`*) of any type plus a sequence of* CASL *specifications, one for each object class, each one using* `Ident_SP`*.*

$$\frac{objdia_i, \rho_i \vdash_{\mathrm{Obj}} \mathtt{SP}_i, \rho_{i+1} \quad i = 1, \ldots, n}{objdia_1; \ldots; objdia_n, \rho_1 \vdash_{\mathrm{Objs}} \begin{array}{l} \textbf{spec } \mathtt{Ident\_SP} = \\ \mathtt{Integer\_SP} \textbf{ with } \mathtt{int} \mapsto \mathtt{Ident} \\ \mathtt{SP}_1 \\ \ldots \\ \mathtt{SP}_n \end{array} , \rho_{n+1}}$$

Object Class Diagram

The semantics of an object class diagram is a CASL *specification with a sort, the constructor denoting the initial object and the operations corresponding to the field extractors and to the methods of the class itself.*

$$\frac{\begin{array}{l} relations,\ \rho_1 \vdash_{\mathrm{Rels}} \mathtt{rels} \\ fields,\ \rho_1 \vdash_{\mathrm{Fields}} \mathtt{fields},\ \rho_2 \\ vardecs \vdash_{\mathrm{Vars}} \mathtt{vardecs} \\ mets_1,\ \rho_2 \vdash_{\mathrm{Methods}} \mathtt{mets}_1 \\ mets_2,\ \rho_2 \vdash_{\mathrm{Methods}} \mathtt{mets}_2 \end{array}}{(T, relations, vardecs, fields, mets_1, mets_2), \rho \vdash_{\mathrm{Obj}} \begin{array}{l} \textbf{spec } T\mathtt{\_SP} = \\ \textbf{free}\{ \\ \mathtt{Ident\_SP} \textbf{ and } \mathtt{rels} \\ \textbf{then} \\ \textbf{type } T ::= \mathtt{initial} \\ \textbf{vars } \mathtt{vardecs} \\ \textbf{op } \mathtt{ID}\text{: } T \rightarrow \mathtt{Ident} \\ \mathtt{fields} \\ \mathtt{mets}_1 \\ \mathtt{mets}_2 \\ \} \end{array} , \rho_2}$$

if $\rho_1 = \mathsf{AddType}(T, \rho)$

Relationships 1

The semantics of the `use` *and* `inheritance` *relations are expressed using the* ***then*** *con-*

struct (clientship relations are ignored).

$$\frac{}{(\texttt{client}\ PT_1; \ldots; PT_n, \texttt{use}\ T_1; \ldots; T_k, \texttt{inherit}\ T), \rho \vdash_{\text{Rels}} \begin{array}{l} T_1\_\mathtt{SP}\ \ \textbf{and}\ \ldots\ \textbf{and}\ T_k\_\mathtt{SP}\ \textbf{and} \\ T\_\mathtt{SP} \\ \textbf{with}\ T \mapsto \mathsf{CurrentType}(\rho) \\ \textbf{then} \end{array}}$$

Relationships 2
If there is no `inheritance` *relation, the rename clause is not used (clientship relations are ignored).*

$$\frac{}{(\texttt{client}\ PT_1; \ldots; PT_n, \texttt{use} T_1; \ldots; T_k), \rho \vdash_{\text{Rels}} T_1\_\mathtt{SP}\ \textbf{and}\ \ldots\ \textbf{and}\ T_k\_\mathtt{SP}\ \textbf{then}}$$

Fields

$$\frac{\mathit{field}_i, \rho_i \vdash_{\text{Field}} \mathtt{SP}_i, \rho_{i+1} \quad i = 1, \ldots, n}{\mathit{field}_1; \ldots; \mathit{field}_n, \rho_1 \vdash_{\text{Fields}} \mathtt{SP}_1 \ldots \mathtt{SP}_n, \rho_{n+1}}$$

Field
The semantics of a field is an operation that, applied to a value of the object sort returns a value of the field type. This operation, when applied to the initial state, returns the initial value for the field, if it has been specified in the `JTN` *diagram.*

$$\frac{\mathit{exp} \vdash_{\text{Exp}} \mathtt{term}}{(f, T, \mathit{exp}), \rho \vdash_{\text{Field}} \begin{array}{l} \textbf{op}\ f\colon \mathsf{CurrentType}(\rho) \to T; \\ \textbf{axiom}\ f(\mathtt{initial}) = \mathtt{term}; \end{array}, \rho'}$$

if $\rho' = \mathsf{AddField}(f, \rho)$

$$\frac{}{(f, T), \rho \vdash_{\text{Field}} \textbf{op}\ f : \mathsf{CurrentType}(\rho) \to T;, \rho'}$$

if $\rho' = \mathsf{AddField}(f, \rho)$

Method Definitions

$$\frac{metdef_i, \rho \vdash_{\text{Method}} \texttt{metDef}_i \quad i = 1, \ldots, n}{metdef_1; \ldots; metdef_n, \rho \vdash_{\text{Methods}} \texttt{metDef}_1 \ldots\ \texttt{metDef}_n}$$

Method Definition 1

The semantics of a method definition returning a result of type T_r is an operation whose arguments are the arguments of the method plus one of the current sort T. The operation returns a value of sort T plus a value of sort T_r. An axiom says that the identity of the T element returned by the method (representing the new state of the object) has the same identity of the parameter of kind T(representing the object state before the method call). The variables are declared locally to the axioms; their name starts with an "_" character to avoid name clash with the variables declared by the user.

$$\frac{mbody, \rho \vdash_{\text{MBody}} \texttt{AXs}}{(met,\ T_1; \ldots; T_k,\ T_r,\ mbody), \rho' \vdash_{\text{Method}} \begin{array}{l} \textbf{type}\ T\ \texttt{X}\ T_r ::= (\_,\_)(\mathrm{E}_1{:}\, T;\ \mathrm{E}_2{:}\, T_r); \\ \textbf{op}\ met{:}\ T_1\times \ldots \times\ T_k\times\ T\to\ T\ \texttt{X}\ T_r; \\ \textbf{vars}\ \_\mathtt{x}{:}\, T;\_\mathtt{x}_1{:}\, T_1;\ \ldots;\_\mathtt{x}_k{:}\, T_k;\mathtt{res}{:}\ T_r; \\ \textbf{axioms} \\ \mathtt{ID}(\mathrm{E}_1(met(\_\mathtt{x}_1,\ \ldots,\ \_\mathtt{x}_k,\_\mathtt{x})))\ = \mathtt{ID}(\_\mathtt{x}); \\ \texttt{AXs} \end{array}}$$

if $\mathsf{CurrentType}(\rho) = T$ and $\rho' = \mathsf{SetReturnType}(met,\ T_r,\rho)$

Method Definition 2

The semantics of a method definition with no returned result is an operation whose arguments are the arguments of the method plus one of the current sort. It returns a value of the current sort (with the same identity of the argument of type T).

$$\frac{mbody, \rho' \vdash_{\text{MBody}} \texttt{AXs}}{(met,\ T_1; \ldots; T_k, \texttt{void}, mbody), \rho \vdash_{\text{Method}} \begin{array}{l} \textbf{op}\ met{:}\ T_1\times \ldots \times\ T_k\times\ T\to\ T; \\ \textbf{vars}\ \_\mathtt{x}{:}\, T;\_\mathtt{x}_1{:}\, T_1;\ \ldots\ ;\_\mathtt{x}_k{:}\, T_k; \\ \textbf{axiom}\ \mathtt{ID}(met(\_\mathtt{x}_1,\ \ldots\ ,\ \_\mathtt{x}_k,\_\mathtt{x}))\ = \mathtt{ID}(\_\mathtt{x}); \\ \texttt{AXs} \end{array}}$$

if $\mathsf{CurrentType}(\rho) = T$ and $\rho' = \mathsf{SetReturnType}(met,\ \texttt{void},\rho)$

Method Body

The semantics of a method body is analogous to those of an operation body, and returns a set of CASL *axioms.*

$$\frac{metcase_i, \rho_i \vdash_{\text{MCase}} \texttt{AXs}_i, \rho_{i+1} \quad i = 1, \ldots, k}{metcase_1; \ldots; metcase_k, \rho \vdash_{\text{MBody}} \begin{array}{l}\textbf{preds} \\ \texttt{NoMatch}_{[pt_1]}: \ldots; \\ \ldots \\ \texttt{NoMatch}_{[pt_n]}: \ldots; \\ \textbf{axioms} \\ NoMatchDef_1 \ldots NoMatchDef_n \\ \texttt{AXs}_1 \ldots \texttt{AXs}_k\end{array}}$$

if $\rho_1 = \mathsf{SetPattAx}(\mathbf{true}, \rho)$

$\texttt{NoMatch}_{[pt]}$ are the predicates checking whether an element matches pt, introduced before for the operation bodies.

Method Case

$$\frac{\begin{array}{l} stms_i \rho \vdash_{\text{Stmns}} \texttt{Ax}_i, \texttt{\_st}_i \\ bexp_i \vdash_{\text{Exp}} \texttt{bterm}_i \ i = 1, \ldots, m \end{array}}{met(pt_1; \ldots; pt_n), (bexp_1, exp_1); \ldots; (bexp_m, exp_m), \rho \vdash_{\text{MCase}} \texttt{AXs}_1 \ldots \texttt{AXs}_m, \rho'}$$

if for $i = 1, \ldots, m$
$\texttt{Ax} = \mathsf{GetPattAx}(\rho)$
$\texttt{AXs}_i \equiv \texttt{Ax} \wedge \texttt{bterm}_1 = \mathbf{false} \wedge \ldots \wedge \texttt{bterm}_{i-1} = \mathbf{false} \wedge$
$\texttt{bterm}_i = \mathbf{true} \wedge \texttt{x}_1 = pt_1 \wedge \ldots \wedge \texttt{x}_n = pt_n \Rightarrow met(\texttt{x}_1; \ldots; \texttt{x}_n, \texttt{\_x}) = \mathsf{result} \wedge$;
if $\mathsf{ReturnType}(\rho) \neq \texttt{void}$ then $\mathsf{result} = (\texttt{res}(\texttt{\_st}_i), \texttt{\_st}_i)$ else $\mathsf{result} = \texttt{\_st}_i$.
$\rho' = \mathsf{SetPattAx}(\rho, \texttt{Ax} \wedge \texttt{NoMatch}_{[pt_1]}(\texttt{x}_1) \wedge \ldots \wedge \texttt{NoMatch}_{[pt_n]}(\texttt{x}_n))$

Statements

The semantics of a statement list is a CASL *axiom. The semantics of a statement is an axiom and a new variable, describing the state of the system after such statement has been executed. We assume to have an infinite sequence of state variables (of the sort of the current object class), the first one being* $\texttt{\_st}_0$*, and a next operation over them (*$next(\texttt{\_st}_i) = \texttt{\_st}_{i+1}$*). For simplicity, we consider only assignments of fields, forgetting, for the moment, assignments of auxiliary variables. This allows to simplify the state local to a method call, as we do not have to specify, for each new state, which variables change their value and which do not.*

$$\frac{\begin{array}{l} \texttt{\_st}, stm_1, \rho \vdash_{\text{Stmn}} \texttt{Ax}_1, \texttt{\_st}_1 \\ \ldots \\ \texttt{\_st}_{n-1}, stm_n, \rho \vdash_{\text{Stmn}} \texttt{Ax}_n, \texttt{\_st}_n \end{array}}{stm_1; \ldots; stm_n, \rho \vdash_{\text{Stmns}} \texttt{Ax}_1 \wedge \ldots \wedge \texttt{Ax}_n, \texttt{\_st}_n}$$

Field Assignment

The semantics of a field assignment is an axiom and a variable denoting the new state. The axiom says that the value of the field in the new state is the value of the expression, while the values of the other fields of the object are unchanged.

$$\frac{\texttt{\_st}, exp, \rho \vdash_{\overline{\text{MExp}}} \texttt{term}, \texttt{\_st}'}{\texttt{\_st}, f = exp, \rho \vdash_{\overline{\text{Stmn}}} f(\texttt{\_st}'') = \texttt{term} \wedge \mathsf{Unchanged}(fnames, \texttt{\_st}'', \texttt{\_st}'), \texttt{\_st}''}$$

if $\texttt{\_st}''= \text{next}(\texttt{\_st}')$ and
$\mathsf{FieldNames}(\mathsf{CurrentType}(\rho),\rho)= g_1; \ldots ; g_i; f; g_{i+1} g_n$, $fnames= g_1; \ldots ; g_i, g_{i+1} g_n$,

the formula $\mathsf{Unchanged}$ is defined by:
$\mathsf{Unchanged}(g_1; \ldots ; g_n, x, x') \equiv g_1(x) = g_1(x') \wedge \ \ldots \ \wedge g_n(x) = g_n(x')$

Return Assignment

The semantics of the assignment to the return variable is an axiom,saying that the value of the returned variable is the value of the expression, and a new state.

$$\frac{\texttt{\_st}, exp, \rho \vdash_{\overline{\text{MExp}}} \texttt{term}, \texttt{\_st}'}{\texttt{\_st}, \texttt{res} = exp, \rho \vdash_{\overline{\text{Stmn}}} \texttt{res} = \texttt{term}, \texttt{\_st}'}$$

Method Call

The semantics of a method call statement is the new state returned by the semantics of the method call expression.

$$\frac{\texttt{\_st}, met(exp_1; \ldots ; exp_n), \rho \vdash_{\overline{\text{MExp}}} \texttt{term}, \texttt{\_st}'}{\texttt{\_st}, met(exp_1; \ldots ; exp_n), \rho \vdash_{\overline{\text{Stmn}}} \texttt{\_st}'}$$

4.3.4 Method Expressions

Field extractor

The semantics of a field extraction expression is the application of the field operation to the current state. It does not change the state.

$$\frac{}{\_\mathtt{st}, f, \rho \vdash_{\text{MExp}} f(\_\mathtt{st}), \_\mathtt{st}}$$

Constructor

The semantics of a constructor expression is the constructor applied to the semantics of the internal expressions. The state may be changed by the evaluation of the arguments.

$$\frac{\_\mathtt{st}_i, exp_i, \rho \vdash_{\text{MExp}} \mathtt{term}_i, \_\mathtt{st}_{i+1}, i = 1, \ldots, n}{con(exp_1; \ldots; exp_n), \_\mathtt{st}_1 \vdash_{\text{MExp}} con(\mathtt{term}_1, \ldots, \mathtt{term}_n), \_\mathtt{st}_{n+1}}$$

Operation Call

The semantics of the evaluation of an operation is the value of the operation applied to the semantics of the arguments plus a new state variable (the operation may have side-effects, modifying the state).

$$\frac{\_\mathtt{st}_i, exp_i, \rho \vdash_{\text{MExp}} \mathtt{term}_i, \_\mathtt{st}_{i+1} \ i = 1, \ldots, n}{opn(exp_1; \ldots; exp_n), \_\mathtt{st}_1, \rho \vdash_{\text{MExp}} opn(\mathtt{term}_1, \ldots, \mathtt{term}_n), \_\mathtt{st}_{n+1}}$$

Method Call

The semantics of a method call is the value and the new state returned by the corresponding CASL *method applied to the semantics of the arguments and to the state resulting from the evaluation of the arguments.*

$$\frac{\_\mathtt{st}_i, exp_i, \rho \ \vdash_{\text{MExp}} \mathtt{term}_i, \_\mathtt{st}_{i+1} \ for \ i = 1, \ldots, n}{\_\mathtt{st}_1, met(exp_1; \ldots; exp_n), \rho \ \vdash_{\text{MExp}} \begin{array}{l} \mathrm{E}_1(met(\mathtt{term}_1, \ \ldots \ , \ \mathtt{term}_n, \ \_\mathtt{st}_{n+1})), \\ \mathrm{E}_2(met(\mathtt{term}_1, \ \ldots \ , \ \mathtt{term}_n, \_\mathtt{st}_{n+1})) \end{array}}$$

if $\mathsf{ReturnType}(\rho) \neq \mathtt{void}$

$$\frac{\_\mathtt{st}_i, exp_i, \rho \vdash_{\text{MExp}} \mathtt{term}_i, \_\mathtt{st}_{i+1} \quad i = 1, \ldots, n}{\_\mathtt{st}_1, met(exp_1; \ldots; exp_n), \rho \ \vdash_{\text{MExp}} met(\mathtt{term}_1, \ldots, \mathtt{term}_n), \_\mathtt{st}_{n+1}}$$

if $\mathsf{ReturnType}(\rho) = \mathtt{void}$

4.3.5 Process Class

The semantics of a process diagram is a CASL specification describing an *lts*=(*State*,*Label*,$\rightarrow$).

The elements of the set *State* are determined by the state constructors listed in the behaviour diagram of the class.

The elements of the set *Label* represent the method calls and the messages over the asynchronous and synchronous channels.

The definition of the transition predicate $\rightarrow$ comes directly from the behaviour diagram; for each node of the behaviour diagram, corresponding to a state, we transform the alternatives into axioms describing the transitions starting from that state.

The CASL specification of a process class T consists of a sort T with the constructors and a series of axioms defining the transition predicate $\rightarrow$. The labels terms are the ones defined by the `LabSP` specification that is used by each process class because we assume that the `JTN` specification is correct, thus we are guaranteed that each process calls only the method of the referenced objects and uses only the names of the channels declared in its interface.

Process Class Diagrams

The semantics of the list of the process classes returns the specification `LabSP` *of the labels describing method calls and the communications over the synchronous and asynchronous channels and the specification of the various process classes. The specification* `ValueSP` *contains all the values appearing in the labels of the processes, that is all the elements of the already declared datatypes and object sorts.*

$$\frac{pdia_i, \rho \vdash_{\overline{\text{Proc}}} \mathtt{SP}_i \quad i = 1, \ldots, n}{pdia_1, \ldots, pdia_n, \rho \vdash_{\overline{\text{Procs}}} \mathit{Values}\ \mathit{Labels}\ \mathtt{SP}_1\ \ldots \mathtt{SP}_n}$$

if $\mathsf{Sorts}(\rho) = T_1; \ldots; T_n$

$$\mathit{Values} \equiv \begin{cases} \textbf{spec } \mathtt{ValueSP} = \\ T_1\mathtt{\_SP} \ \textbf{ and } \ldots \textbf{ and } T_n\mathtt{\_SP} \textbf{ and } \mathtt{Ident\_SP} \\ \textbf{then} \\ \textbf{freetype}\{ \ \mathtt{value} ::= \textbf{sort } T_1 \mid \ldots \mid \textbf{sort } T_n \mid \\ \textbf{sort } \mathtt{Ident} \mid \mathtt{nomess};\} \\ \textbf{end} \end{cases}$$

$$
Labels \equiv \begin{cases}
\textbf{spec}\ \texttt{LabSP} = \\
\texttt{ValueSP}\ \textbf{and}\ \texttt{String\_SP} \\
\textbf{then} \\
\textbf{freetypes}\{\ \texttt{MethCallS} ::= \_.\_(\_)(\texttt{Ident}, \texttt{String},\texttt{value})\ \mid \\
\_ \leftarrow \_.\_(\_)(\texttt{value},\texttt{Ident},\texttt{String},\texttt{value}) \mid \\
\texttt{Label} ::= \texttt{!}\_(\_)(\texttt{String},\texttt{value})\ \mid \\
\%\%\ \text{asynchronous and synchronous non blocking write} \\
\texttt{?}\_(\_)(\texttt{String},\texttt{value})\ \mid \\
\%\%\ \text{asynchronous and synchronous non blocking read} \\
\texttt{No?}(\_)(\texttt{String})\ \mid \%\%\ \text{failed asynchronous and synchronous read} \\
\_\ \texttt{!}\_(\_)(\texttt{Ident},\texttt{String},\texttt{value})\ \mid \%\%\ \text{asynchronous write with name} \\
\_\ \texttt{?}\_(\_)(\texttt{Ident},\texttt{String},\texttt{value})\ \mid \%\%\ \text{asynchronous read with name} \\
\texttt{!!}\_(\_)(\texttt{String},\texttt{value})\ \mid \%\%\ \text{synchronous blocking write} \\
\texttt{??}\_(\_)(\texttt{String},\texttt{value})\ \mid \%\%\ \text{synchronous blocking read} \\
\texttt{resumed}(\texttt{String})\ \mid \%\%\ \text{resumed after blocking}\ \textbf{end}
\end{cases}
$$

In the `JTN` specification, a process can perform asynchronous (non blocking) or synchronous (blocking) communications with a message or a signal exchange. As such activities can be unsuccessful (the communication completes without a value or the synchronization cannot be immediately completed), some process states have a choice among several alternatives. Thus we can try to read over other asynchronous channels if the previous ones have no available messages, or to check for another partner if the previous ones are not ready to immediately synchronize. This is rendered in CASL by defining transitions that represent both the successful and the unsuccessful communications requiring a certain variety of labels.

The asynchronous communication requires two labels for the asynchronous read and write operations (`?`, `!`) plus one label for the uncompleted asynchronous read operation (`No!`)(an asynchronous write is always possible!).

The synchronous communication requires two labels for the non-blocking successful (test-and-set) read (`?`) and write (`!`) and unsuccessful (`No?`, `No!`) operations over synchronous channels plus the labels for the blocking read (`??`) and write (`!!`). A special label describes the termination of the blocking synchronization (`resumed`). A label (`call`) describes the call of an object method.

Process Class

The semantics of a process class is the specification of the process identities, states and labels plus the specification of the state activities. The relations and the interface diagram have no semantic value (they just contain information used to check the static correctness of the process specification). For example, the list of object references has no semantic coun-

terpart: they are not part of the process state; the object references used in the transitions are just strings (see the definition of sort `MethCallS`*). We assume that the* `JTN` *specifications are statically correct, thus the method calls in the behavioural diagram can only be sent to the objects listed in the interface, and the names and the values of the method calls are correct w.r.t. the classes of these objects.*

Let us recall that inheritance between processes is treated during the pre-processing, that, given a class P that inherits from P', simply duplicates the diagrams for P' and adds the new channels and objects references appearing in the interface plus the new activities declared for P into the behaviour diagram of P'. This result in the new, complete, interface and body diagrams for class P.

The semantics of the behaviour diagram needs the information returned by the semantics of the interface diagram, because the list of the synchronous and asynchronous channel identifiers is used in the semantics of the alternatives with multiple communication capabilities, to recognize if an identifier denotes a synchronous or an asynchronous channel.

$$\frac{\begin{array}{l} \textit{relations} \vdash_{\text{Rels}} \texttt{rels} \\ \textit{interface},\ \rho \vdash_{\text{Inter}} \rho' \\ \textit{beh},\ \rho' \vdash_{\text{Behav}} \texttt{vardecs},\ \texttt{cons},\ \texttt{Ax}_{\mathit{Init}},\ \texttt{beh} \end{array}}{(T, \textit{relations}, \textit{interface}, \textit{beh}), \rho \vdash_{\text{Proc}} \begin{array}{l} \textbf{spec}\ T\texttt{\_SP}\ = \\ \textbf{free}\{\ \texttt{Ident\_SP}\ \textbf{and}\ \texttt{LabSP}\ \textbf{and}\ \texttt{rels} \\ \textbf{then} \\ \textbf{dtype}\ T ::= \texttt{cons};\mathsf{Susp}(\texttt{Ident},\ T); \\ \textbf{vars}\ \texttt{\_st},\texttt{\_st}'\texttt{:}\ T;\ \texttt{xlab: Label}; \\ \texttt{vardecs}; \\ \texttt{Ax}_{\mathit{Init}} \\ \texttt{beh} \\ \} \end{array}}$$

Process Interface

The semantics of the process interface returns a new environment enriched by the channel declarations.

$$\frac{}{(\textit{obrefs}, \textit{asynchs}, \textit{synchs}), \rho \vdash_{\text{Inter}} \rho'}$$

if $\rho' = \mathsf{AddSchanNames}(\textit{synchs},\ \mathsf{AddAchanNames}(\textit{asynchs},\ \rho))$

Behaviour Diagram

The semantics of the behaviour diagram is the specification of the process state constructors *plus the declarations of the variables used in the behaviour graph and the axioms defining the transition predicate.*

$$\frac{\begin{array}{l} cons, \rho \vdash_{\overline{\text{PCons}}} \texttt{cons} \\ vardecs \vdash_{\overline{\text{Vars}}} \texttt{vardecs} \\ exps, \rho \vdash_{\overline{\text{Init}}} \texttt{Ax}_{Init} \\ bgraph, \rho \vdash_{\overline{\text{BGraph}}} \texttt{beh} \end{array}}{(cons, vardecs, exps, bgraph), \rho \vdash_{\overline{\text{Behav}}} \texttt{vardecs}, \texttt{cons}, \texttt{Ax}_{Init}, \texttt{beh}}$$

Process State Constructor List

The semantics of a `JTN` *constructor list is the list of the* CASL *constructors returned by the semantics of each* `JTN` *constructor.*

$$\frac{con_i, \rho \vdash_{\overline{\text{PCon}}} \texttt{con}_i \quad i = 1, \ldots, n}{con_1; \ldots; con_n, \rho \vdash_{\overline{\text{PCons}}} \texttt{con}_1 \mid \ldots \mid \texttt{con}_n}$$

Process State Constructor

The semantics of a process state constructor is different from the semantics of a datatype constructor, because each process state constructor has an extra integer arguments that is used when giving the axioms for the process activity, to render the ordering of the transitions leaving a state (it is explained in more details in the following). Furthermore, every process has its own identity, that is an additional argument for each constructor. The `ID` *operation is the identity extractor whose value does not change after a transition.*

$$\frac{}{con(T_1; \ldots; T_n) \vdash_{\overline{\text{PCon}}} con(\texttt{ID} : \texttt{Ident}; T_1; \ldots; T_n; \texttt{int})}$$

Initial State

The initial state is a particular term constructed with `initial`*. It corresponds to an axiom defining the values of the components in the initial state.*

$$\frac{exp_i \vdash_{\overline{\text{MExp}}} \texttt{exp}_i}{exp_1; \ldots; exp_n, \rho \vdash_{\overline{\text{Init}}} \begin{array}{l} \textbf{vars}\ \_x\text{: CurrentType}(\rho) \\ \textbf{pred}\ \texttt{IsInitial}\text{: CurrentType}(\rho)\ ; \\ \textbf{axioms} \\ \texttt{IsInitial}(\_x)\ \Rightarrow \_x = \texttt{initial}(\texttt{exp}_1, \ldots, \texttt{exp}_n, 0) \end{array}}$$

Behaviour Graph

The semantics of a behaviour graph is the list of the axioms that are the semantics of each single state case.

$$\frac{state\_case_i, \rho \vdash_{\text{StateCase}} \mathtt{Ax}_i \ \ i = 1, \ldots, n}{state\_case_1; \ldots; state\_case_n, \rho \vdash_{\text{BGraph}} \mathtt{Ax}_1 \ldots \mathtt{Ax}_n}$$

State case

The semantics of a statecase is analogous to that of an operation body, and returns a set of CASL *axioms.*

$$\frac{transcase_i, \rho_i \vdash_{\text{TranCase}} \mathtt{AXs}_i, \rho_{i+1} \quad i = 1, \ldots, k}{transcase_1; \ldots; transcase_k, \rho \vdash_{\text{StateCase}} \begin{array}{l} \textbf{preds} \\ \mathtt{NoMatch}_{[pt_1]}\colon \ldots; \\ \ldots \\ \mathtt{NoMatch}_{[pt_n]}\colon \ldots; \\ \textbf{axioms} \\ NoMatchDef_1 \ldots NoMatchDef_n \\ \mathtt{AXs}_1 \ldots \mathtt{AXs}_k \end{array}}$$

if $\rho_1 = \mathsf{SetPattAx}(\textbf{true}, \rho)$

$\mathtt{NoMatch}_{[pt]}$ are the predicates checking whether an element matches pt, introduced before for the operation bodies.

Transition Case

The semantics of a transition case is the list of axioms describing some transition capabilities. The premises of the axioms require that the previous pattern are not matched and and the previous conditions for the same pattern are not satisfied.

$$\frac{\begin{array}{l} con(pt_1; \ \ldots \ ; \ pt_n), labst_i, \ \rho \vdash_{\text{TRANSALT}} \\ \mathtt{Ax}_i \\ bexp_i \vdash_{\text{Exp}} \mathtt{bterm}_i \ \ i = 1, \ldots, m \end{array}}{con(pt_1; \ldots; pt_n), (bexp_1, labst_1); \ldots; (bexp_m, labst_m), \rho \vdash_{\text{TranCase}} \mathtt{AXs}_1 \ldots \mathtt{AXs}_m, \rho'}$$

if $i = 1, \ldots, m$

$\mathtt{Ax} = \mathsf{GetPattAx}(\rho)$

$\mathtt{AXs}_i \equiv \mathtt{Ax} \wedge \mathtt{bterm}_1 = \textbf{false} \wedge \ldots \wedge \mathtt{bterm}_{i-1} = \textbf{false} \wedge$

$\mathtt{bterm}_i = \textbf{true} \wedge \mathtt{x}_1 = pt_1 \wedge \ldots \wedge \mathtt{x}_n = pt_n \Rightarrow \mathtt{Ax}_i;$

$\rho' = \mathsf{SetPattAx}(\rho, \mathtt{Ax} \wedge \mathtt{NoMatch}_{[pt_1]}(\mathtt{x}_1) \wedge \ldots \wedge \mathtt{NoMatch}_{[pt_n]}(\mathtt{x}_n))$

4.3.5.0.1 Single Transition Alternative

Method call

The semantics of a transition alternative consisting of a method call (with or without return value) is the axiom defining the transition capability labelled by the method call.

pCall1

$$\frac{\begin{array}{c} exp_i \vdash_{\overline{\text{MExp}}} \texttt{exp}_i \;\; i = 1 \ldots k \\ exp_j{}' \vdash_{\overline{\text{MExp}}} \texttt{exp}_j{}' \;\; j = 1 \ldots m \end{array}}{\begin{array}{l} (con(pt_1; \ldots ; pt_n), (oid.met(exp_1; \ldots ; exp_k), con'(exp_1{}'; \ldots ; exp_m{}'))), \rho \vdash_{\overline{\text{TRANSALT}}} \\ con(pid, \mathtt{x}_1, \ldots, \mathtt{x}_n, 0) \xrightarrow{\texttt{call}(oid.met(\texttt{exp}_1,\ldots,\texttt{exp}_k))} con'(pid, \texttt{exp}_1{}', \ldots, \texttt{exp}_m{}', 0) \end{array}}$$

pCall2

$$\frac{\begin{array}{c} exp_i \vdash_{\overline{\text{MExp}}} \texttt{exp}_i \;\; i = 1, \ldots, k \\ exp_j{}' \vdash_{\overline{\text{MExp}}} \texttt{exp}_j{}' \;\; j = 1 \ldots m \end{array}}{\begin{array}{l} (con(pt_1; \ldots ; pt_n), \; (x{=} \; oid.met(exp_1; \ldots ; exp_k), con'(exp_1{}'; \ldots ; exp_m{}'))), \; \rho \vdash_{\overline{\text{TRANSALT}}} \\ con(pid, \mathtt{x}_1, \ldots, \mathtt{x}_n, 0) \xrightarrow{\texttt{call}(x \leftarrow oid.met(\texttt{exp}_1,\ldots,\texttt{exp}_k))} con'(\texttt{exp}_1{}'; \ldots ; \texttt{exp}_m{}', 0) \end{array}}$$

Asynchronous communication

Analogous for the alternative corresponding to a write operation over an asynchronous channel. If the communication is named, it is specified a receiver for the message.

pWrite

$$\frac{\begin{array}{c} exp \vdash_{\overline{\text{MExp}}} \texttt{exp} \\ exp_i{}' \vdash_{\overline{\text{MExp}}} \texttt{exp}_i{}' \;\; i = 1, \ldots, m \end{array}}{\begin{array}{l} (con(pt_1; \ldots ; pt_n), (!\, ach\_id(exp), con'(exp_1{}'; \ldots ; exp_m{}'))), \rho \vdash_{\overline{\text{TRANSALT}}} \\ con(pid, \mathtt{x}_1, \ldots, \mathtt{x}_n, 0) \xrightarrow{!ach\_id(\texttt{exp})} con'(pid, \texttt{exp}_1{}'; \ldots ; \texttt{exp}_m{}', 0) \end{array}}$$

pNamedWrite1

$$\frac{\begin{array}{c} exp \vdash_{\overline{\text{MExp}}} \texttt{exp} \\ exp_i{}' \vdash_{\overline{\text{MExp}}} \texttt{exp}_i{}' \;\; i = 1, \ldots, m \end{array}}{\begin{array}{l} (con(pt_1; \ldots ; pt_n), (pid!\, ach\_id(exp), con'(exp_1{}'; \ldots ; exp_m{}'))), \rho \vdash_{\overline{\text{TRANSALT}}} \\ con(pid, \mathtt{x}_1, \ldots, \mathtt{x}_n, 0) \xrightarrow{pid!ach\_id(\texttt{exp})} con'(pid, \texttt{exp}_1{}'; \ldots ; \texttt{exp}_m{}', 0) \end{array}}$$

pNamedWrite2

$$\frac{\begin{array}{l} \mathit{exp}_1 \vdash_{\text{MExp}} \mathit{exp}_1 \\ \mathit{exp}_2 \vdash_{\text{MExp}} \mathit{exp}_2 \\ \mathit{exp}_i{}' \vdash_{\text{MExp}} \mathtt{exp}_i{}' \;\; i = 1, \ldots, m \end{array}}{\begin{array}{l} (\mathit{con}(\mathit{pt}_1; \ldots ; \mathit{pt}_n),(\mathit{exp}_1 \,!\, \mathit{ach\_id}(\mathit{exp}_2),\mathit{con}'(\mathit{exp}_1{}'; \ldots ; \mathit{exp}_m{}'))),\rho \vdash_{\text{TRANSALT}} \\ \mathit{con}(\mathit{pid}, \mathtt{x}_1, \ldots, \mathtt{x}_n, 0) \xrightarrow{\mathit{exp}_1 ! \mathit{ach\_id}(\mathit{exp}_2)} \mathit{con}'(\mathit{pid}, \mathtt{exp}_1{}'; \ldots ; \mathtt{exp}_m{}', 0) \end{array}}$$

pRead

$$\frac{\mathit{exp}_i{}' \vdash_{\text{MExp}} \mathtt{exp}_i{}' \;\; i = 1, \ldots, m}{\begin{array}{l} (\mathit{con}(\mathit{pt}_1; \ldots ; \mathit{pt}_n),(?\,\mathit{ach\_id}(x),\mathit{con}'(\mathit{exp}_1{}'; \ldots ; \mathit{exp}_m{}'))),\rho \vdash_{\text{TRANSALT}} \\ \mathit{con}(\mathit{pid}, \mathtt{x}_1, \ldots, \mathtt{x}_n, 0) \xrightarrow{?\mathit{ach\_id}(x)} \mathit{con}'(\mathit{pid}, \mathtt{exp}_1{}'; \ldots ; \mathtt{exp}_m{}', 0) \end{array}}$$

pNamedRead

$$\frac{\mathit{exp}_i{}' \vdash_{\text{MExp}} \mathtt{exp}_i{}' \;\; i = 1, \ldots, m}{\begin{array}{l} (\mathit{con}(\mathit{pt}_1; \ldots ; \mathit{pt}_n),(x_1 ?\, \mathit{ach\_id}(x_2),\mathit{con}'(\mathit{exp}_1{}'; \ldots ; \mathit{exp}_m{}'))),\rho \vdash_{\text{TRANSALT}} \\ \mathit{con}(\mathit{pid}, \mathtt{x}_1, \ldots, \mathtt{x}_n, 0) \xrightarrow{x_1 ? \mathit{ach\_id}(x_2)} \mathit{con}'(\mathit{pid}, \mathtt{exp}_1{}'; \ldots ; \mathtt{exp}_m{}', 0) \end{array}}$$

Synchronous communication

The semantics of an alternative consisting of an operation over a synchronous channel returns three axioms expressing that the operations can be immediately completed or the process can be suspended; from the suspended state, the process can be resumed passing into the final state of the alternative. Notice that the communication can also be a signal exchange (in this case the expression is just the `nomess` *value)*

pSynchWrite

$$\frac{\begin{array}{l} \mathit{exp} \vdash_{\text{MExp}} \mathtt{exp} \\ \mathit{exp}_i{}' \vdash_{\text{MExp}} \mathtt{exp}_i{}' \;\; i = 1, \ldots, m \end{array}}{\begin{array}{l} (\mathit{con}(\mathit{pt}_1; \ldots ; \mathit{pt}_n),(\,!\, \mathit{sch\_id}(\mathit{exp}),\mathit{con}'(\mathit{exp}_1{}'; \ldots ; \mathit{exp}_m{}'))),\rho \\ \vdash_{\text{TRANSALT}} \\ \mathit{con}(\mathit{pid}, \mathtt{x}_1, \ldots, \mathtt{x}_n, 0) \xrightarrow{!!\mathit{sch\_id}(\mathsf{exp})} \mathsf{Susp}(\mathit{sch\_id}, \mathit{con}'(\mathit{pid}, \mathtt{exp}_1{}'; \ldots ; \mathtt{exp}_m{}', 0)), \\ \mathit{con}(\mathit{pid}, \mathtt{x}_1, \ldots, \mathtt{x}_n, 0) \xrightarrow{!\mathit{sch\_id}(\mathsf{exp})} \mathit{con}'(\mathit{pid}, \mathtt{exp}_1{}'; \ldots ; \mathtt{exp}_m{}', 0), \\ \mathsf{Susp}(\mathit{sch\_id}, \mathit{st}) \xrightarrow{\mathtt{resumed}(\mathit{sch\_id})} \mathit{st} \end{array}}$$

Analogous for the synchronous read operation.

pSynchRead

$$\frac{exp_i' \vdash_{\text{MExp}} \mathtt{exp}_i' \; i = 1, \ldots, m}{\begin{array}{l}(con(pt_1; \ldots ; pt_n),(? \; sch\_id(x), con'(exp_1'; \ldots ; exp_m'))), \rho \vdash_{\text{TRANSALT}} \\ con(pid, \mathtt{x}_1, \ldots, \mathtt{x}_n, 0) \xrightarrow{??sch\_id(x)} \mathsf{Susp}(sch\_id, con'(pid, \mathtt{exp}_1'; \ldots ; \mathtt{exp}_m', 0)), \\ con(pid, \mathtt{x}_1, \ldots, \mathtt{x}_n, 0) \xrightarrow{?sch\_id(x)} con'(pid, \mathtt{exp}_1'; \ldots ; \mathtt{exp}_m', 0), \\ \mathsf{Susp}(sch\_id, st) \xrightarrow{\mathtt{resumed}(sch\_id)} st\end{array}}$$

4.3.5.0.2 Multiple transition alternatives

Synchronous Transition List

If the alternative associated with the transition case for the pattern $con(pt_1; \ldots ; pt_n)$ is a list of synchronous read/write operations, the semantics describes that the process from the state $con(pt_1, \ldots ,pt_n,0)$ tries to perform each non-blocking operation until one can be completed; the uncompleted execution of the j-th operation of the alternative leads from state $con(pt_1, \ldots ,pt_n,j\text{-}1)$ to $con(pt_1, \ldots ,pt_n,j)$ where it can perform the (j+1)-th operation in the list of th alternatives. If no operation can be immediately performed, the process performs the blocking operation corresponding to the last transition of the list: notice that the uncompleted execution of the k-th operation of the alternative from state $con(pt_1, \ldots ,pt_n,k\text{-}1)$ returns to the same state, where the only one capability is to try to perform the k-th operation again.

$$\frac{exp_i^j \vdash_{\text{MExp}} \mathtt{exp}_i^j \, j = 1, \ldots, k, i = 1, \ldots, m_j}{\begin{pmatrix} con(pt_1, \ldots ,pt_n), \\ (l_1, con_1(exp_1^1, \ldots ,exp_{m_1}^1)); \\ \ldots; \\ (l_k, \; con_k(exp_1^k, \ldots ,exp_{m_k}^k) \end{pmatrix}, \rho \vdash_{\text{TRANSALT}} \mathtt{Ax}_1, \ldots, \mathtt{Ax}_k}$$

if for $i = 1, \ldots, k-1$ $\mathtt{IsSynch}(l_i, \rho)$ and

$\mathtt{Ax}_i \equiv$

pMultSynch1

$con(pid, \mathtt{x}_1, \ldots, \mathtt{x}_n, i-1) \xrightarrow{l_i} con_i(pid, \mathtt{exp}_1{}^i, \ldots, \mathtt{exp}_{m}{}_i^i, 0);$

$con(pid, \mathtt{x}_1, \ldots, \mathtt{x}_n, i-1) \xrightarrow{\mathsf{Failure}(l_i)} con(pid, \mathtt{x}_1, \ldots, \mathtt{x}_n, i);$

$\mathtt{Ax}_k \equiv$

pMultSynch2

$con(pid, \mathtt{x}_1, \ldots, \mathtt{x}_n, k) \xrightarrow{\mathsf{Failure}(l_k)} con(pid, \mathtt{x}_1, \ldots, \mathtt{x}_n, k);$

$con(pid, \mathtt{x}_1, \ldots, \mathtt{x}_n, k-1) \xrightarrow{l_k} con_k(pid, \mathtt{exp}_1^k, \ldots, \mathtt{exp}_{m_k}^k, 0);$

where `IsSynch` is the formula that tests if a label describes a communication over a synchronous channel. It is defined by:

$sch\_id \in$ SchanNames(CurrentType(ρ),ρ) $\Rightarrow$ `IsSynch`($iosch\_id(exp)$, ρ)

Failure: `Label` $\rightarrow$ `Label` is a partial function that returns the label describing the failure of a communication; it is defined by:

- Failure($?ch(x)$) = `No?`(ch)
- Failure($!ch(exp)$) = `No!`(ch)
- Failure($exp_1?ch(exp_2)$) = `No?`(ch)

and $x \in ll$ checks if x belongs to list ll.

Synchronous Transition List with Else

If there is an else transition, the transition labelled by the unsuccessful execution of the last non-else label ends in the final state of the else transition.

$$\frac{\begin{array}{l} exp_i{}' \vdash_{\text{MExp}} \ \mathtt{exp}_i{}' i = 1, \ldots, m \\ exp_i{}^j \vdash_{\text{MExp}} \ \mathtt{exp}_i{}^j j = 1, \ldots, k, i = 1, \ldots, m_j \end{array}}{\begin{pmatrix} con(pt_1, \ \ldots \ ,pt_n), \\ (l_1, con_1(exp_1^1, \ \ldots \ ,exp_{m_1}^{\ 1})); \\ \ldots; \\ (l_k, \ con_k(exp_1^k, \ \ldots \ ,exp_{m_k}^{\ k})), \\ (\mathtt{else}, \ con'(exp_1{}', \ \ldots \ ,exp_m{}'))) \end{pmatrix}, \rho \vdash_{\text{TRANSALT}} \mathtt{Ax}_1 \ldots \mathtt{Ax}_k}$$

if for i = 1, ... , k-1, `IsSynch`(l_i) and

$\mathtt{Ax}_i \equiv$

pMultSynchElse1

$con(pid, \mathtt{x}_1, \ldots, \mathtt{x}_n, i-1) \xrightarrow{l_i} con_i(pid, \mathtt{exp}_1{}^i, \ldots, \mathtt{exp}_{m_i}{}^i, 0);$

$con(pid, \mathtt{x}_1, \ldots, \mathtt{x}_n, i-1) \xrightarrow{\mathsf{Failure}(l_i)} con(pid, \mathtt{x}_1, \ldots, \mathtt{x}_n, i);$

$\mathtt{Ax}_k \equiv$

pMultSynchElse2

$con(pid, \mathtt{x}_1, \ldots, \mathtt{x}_n, k-1) \xrightarrow{l_k} con_k(pid, \mathtt{exp}_1{}^k, \ldots, \mathtt{exp}_{m_k}{}^k, 0);$

$con(pid, \mathtt{x}_1, \ldots, \mathtt{x}_n, k-1) \xrightarrow{\mathsf{Failure}(l_k)} con'(pid, \mathtt{exp}'_1, \ldots, \mathtt{exp}_m{}', 0);$

Asynchronous Transition list

The semantics of a list of asynchronous read operations is a sequence of axioms expressing that the process tries to read from each channel, in the order; if no reading is successful, *the process starts again trying to read from the first channel.*

$$\frac{exp_i^j \vdash_{\overline{\text{MExp}}} \mathtt{exp}_i^j j = 1,\ldots,k\ i = 1,\ldots,m_j}{\begin{pmatrix}(con(\mathtt{x}_1, \ldots, \mathtt{x}_n),\\ (l_1, con_1(exp_1^1, \ldots, exp_{m_1}^1));\\ \ldots;\\ (l_k,\ con_k(exp_1^k, \ldots, exp_{m_k}^k)))\end{pmatrix}, \rho \vdash_{\overline{\text{TRANSALT}}} \mathtt{Ax}_1 \ldots \mathtt{Ax}_k}$$

if for $i = 1,\ldots,k-1$ $\mathtt{IsAsynch}(l_i,\rho)$ and

$\mathtt{Ax}_i \equiv$

pMultAsynch1

$$con(pid, \mathtt{x}_1, \ldots, \mathtt{x}_n, i-1) \xrightarrow{l_i} con_i(pid, \mathtt{exp}_1{}^i, \ldots, \mathtt{exp}_{m_i}{}^i, 0)$$

$$con(pid, \mathtt{x}_1, \ldots, \mathtt{x}_n, i-1) \xrightarrow{\mathsf{Failure}(l_i)} con(pid, \mathtt{x}_1, \ldots, \mathtt{x}_n, i)$$

$\mathtt{Ax}_{k+1} \equiv$

pMultAsynch2

$$con(pid, \mathtt{x}_1, \ldots, \mathtt{x}_n, k-1) \xrightarrow{\mathsf{Failure}(l_k)} con(pid, \mathtt{x}_1, \ldots, \mathtt{x}_n, 0)$$

$$con(pid, \mathtt{x}_1, \ldots, \mathtt{x}_n, k-1) \xrightarrow{l_k} con_k(pid, pt_1^k, \ldots, pt_{m_k}^k, 0)$$

where `IsAsynch` is the formula that checks if a communication is asynchronous:

$\mathtt{IsAsynch}(ioach\_id(exp),\ \rho) \equiv ach\_id \in \mathsf{AchanNames}(\mathsf{CurrentType}(\rho),\rho)$ tests if ach_id is an asynchronous channel for simple asynchronous communications.

$\mathtt{IsBsynch}(exp_1 ioach\_id(exp_2),\ \rho) \equiv ach\_id \in \mathsf{NchanNames}(\mathsf{CurrentType}(\rho),\rho)$ tests if ach_id is an asynchronous broadcast channel.

$\mathtt{IsNsynch}(exp_1 ioach\_id(exp_2),\ \rho) \equiv ach\_id \in \mathsf{BchanNames}(\mathsf{CurrentType}(\rho),\rho)$ tests if ach_id is an asynchronous named channel.

Asynchronous Transition List with else

If there is an else transition, the transition labelled by the unsuccessful execution of the last non-else label ends in the final state of the else transition.

$$\frac{\begin{array}{l} exp_i{}' \vdash_{\overline{\text{MExp}}}\ \mathtt{exp}_i{}'\text{i=1, \ldots ,q} \\ exp_i{}^j \vdash_{\overline{\text{MExp}}}\ \mathtt{exp}_i{}^j\text{j = 1, \ldots ,k, i=1,} \\ \ldots,\ \text{m}_j \end{array}}{\begin{pmatrix} con(\mathtt{x}_1, \ldots, \mathtt{x}_n),\\ (l_1, con_1(exp_1^1, \ldots, exp_{m_1}^1));\\ \ldots;\\ (l_k,\ con_k(exp_1^k, \ldots, exp_{m_k}^k)))\\ (\mathtt{else}, con'(pid, exp_1', \ldots, exp_q' \end{pmatrix}, \rho \vdash_{\overline{\text{TRANSALT}}} \mathtt{Ax}_1 \ldots \mathtt{Ax}_k}$$

if for $i = 1, \ldots, k-1$ $\texttt{IsAsynch}(l_i, \rho)$ and
$\texttt{Ax}_i \equiv$
pMultAsynchElse1
$$con(pid, \texttt{x}_1, \ldots, \texttt{x}_n, i-1) \xrightarrow{l_i} con_i(pid, \texttt{exp}_1{}^i, \ldots, \texttt{exp}^i_{m_i}, 0)$$
$$con(pid, \texttt{x}_1, \ldots, \texttt{x}_n, i-1) \xrightarrow{\mathsf{Failure}(l_i)} con(pid, \texttt{x}_1, \ldots, \texttt{x}_n, i)$$
$\texttt{Ax}_k \equiv$
pMultAsynchElse2
$$con(pid, \texttt{x}_1, \ldots, \texttt{x}_n, k-1) \xrightarrow{\texttt{No}?(ach\_id_k())} con'(pid, \texttt{exp}_1{}', \ldots, \texttt{exp}'_q, 0)$$
$$con(pid, \texttt{x}_1, \ldots, \texttt{x}_n, k-1) \xrightarrow{?ach\_id_k(x_k)} con_k(pid, \texttt{exp}_1{}^k, \ldots, \texttt{exp}_{m}{}^k_k, 0)$$

Asynchronous Named Transition list
The semantics of a list of asynchronous read operations is a sequence of axioms expressing that the process tries to read from each channel, in the order; if no reading is successful, the process starts again trying to read from the first channel.

$$\frac{exp_i{}^j \vdash_{\text{MExp}} \texttt{exp}_i{}^j \, for j = 1, \ldots k, i = 1, \ldots, m_j}{\begin{pmatrix} con(pt_1, \ \ldots \ ,pt_n), \\ (l_1, con_1(exp^1_1, \ \ldots \ ,exp_{m}{}^1_1)); \\ \ldots; \\ (l_k, \ con_k(exp^k_1, \ \ldots \ ,exp_{m}{}^k_k)) \end{pmatrix}, \rho \vdash_{\text{TRANSALT}} \texttt{Ax}_1 \ldots \texttt{Ax}_k}$$

if for $i = 1, \ldots, k-1$ $\texttt{IsNsynch}(l_i, \ \rho)$
$\texttt{Ax}_i \equiv$
pMultNamedElse1
$$con(pid, \texttt{x}_1, \ldots, \texttt{x}_n, i-1) \xrightarrow{l_i} con_i(pid, \texttt{exp}_1{}^i, \ldots, \texttt{exp}_{m}{}^i_i, 0)$$
$$con(pid, \texttt{x}_1, \ldots, \texttt{x}_n, i-1) \xrightarrow{\mathsf{Failure}(l_i)} con(pid, \texttt{x}_1, \ldots, \texttt{x}_n, i)$$
$\texttt{Ax}_{K+1} \equiv$
pMultNamedElse2
$$con(pid, \texttt{x}_1, \ldots, \texttt{x}_n, j, k) \xrightarrow{\mathsf{Failure}(l_k)} con(pid, \texttt{x}_1, \ldots, \texttt{x}_n, 0)$$
$$con(pid, \texttt{x}_1, \ldots, \texttt{x}_n, j, k) \xrightarrow{l_k} con_k(pid, \texttt{exp}_1{}^k, \ldots, \texttt{exp}_{m}{}^k_k, 0)$$

where IsNamedAsynchronous(ach_id, ach_ids) $\equiv$ In(ach_id,n,ach_ids) tests if ach_id is an asynchronous channel for named communications.

4.4 Architecture Diagram

The architecture diagram corresponds to the specification of the generalized label transition system *glts*=(*Info*,*State*,*Label*,$\rightarrow$) that models the overall system described by the `JTN` specification.

The elements of the set *State* are n-tuple of components, that are the objects, the processes and the channels composing the system.

The elements of the set *Label* describe the internal action (usually omitted) or the message passing with external entities (files or sockets).

The elements of *Info* (that are lists of labels and process identifiers) describe, for each transition, which internal components participated in the global transition and with which activity.

The transition predicate $\rightarrow$ is defined by the cooperations of the architecture diagram.

The Casl counterpart consists of a specification SYSTEM defining a dynamic sort `system`, whose only constructor is an n-upla with a component for each object or process component in the architecture diagram plus a component for each channel, a sort `lab_system` for the labels describing interactions with the external world, that is for the read and write operation over external entities, a sort `info_system` describing the transitions performed by the subcomponents.

Architecture Diagram
The architecture diagram corresponds to a specification defining the dynamic sort of the system states, the sort of the labels and containing the axioms that define the system transition capabilities.

$$\frac{\begin{array}{l} comps,\ \rho \vdash_{\overline{\text{Comps}}} \texttt{usedtypes}, \texttt{objcomp}, \texttt{prcomp}, \rho' \\ objcoops,\ \rho' \vdash_{\overline{\text{OCoops}}} \texttt{AXs}_{Ob} \\ Acoops,\ \rho' \vdash_{\overline{\text{ACoops}}} \texttt{statecomp2}, \texttt{AXs}_{Ach} \\ scoops,\ \rho' \vdash_{\overline{\text{SCoop}}} \texttt{statecomp3}, \texttt{AXs}_{Sch} \end{array}}{\begin{array}{l} (comps,\ (objcoops, \\ Acoops,\ scoops)) \end{array} \vdash_{\overline{\text{Arch}}} \begin{array}{l} \textbf{spec}\ \text{SYSTEM} \ = \\ \textbf{free}\{ \\ \texttt{usedtypes}\ \ \textbf{and}\ \texttt{Ident\_SP}\ \textbf{and}\ \texttt{LabSP} \\ \textbf{then} \\ \textbf{types} \\ \texttt{Psort} \\ \texttt{system} ::= \_;\ldots;\_: (\texttt{objcomp};\texttt{prcomp};\texttt{statecomp2};\texttt{statecomp3}) \\ \texttt{info\_system} ::= \text{List}(\texttt{Pcomp} \times \texttt{String} \times \texttt{Pcomp}) \\ \texttt{lab\_system} ::= \texttt{Read}(\texttt{String};\texttt{value}) \mid \texttt{Write}(\texttt{String};\texttt{value}) \\ \textbf{axioms} \\ \texttt{AXs}_{Ob} \\ \texttt{AXs}_{Ach} \\ \texttt{AXs}_{Sch} \\ \} \end{array}}$$

if if $\texttt{prcomp} \equiv T_1; \ldots; T_n$, $\texttt{Psort} \equiv \texttt{Psort} ::= \textbf{sort}\ T_1 \mid \ldots \mid T_n$

Components

The semantics of the component list adds to the environment the object and process identities (elements of the sort Ident*), returns the list of the specifications defining the components and the list of the types of the components that are in the system state.*

$$\frac{}{\begin{pmatrix} (\texttt{ob}, oid_1, T_1); \\ \ldots; \\ (\texttt{ob}, oid_m, T_m); \\ (\texttt{pr}, pid_1, T_{m+1}); \\ \ldots; \\ (\texttt{pr},\ pid_n, T_{m+n} \end{pmatrix}, \rho \vdash_{\overline{\text{Comps}}} \begin{array}{l} T'_1\texttt{\_SP}\ \ \textbf{and}\ \ldots\ \textbf{and}\ T'_m\texttt{\_SP}, \\ T_1; \ldots; T_m, \\ T_{m+1}\ ; \ldots; T_{m+n}, \\ \mathsf{AddComponents}((\texttt{ob}, oid_1,\ T_1,\ 1); \ldots ;, \\ (\texttt{pr},\ pid_n,\ T_{m+n}, \text{n}),\ \rho) \end{array}}$$

if $\text{DistinctNames}(T_1, \ldots, T_n) = T'_1; \ldots; T'_m$

Object Cooperations

$$\frac{objcoop_i, \rho \vdash_{\overline{\text{OCoop}}} \texttt{Ax}_i \ \ i = 1, \ldots, n}{objcoop_1; \ldots; objcoop_n, \rho \vdash_{\overline{\text{OCoops}}} \texttt{Ax}_1 \ldots \texttt{Ax}_n}$$

Object Cooperation

The semantics of an object cooperation is an axiom stating that if the process performs a method call to the object (using its local name for the object), the whole system performs an internal transition whose final state contains the new state of the process after the method call has returned and the new state of the object after the method execution.

$$\frac{}{(oid, pid, ln, mets), \rho \ \vdash_{\text{OCoop}} \mathtt{Ax}_1, \mathtt{Ax}_2}$$

if

$\mathtt{Ax}_1 \equiv$ **CCall1**

$\mathtt{ID(oc)} = \mathsf{IdentComp}(oid, \rho) \land \mathtt{ID(pc)} = \mathsf{IdentComp}(pid,\rho) \land$

$met(\mathtt{pl}, \mathtt{oc}) = (\mathtt{res},\mathtt{oc}') \land \mathtt{pc} \xrightarrow{\mathtt{call(res}\leftarrow\, ln.met\mathtt{(pl))}} \mathtt{pc}' \ \Rightarrow$

$(\mathtt{pc},\, \mathtt{res}{\leftarrow}ln.met(\mathtt{pl}),\, \mathtt{pc}'){:}\mathtt{pc} \mid \mathtt{oc} \mid \mathtt{PasC} \xrightarrow{\tau} \mathtt{pc}' \mid \mathtt{oc}' \mid \mathtt{PasC}$

$\mathtt{Ax}_2 \equiv$ **CCall2**

$\mathtt{ID(oc)} = \mathsf{IdentComp}(oid,\rho) \ \land \mathtt{ID(pc)} = \mathsf{IdentComp}(pid,\rho) \ \land$

$met(\mathtt{pl}, \mathtt{oc}) = \mathtt{oc}' \land \mathtt{pc} \xrightarrow{\mathtt{call(}ln.met\mathtt{(pl))}} \mathtt{pc}' \ \Rightarrow$

$(\mathtt{pc}, ln.met(\mathtt{pl}), \mathtt{pc}'){:}\mathtt{pc} \mid \mathtt{oc} \mid \mathtt{PasC} \xrightarrow{\tau} \mathtt{pc}' \mid \mathtt{oc}' \mid \mathtt{PasC}$

Asynchronous Cooperations

The semantics of the asynchronous cooperations is a part of the arity of the system state constructor and a list of axioms.

$$\frac{Acoop_i, \rho \vdash_{\text{ACoop}} stcomps_i, \mathtt{Ax}_i \quad i = 1, \ldots, n}{Acoop_1; \ldots; Acoop_n, \rho \vdash_{\text{ACoops}} stcomps_1; \ldots; stcomps_n, \mathtt{Ax}_1 \ldots \mathtt{Ax}_n}$$

Asynchronous simple cooperation with the outside

The semantics of an asynchronous cooperation with the outside does not return any system state component but only some axioms about the cooperation

$$\frac{(ach\_id,\, T, \mathtt{s}, (\mathtt{EXT},\, extid),\, ocomps) \vdash_{\text{SACoop}} \mathtt{Ax}}{(ach\_id,\, T, \mathtt{s}, (\mathtt{EXT},\, extid),\, ocomps) \vdash_{\text{ACoop}} \mathtt{Ax}}$$

$$\frac{(ach\_id,\, T, \mathtt{s},\, icomps, (\mathtt{EXT},\, extid)) \vdash_{\text{SACoop}} \mathtt{Ax}}{(ach\_id,\, T, \mathtt{s},\, icomps, (\mathtt{EXT},\, extid)) \vdash_{\text{ACoop}} \mathtt{Ax}}$$

Asynchronous simple cooperation

The semantics of an asynchronous cooperation among processes is the sort of the asynchronous channel plus some axioms

$$\frac{(ach\_id,\ T, \mathtt{s}, icomps, ocomps) \vdash_{\overline{\text{SACoop}}} \mathtt{Ax}}{(ach\_id,\ T, \mathtt{s}, icomps, ocomps) \vdash_{\overline{\text{ACoop}}} \mathtt{Achan}[T], \mathtt{Ax}} \quad \textbf{if} \quad icomps \neq (\mathtt{EXT},\ extid)$$

Asynchronous broadcast cooperation

The semantics of a broadcast asynchronous cooperation is the sort of the broadcast asynchronous channel plus some axioms

$$\frac{(ach\_id,\ T, \mathtt{b}, icomps, ocomps) \vdash_{\overline{\text{BACoop}}} \mathtt{Ax}}{(ach\_id,\ T, \mathtt{b}, icomps, ocomps) \vdash_{\overline{\text{ACoop}}} \mathtt{Bchan}[T], \mathtt{Ax}}$$

Asynchronous named cooperation

The semantics of an asynchronous cooperation is the sort of the named asynchronous channel plus some axioms

$$\frac{(ach\_id,\ T, \mathtt{n}, icomps, ocomps) \vdash_{\overline{\text{NACoop}}} \mathtt{Ax}}{(ach\_id,\ T, \mathtt{n}, icomps, ocomps) \vdash_{\overline{\text{ACoop}}} \mathtt{NAchan}[T], \mathtt{Ax}}$$

Asynchronous simple cooperation 1

The semantics of a simple cooperation with input from the external environment is a list of axioms stating that any process of the cooperation can read the data and consume it.

$$\frac{}{(ach\_id,\ T, \mathtt{s}, (\mathtt{EXT}, extid), (pid_1, ln_1); \ldots; (pid_m, ln_m)), \rho \vdash_{\overline{\text{SACoop}}} \mathtt{Ax}_1 \ldots \mathtt{Ax}_m}$$

if $\mathtt{Ax}_i \equiv$

CReadOut$_i$)

$$\mathtt{ID(pc)} = \mathsf{IdentComp}(pid_i,\ \rho)\ \wedge \mathtt{pc} \xrightarrow{?ln_i(\mathtt{x}_i)} \mathtt{pc´} \Rightarrow$$

$$(\mathtt{pc}, ?ln_i(\mathtt{x}_i), \mathtt{pc´})\mathtt{:pc} \mid \mathtt{PasC} \xrightarrow{\mathtt{Read}(extid, \mathtt{x}_i)} \mathtt{pc} \mid \mathtt{PasC}$$

CFailedReadOut$_i$)

$$\texttt{ID(pc)} = \mathsf{IdentComp}(pid_i,\, \rho) \;\wedge\; \texttt{pc} \xrightarrow{\texttt{No?}(ln_i)} \texttt{pc´} \;\Rightarrow$$
$$(\texttt{pc,No?}(ln_i)\texttt{,pc´})\texttt{: pc | PasC} \xrightarrow{\tau} \texttt{pc | PasC}$$

Asynchronous simple cooperation 2
The semantics of a cooperation with output to the external environment is a list of axioms stating that any process of the cooperation can write data outside.

$$\frac{}{(ach\_id,\, T, \texttt{s}, (pid_1, ln_1); \ldots; (pid_m, ln_m), (\texttt{EXT}, extid)) \vdash_{\text{ACoop}} \texttt{Ax}_1 \ldots \texttt{Ax}_m}$$

if $\texttt{Ax}_i \equiv$

CWriteOut$_i$**)**

$$\texttt{ID(pc)} = \mathsf{IdentComp}(pid_i,\, \rho) \;\wedge\; \texttt{pc} \xrightarrow{!ln_i(\texttt{x}_i)} \texttt{pc´} \;\Rightarrow$$
$$(\texttt{pc,}!\,ln(\texttt{x}_i)\texttt{,pc´})\texttt{:pc | PasC} \xrightarrow{\texttt{Write}(extid,\texttt{x}_i)} \texttt{pc´ | PasC}$$

Asynchronous simple cooperation 3
The semantics of a simple cooperation among process returns some axioms. A reader process can read a value only if there is some data available on the channel; a writer process can always write data on the channel. For every read/write operation of a process, there is an internal transition of the system.

$$\frac{}{\begin{pmatrix} ach\_id,\ \texttt{s},\ T, \\ (pid_1{}^O,\ ln_1{}^O);\ \ldots\ ;\ (pid_m{}^O, ln_m{}^O), \\ (pid_1{}^I, ln_1{}^I);\ \ldots\ ;\ (pid_n{}^I, ln_n{}^I) \end{pmatrix}, \rho \vdash_{\text{SACoop}} \begin{matrix} \texttt{Ax}_1{}^I \ldots \texttt{Ax}_n{}^I \\ \texttt{Ax}_1{}^O \ldots \texttt{Ax}_m{}^O \end{matrix}}$$

if for $i = 1, \ldots m$

$\texttt{Ax}_i{}^O \equiv$

CWriteIn)

$$\texttt{ID(ach)=}\ ach\_id \wedge \texttt{Write(ach,x}_i\texttt{)} = \texttt{ach´} \wedge \texttt{ID(pc)} = \mathsf{IdentComp}(pid_i^O,\, \rho) \;\wedge$$
$$\texttt{pc} \xrightarrow{!ln_i{}^O(\texttt{x}_i)} \texttt{pc´} \;\Rightarrow$$
$$(\texttt{pc,}!\,ln_i{}^O(\texttt{x}_i)\texttt{,pc´})\texttt{: pc | ach | PasC} \xrightarrow{\tau} \texttt{pc´ | ach´ | PasC;}$$

for $i = 1, \ldots n$,
$\texttt{Ax}_i{}^I \equiv$

CReadIn)

$\texttt{ID(ach)} = \mathit{ach\_id} \wedge \texttt{Read(ach)} = (\texttt{ach}',\texttt{x}_i) \ \wedge$

$\texttt{ID(pc)} = \mathsf{IdentComp}(\mathit{pid}_i{}^I,\rho) \ \wedge \texttt{pc} \xrightarrow{?\mathit{ln}_i{}^I(\texttt{x}_i)} \texttt{pc}' \ \Rightarrow$

$(\texttt{pc},?\mathit{ln}_i{}^I(\texttt{x}_i),\texttt{pc}')\text{:}\ \texttt{pc} \mid \texttt{ach} \mid \texttt{PasC} \xrightarrow{\tau} \texttt{pc}' \mid \texttt{ach}' \mid \texttt{PasC}$

CFailedReadIn)

$\texttt{ID(ach)} = \mathit{ach\_id} \wedge \texttt{Unavailable(ach)} \ \wedge \texttt{ID(pc)} = \mathsf{IdentComp}(\mathit{pid}_i{}^I,\rho) \ \wedge$

$\texttt{pc} \xrightarrow{\texttt{No?}(\mathit{ln}_i{}^O)} \texttt{pc}' \ \Rightarrow$

$(\texttt{pc},\texttt{No?}(\mathit{ln}_i{}^O),\texttt{pc}')\text{:}\ \texttt{pc} \mid \texttt{ach} \mid \texttt{PasC} \xrightarrow{\tau} \texttt{pc}' \mid \texttt{ach} \mid \texttt{PasC;}$

Asynchronous broadcast cooperation

The semantics of a broadcast cooperation returns some axioms. The first axiom requires that the channel is used for reading only by the processes described as readers in the cooperation. For each process using the broadcast channel, there is a separate storage space where it can read from /write to. A reader process can read a value only if there is some data available on the storage space corresponding to the process; a writer process can always write data on the channel. These data are replicated on the storage space for every reader process. For every read/write operation of one process, there is an internal transition of the system.

$$\begin{pmatrix} \mathit{ach\_id},\ \texttt{b},\ T, \\ (\mathit{pid}_1{}^O,\ \mathit{ln}_1{}^O);\ \ldots\ ;\ (\mathit{pid}_m{}^O,\mathit{ln}_m{}^O), \\ (\mathit{pid}_1{}^I,\mathit{ln}_1{}^I);\ \ldots\ ;\ (\mathit{pid}_n{}^I,\mathit{ln}_n{}^I) \end{pmatrix}, \rho \vdash_{\text{BACoop}} \begin{matrix} \texttt{Ax}_1{}^I \ldots \texttt{Ax}_n{}^I \\ \texttt{Ax}_1{}^O \ldots \texttt{Ax}_m{}^O \end{matrix}$$

if for $i = 1, \ldots m$

$\texttt{Ax}_i{}^O \ \equiv$

CWriteBr)

$\texttt{ID(ach)} = \mathit{ach\_id} \wedge \texttt{Oid}_i = \mathsf{IdentComp}(\mathit{pid}_i{}^O,\ \rho) \ \wedge$

$\texttt{WriteBr(ach,x}_i) = \texttt{ach}' \wedge \texttt{ID(pc)} = \texttt{Oid}_i \wedge$

$\texttt{pc} \xrightarrow{!\mathit{ln}_i{}^O(\texttt{x}_i)} \texttt{pc}' \ \Rightarrow$

$(\texttt{pc},!\mathit{ln}_i{}^O(\texttt{x}_i),\texttt{pc}')\text{:}\ \texttt{pc} \mid \texttt{ach} \mid \texttt{PasC} \xrightarrow{\tau} \texttt{pc}' \mid \texttt{ach}' \mid \texttt{PasC;}$

for $i = 1, \ldots n$

$\texttt{Ax}_i{}^I \ \equiv$

CReadBr

$\mathtt{ID(ach)}= \mathit{ach\_id} \wedge \mathtt{ReadBr(ach,}\mathsf{IdentComp}(\mathit{pid_i}^I,\rho)) = (\mathtt{x}_i,\mathtt{ach}') \ \wedge$

$\mathtt{ID(pc)} = \mathsf{IdentComp}(\mathit{pid_i}^I,\rho) \ \wedge \mathtt{pc} \xrightarrow{?ln_i^I(\mathtt{x}_i)} \mathtt{pc}' \ \Rightarrow$

$(\mathtt{pc},?ln_i^I(\mathtt{x}_i),\mathtt{pc}')\colon \mathtt{pc} \mid \mathtt{ach} \mid \mathtt{PasC} \xrightarrow{\tau} \mathtt{pc}' \mid \mathtt{ach}' \mid \mathtt{PasC}$

CFailedReadBr

$\mathtt{ID(ach)}= \mathit{ach\_id} \wedge \mathtt{Unavailable(ach,}\ \mathit{pid_i}^I) \ \wedge \mathtt{ID(pc)} = \mathsf{IdentComp}(\mathit{pid_i}^I,\rho) \ \wedge$

$\mathtt{pc} \xrightarrow{\mathtt{No}?(ln_i^I)} \mathtt{pc}' \ \Rightarrow$

$(\mathtt{pc},\mathtt{No}?(ln_i^I),\mathtt{pc}')\colon \mathtt{pc} \mid \mathtt{ach} \mid \mathtt{PasC} \xrightarrow{\tau} \mathtt{pc}' \mid \mathtt{ach} \mid \mathtt{PasC};$

Asynchronous broadcast external cooperation

The semantics of a broadcast cooperation with outside is a list of axioms stating that any process taking part in the cooperation can read the data and consume it.

$$\frac{}{(\mathit{ach\_id},\ T, \mathtt{b}, (\mathtt{EXT},\ \mathit{extid}), (\mathit{pid}_1, \mathit{ln}_1); \ldots; (\mathit{pid}_m, \mathit{ln}_m)), \rho \vdash_{\text{ACoop}} \mathtt{Ax}_1 \ldots \mathtt{Ax}_m}$$

if $\mathtt{Ax}_i \equiv$

CReadOut$_i$)

$\mathtt{ID(pc)} = \mathsf{IdentComp}(\mathit{pid}_i,\ \rho) \ \wedge \mathtt{pc} \xrightarrow{?ln_i(\mathtt{x}_i)} \mathtt{pc}' \Rightarrow$

$(\mathtt{pc},?ln_i(\mathtt{x}_i),\mathtt{pc}')\colon\mathtt{pc} \mid \mathtt{PasC} \xrightarrow{\mathtt{Read}(\mathit{extid},\mathtt{x}_i)} \mathtt{pc}' \mid \mathtt{PasC}$

CFailedReadOut$_i$)

$\mathtt{ID(pc)} = \mathsf{IdentComp}(\mathit{pid}_i,\ \rho) \ \wedge \mathtt{pc} \xrightarrow{\mathtt{No}?(ln_i)} \mathtt{pc}' \ \Rightarrow$

$(\mathtt{pc},\mathtt{No}?(ln_i),\mathtt{pc}')\colon \mathtt{pc} \mid \mathtt{PasC} \xrightarrow{\tau} \mathtt{pc}' \mid \mathtt{PasC}$

Asynchronous named cooperation

The semantics of a cooperation for named communications returns some axioms. A process can write a message on the channel specifying the receiver, that is the identity of the process the message is sent to. A process can read a message in the channel that has been sent to it. Every read/write operation is an internal transition of the system. Let us recall that a named communication cannot involve external entities.

$$\frac{}{\begin{pmatrix} \mathit{ach\_id},\ T,\ \mathtt{n}, \\ (\mathit{pid}_1{}^O,\ \mathit{ln}_1{}^O);\ \ldots\ ;\ (\mathit{pid}_m{}^O,\mathit{ln}_m{}^O), \\ (\mathit{pid}_1{}^I,\mathit{ln}_1{}^I);\ \ldots\ ;\ (\mathit{pid}_n{}^I,\mathit{ln}_n{}^I) \end{pmatrix}, \rho \vdash_{\text{NACoop}} \begin{matrix} \mathtt{Ax}_1{}^I \ \ldots \mathtt{Ax}_n{}^I \\ \mathtt{Ax}_1{}^O \ \ldots \mathtt{Ax}_m{}^O \end{matrix}}$$

if

for $i = 1, \ldots m$,
$\mathtt{Ax}_i{}^O \equiv$

CWriteN

$\mathtt{ID(ach)}= \mathit{ach\_id} \wedge \mathtt{Oid}_i = \mathsf{IdentComp}(\mathit{pid}_i^O, \rho) \wedge \mathtt{ach}' = \mathtt{WriteN}(\mathtt{ach}, \mathtt{x}_i, \mathtt{Oid}_i, x)$
$\wedge \ \wedge \mathtt{ID(pc)} = \mathtt{Oid}_i \wedge$
$\mathtt{pc} \xrightarrow{x!\mathit{ln}_i{}^I(\mathtt{x}_i)} \mathtt{pc}' \Rightarrow$
$(\mathtt{pc}, x!\mathit{ln}_i{}^I(\mathtt{x}_i), \mathtt{pc}')\colon \mathtt{pc} \mid \mathtt{ach} \mid \mathtt{PasC} \xrightarrow{\tau} \mathtt{pc}' \mid \mathtt{ach}' \mid \mathtt{PasC};$

for $i = 1, \ldots, n$ $\mathtt{Ax}_i{}^I \equiv$

CReadN)

$\mathtt{ID(ach)}= \mathit{ach\_id} \wedge \mathtt{Iid}_i = \mathsf{IdentComp}(\mathit{pid}_i{}^I, \rho) \wedge \mathtt{ReadN}(\mathtt{ach}, \mathtt{Iid}_i) = ((\mathtt{y}, \mathtt{x}_i), \mathtt{ach}')$
$\wedge \mathtt{ID(pc)} = \mathtt{Iid}_i \wedge \mathtt{pc} \xrightarrow{\mathtt{y}?\mathit{ln}_i{}^I(\mathtt{x}_i)} \mathtt{pc}' \Rightarrow$
$(\mathtt{pc}, \mathtt{y}?\mathit{ln}_i{}^I(\mathtt{x}_i), \mathtt{pc}')\colon \mathtt{pc} \mid \mathtt{ach} \mid \mathtt{PasC} \xrightarrow{\tau} \mathtt{pc}' \mid \mathtt{ach}' \mid \mathtt{PasC};$

CFailedReadN)

$\mathtt{ID(ach)} = \mathit{ach\_id} \wedge \mathtt{Unavailable}(\mathtt{ach}, \mathtt{Iid}_i) \wedge$
$\mathtt{Iid}_i = \mathsf{IdentComp}(\mathit{pid}_i{}^I, \rho) \wedge \mathtt{ID(pc)} = \mathtt{Iid}_i \wedge$
$\mathtt{pc} \xrightarrow{\mathtt{No}?(\mathit{ln}_i{}^I)} \mathtt{pc}' \Rightarrow (\mathtt{pc}, \mathtt{No}?(\mathit{ln}_i{}^I), \mathtt{pc}')\colon \mathtt{pc} \mid \mathtt{ach} \mid \mathtt{PasC} \xrightarrow{\tau} \mathtt{pc}' \mid \mathtt{ach} \mid \mathtt{PasC};$

Synchronous Cooperations

The semantics of the synchronous cooperations is a part of the arity of the system state constructor and a list of axioms.

$$\frac{\mathit{scoop}_i, \rho \vdash_{\mathrm{SCoop}} \mathit{stcomps}_i, \mathtt{Ax}_i \quad i = 1, \ldots, n}{\mathit{scoop}_1; \ldots; \mathit{scoop}_n, \rho \vdash_{\mathrm{SCoops}} \mathit{stcomps}_1 \times \cdots \times \mathit{stcomps}_n, \mathtt{Ax}_1 \ldots \mathtt{Ax}_n}$$

The semantics of a synchronous cooperation does not distinguish the channels for synchronous cooperation that are shared among two process and the channels shared by n readers and m writers. The axioms ensure that a reader cannot try to synchronize with a writer if there already exists another reader waiting for the synchronization completion.

Synchronous Cooperation with signal exchange

The semantics of a synchronous cooperation returns a sort (of the element describing the synchronous connector) plus some axioms. A reader (writer) process can read (write) and complete the synchronization only if there is some other writer (reader) process already suspended on the channel; the suspended process is resumed and both process completes the communication. Otherwise, the process is suspended until a writer (reader) tries to complete the synchronization. For every read/write operation of one process, there is an internal transition of the system.

$$\begin{pmatrix} sch\_id, \texttt{void}, \\ (pid_1{}^O,\ ln_1{}^O);\ \ldots\ ;\ (pid_m{}^O, ln_m{}^O), \\ (pid_1{}^I, ln_1{}^I);\ \ldots\ ;\ (pid_n{}^I, ln_n{}^I \end{pmatrix}, \rho \vdash_{\overline{\text{SCoop}}} SigSyChan \begin{array}{l} \texttt{Ax}_1{}^I \ldots \texttt{Ax}_n{}^I \\ \texttt{Ax}_1{}^O \ldots \texttt{Ax}_m{}^O \end{array}$$

if for $i = 1, \ldots m, j = 1, \ldots, n$

$\texttt{Ax}_j{}^I \equiv$

CSynchRead

$\texttt{ID(sch)} = sch\_id \wedge \texttt{OKsynch(sch)} \wedge \texttt{ID(pc)} = \mathsf{IdentComp}(pid_j{}^I, \rho)$

$\wedge\ \texttt{ID(pc}_S) = \texttt{W}(sch\_id) \wedge \texttt{pc} \xrightarrow{?ln()} \texttt{pc}'$

$\wedge\ \texttt{pc}_S \xrightarrow{\texttt{resumed}(ln_j)} \texttt{pc}'_S \Rightarrow$

$((\texttt{pc}, ?ln(), \texttt{pc}'), (\texttt{pc}_S, \texttt{resumed}(ln_j), \texttt{pc}'_S))$:

$\texttt{pc} \mid \texttt{pc}_S \mid \texttt{sch} \mid \texttt{PasC} \xrightarrow{\tau} \texttt{pc}' \mid \texttt{pc}'_S \mid \texttt{Reset(sch)} | \texttt{PasC};$

(CSynchBlockedRead)

$\texttt{ID(sch)} = sch\_id \wedge \neg\ \texttt{OKsynch(sch)} \wedge$

$\texttt{ID(pc)} = \mathsf{IdentComp}(pid_j{}^I, \rho) \wedge$

$\texttt{pc} \xrightarrow{??ln} \mathsf{Susp}(ln, \texttt{pc}') \Rightarrow$

$(\texttt{pc}, ??ln, \texttt{pc}')$: $\texttt{pc} \mid \texttt{sch} \mid \texttt{PasC} \xrightarrow{\tau} \texttt{pc}' \mid \texttt{Register(sch, ID(pc))} \mid \texttt{PasC};$

$\texttt{Ax}_i{}^O \equiv$

(CSynchWrite)

$\texttt{ID(sch)} = sch\_id \wedge \texttt{OKsynch(sch)} \wedge \texttt{ID(pc)} = \mathsf{IdentComp}(pid_j{}^O, \rho) \wedge$

$\texttt{ID(pc}_S) = \texttt{W}(sch\_id) \wedge \texttt{pc} \xrightarrow{!ln()} \texttt{pc}'$

$\wedge\ \texttt{pc}_S \xrightarrow{\texttt{resumed}(ln_i)} \texttt{pc}'_S \Rightarrow$

$((\texttt{pc}, !\,ln(), \texttt{pc}'), (\texttt{pc}_S, \texttt{resumed}(ln_i), \texttt{pc}'_S))$:

$\texttt{pc} \mid \texttt{pc}_S \mid \texttt{sch} \mid \texttt{PasC} \xrightarrow{\tau} \texttt{pc}' \mid \texttt{pc}'_S \mid \texttt{Reset(sch)} | \texttt{PasC};$

CSynchBlockedWrite

$\mathtt{ID(sch)} = sch\_id \wedge \neg\ \mathtt{OKsynch(sch)} \wedge$

$\mathtt{ID(pc)} = \mathsf{IdentComp}(pid_j{}^O)\ \mathtt{pc} \xrightarrow{!!ln()} \mathsf{Susp}(ln, \mathsf{pc}') \Rightarrow$

$(\mathtt{pc}, !!ln(), \mathtt{pc}')$: $\mathtt{pc} \mid \mathtt{sch} \mid \mathtt{PasC} \xrightarrow{\tau} \mathtt{pc}' \mid \mathtt{Register}(\mathtt{sch}, \mathtt{ID(pc)}) \mid \mathtt{PasC};$

$$\begin{pmatrix} sch\_id,\ T, \\ (pid_1{}^O,\ ln_1{}^O);\ \ldots\ ;\ (pid_m{}^O, ln_m{}^O), \\ (pid_1{}^I, ln_1{}^I);\ \ldots\ ;\ (pid_n{}^I, ln_n{}^I \end{pmatrix}, \rho \vdash_{\overline{\text{SCoop}}} SyChan[T]\ \begin{matrix} \mathtt{Ax}_1{}^I \ldots \mathtt{Ax}_n{}^I \\ \mathtt{Ax}_1{}^O \ldots \mathtt{Ax}_m{}^O \end{matrix}$$

if $T \neq$ `void` and for $i = 1, \ldots m, j = 1, \ldots, n$

$\mathtt{Ax}_j{}^I \equiv$

CSynchRead

$\mathtt{ID(sch)} = sch\_id \wedge \mathtt{OKsynch(sch)} = \mathtt{OUT}(pid, msg) \wedge \mathtt{ID(pc)} = \mathsf{IdentComp}(pid_j{}^I, \rho)$

$\wedge\ \mathtt{ID}(\mathtt{pc}_S) = pid \wedge \mathtt{pc} \xrightarrow{?ln(msg)} \mathsf{pc}' \wedge \mathtt{pc}_S \xrightarrow{\mathtt{resumed}(ln_j)} \mathtt{pc}'_S \Rightarrow$

$((\mathtt{pc}, ?ln(msg), \mathtt{pc}'), (\mathtt{pc}_S, \mathtt{resumed}(ln_j), \mathtt{pc}'_S))$:

$\mathtt{pc} \mid \mathtt{pc}_S \mid \mathtt{sch} \mid \mathtt{PasC} \xrightarrow{\tau} \mathtt{pc}' \mid \mathtt{pc}'_S \mid \mathtt{Reset(sch)} \mid \mathtt{PasC};$

(CSynchBlockedRead)

$\mathtt{ID(sch)} = sch\_id \wedge \mathtt{OKsynch(sch)} = \mathtt{nomess} \wedge$

$\mathtt{ID(pc)} = pid \wedge \mathtt{pc} \xrightarrow{??ln()} \mathsf{Susp}(ln, \mathsf{pc}') \Rightarrow$

$(\mathtt{pc}, ??ln(), \mathtt{pc}')$: $\mathtt{pc} \mid \mathtt{sch} \mid \mathtt{PasC} \xrightarrow{\tau} \mathtt{pc}' \mid \mathtt{Register}(\mathtt{sch}, \mathtt{IN}(pid)) \mid \mathtt{PasC};$

$\mathtt{Ax}_i{}^O \equiv$

(CSynchWrite)

$\mathtt{ID(sch)} = sch\_id \wedge \mathtt{OKsynch(sch)} = \mathtt{IN}(pid) \wedge \mathtt{ID(pc)} = \mathsf{IdentComp}(pid_j{}^O, \rho) \wedge$

$\mathtt{ID}(\mathtt{pc}_S) = pid \wedge \mathtt{pc} \xrightarrow{!ln(msg)} \mathsf{pc}' \wedge \mathtt{pc}_S \xrightarrow{\mathtt{resumed}(ln_i)} \mathtt{pc}'_S \Rightarrow$

$((\mathtt{pc}, !ln(msg), \mathtt{pc}'), (\mathtt{pc}_S, \mathtt{resumed}(ln_i), \mathtt{pc}'_S))$:

$\mathtt{pc} \mid \mathtt{pc}_S \mid \mathtt{sch;PasC} \xrightarrow{\tau} \mathtt{pc}' \mid \mathtt{pc}'_S \mid \mathtt{Reset(sch)} \mid \mathtt{PasC};$

CSynchBlockedWrite

$\mathtt{ID(sch)} = sch\_id \wedge \mathtt{OKsynch(sch)} = \mathtt{nomess} \wedge$

$\mathtt{ID(pc)} = \mathsf{IdentComp}(pid_i{}^O, \rho) \wedge \mathtt{pc} \xrightarrow{!!ln()} \mathsf{Susp}(ln, \mathsf{pc}') \Rightarrow$

$(\mathtt{pc}, !!ln(), \mathtt{pc}')$: $\mathtt{pc} \mid \mathtt{sch;PasC} \xrightarrow{\tau} \mathtt{pc}' \mid \mathtt{Register}(\mathtt{sch}, \mathtt{OUT}(\mathtt{ID(pc)}, msg)]) \mid \mathtt{PasC};$

4.5 Specification of parametric asynchronous channels

The connectors for asynchronous communication are defined by a parametric specification with a sort for the connector instances. The constructor for the connector takes the connector identity and the list of the contained values. The operation for reading from the connector returns the first elements in the value list and the new connector state, where the read element is deleted. The operation for writing an element simply adds it to the value list. There is a predicate `Unavailable` to check if there is some element to be read in the connector.

We assume to have the CASL specification of the parametric lists, whose sort is List[T], whereas *Top*, *Pop*and *Push*and `IsEmpty` are the usual operations.

spec ACHAN[T] =
free{ `Ident_SP` **and** List[T]
then
type `Achan`[T] ::= [_,_]: (`ID:Ident`;List(T))
type T `X` `Achan`[T] ::= (_,_)(T; `Achan`[T])
ops `Read`: `Achan`[T] $\rightarrow$ $T\times$`Achan`[T];
`Write`: `Achan`[T] $\times T \rightarrow$ `Achan`[T];
pred `Unavailable`: `Achan`[T]
axioms
`Read`([*id*, *val_L*]) = (*Top*(*val_L*), *Pop*(*val_L*))
`Write`([*id*, *val_L*], `t`) = (*id*, *Push*(`t`,*val_L*))
`Unavailable`([*id*, *val_L*]) $\Leftrightarrow$ `IsEmpty`(*val_L*)
}

The connectors for asynchronous broadcast communication are defined by a parametric specification with a sort for the connector instances, The constructor for the connectors takes the connector identity and the contained values is a list of couples whose first element is the set of identities of the process that have to read the message described by the second element of the couple.

The operation for reading from the connector takes the reader identity and remove it from the set of the process that have to read the first element in the content list; if all the process have read the message, the read element is deleted.

The operation for writing an element adds the couple with the written value plus the list of the processes that will read the broadcasted message.

There is a predicate `Unavailable` to check if there is some element to read in the connector.

We assume to have the CASL specification of the parametric sets, whose sort is Set[T], with the usual operations.

spec ACHANBR[T] =
free{ `Ident_SP` **and** List[T] **and** `Set`(T)
then
type T `X Set[Ident]` ::= (_,_)(T; `Set[Ident]`);
type `Bchan`[T] ::= [_,_](`ID:Ident`;Cont: List[$T\times$ Set(`Ident`)]);
type T `X Bchan`[T] ::= (_,_)(T;`Bchan`[T]);
ops `ReadBr:Bchan`[T] $\times$ `Ident`$\rightarrow$ $T\times$`Bchan`(T);
`WriteBr`: `Bchan`[T] $\times$ $T\times$ List(`Ident`)$\rightarrow$ `Bchan`(T);
pred `AvailableBr`: `Bchan`(T) $\times$ `Ident`;
vars pidS, pidS$'$: `Set(Ident)`; id,pid:`Ident`;
ll: List[$T\times$ Set(`Ident`)] ; $\mathtt{x}_i$: T;
axioms
$pid \in pid\mathrm{S} \wedge pid\mathrm{S}' = pid\mathrm{S}\text{-} \{pid\} \wedge pid\mathrm{S}' \neq \{\} \Rightarrow$
$\mathtt{ReadBr}([id,(\mathtt{x}_i,\ pid\mathrm{S})::ll],\ pid) = (\mathtt{x}_i,(\mathtt{x}_i,pid\mathrm{S}')::ll))$;
$\mathtt{ReadBr}([id,(\mathtt{x}_i,\{pid\})::ll]),\ pid) = (\mathtt{x}_i,\ [id,ll])$;
$pid \notin pid\mathrm{S} \Rightarrow \mathtt{ReadBr}([id,(\mathtt{x}_i,\ pid\mathrm{S})::ll]),\ pid) = \mathtt{ReadBr}([id,ll],\ pid)$;
$\mathtt{WriteBr}([id,\ ll],\ \mathtt{x}_i,\ pid\mathrm{S}) = (id,\ Append((\mathtt{x}_i,pid\mathrm{S}),ll))$;
$\neg\ \mathtt{IsEmpty}(ll) \Rightarrow \mathtt{AvailableBr}([id,(\mathtt{x}_i,\ pid\_\mathrm{L})::ll]),\ pid) \Leftrightarrow$
$pid \in pid\_\mathrm{L} \vee \mathtt{Available}([id,(\mathtt{x}_i,ll)],pid)$;
}

The connectors for asynchronous named communication are defined by a parametric specification with a sort for the connector instances, The constructor for the connector takes the connector identity and the contained values, that is a list of triples whose first element is the message, the second and the third are the sender and the receiver identities.

The operation for reading from the connector takes the reader identity and returns the first element in the content list with the same receiver identity.

The operation for writing an element adds the triple with the written value plus the identities of the sender and of the receiver.

spec NCHAN[T] =
free{
`Ident_SP`
then
type `NAchan`[T] ::= [_,_](`ID:Ident`;List(TX `Ident`X `Ident`));
type `Ident X` T `X NAchan`(T) ::= (_,_, _)(`Ident`; T; `NAchan`(T));
ops `ReadN`: `NAchan`[T] $\times$ `Ident`$\rightarrow$ `Ident`X TX`NAchan`(T);
`WriteN`: `NAchan`[T] $\times T \times$ `Ident`$\times$ `Ident`$\rightarrow$ `NAchan`(T);
vars *snd*,*snd'*,*rec*,*id*,*pid*:`Ident`;*ll*,*ll'*: List[TX `Ident`X `Ident`)]
$\mathtt{x}_i$, $\mathtt{x}_i$': T;
axioms `ReadN`([*id*,($\mathtt{x}_i$,*snd*,*rec*) ::= *ll*]), *rec*) = (*snd*,$\mathtt{x}_i$, [*id*,*ll*]);
pid$\neq$ *rec* $\wedge$ `ReadN`([*id*,*ll*], *pid*)= (*snd'*, $\mathtt{x}_i$',*ll'*) $\Rightarrow$
`ReadN`([*id*,($\mathtt{x}_i$, *snd*, *rec*) ::= *ll*]), *pid*) = (*snd'* $\mathtt{x}_i$', [*id*,*ll'*]);
`WriteN`([*id*, *ll*], $\mathtt{x}_i$, *snd*,*rec*) = (*id*, *Append*(($\mathtt{x}_i$,*snd*, *rec*),*ll*));
}

4.6 Specification of parametric synchronous channels

4.6.0.0.3 Synchronous Signal communication The connector for a synchronous signal communication corresponds to a specification with a sort for the connector instances,

The connectors for asynchronous communication is defined by a specification with a sort for the connector instance. The constructor for the connector takes the connector identity and the identity (possibly null) of the waiting process. An operation (`OKsynch`) checks if there is a waiting process, two operations (`Register`and `Reset`) are used, respectively, to register the identity of a new waiting process and to reset such value.

spec SIGSYCHAN
free{ `Ident_SP`
then
type `ExtId`::= **sort** `Ident` | `none`
type SigSyChan ::= [_,_]: (`ID:Ident`, `W:ExtId`) ;
pred `OKsynch`: SigSyChan;
ops `Register`: SigSyChan$\times$ `Ident`$\leftarrow$ SigSyChan;
`Reset`: SigSyChan$\leftarrow$ SigSyChan;
vars *id*, *id'*: `Ident`;*eid*: `ExtId`;
axioms
`OKsynch`([*id*, *eid*]) $\equiv$ *eid*$\equiv$ `none`;
`Register`([*id*, *eid*], *id'*) =([*id*, *id'*]) ;
`Reset`([*id*, *eid*]) = [*id*, `none`] ; }

4.6.0.0.4 Synchronous Message communication The connector for a synchronous message communication is defined by a specification for the message descriptors and a specifications for the connector itself.

The message constructor takes the identity of the process that communicates, an info that tells if it is a reader or a writer process, and the third component containing either the null value (if the suspended process is a reader), or the message itself (if the suspended process is a writer).

The constructor for the connectors takes the connector identity and a message.

The operation `OKsynch` and `Register` are used to complete the synchronization (if there is an already suspended process), or to be suspended on the channel.

spec `Msg_SP`$[T]$ =
free{
type `MsgS`$[T]$ ::= `OUT`(_,_): (`Ident`; T) |
`IN`(_,_): (`Ident`) |
`nomess`;
}

spec SYCHAN$[T]$ =
free{ `Ident_SP` **and** `MsgS`$[T]$
then
type SyChan$[T]$ ::= [_,_](`ID:Ident`, `MsgS`$[T]$;
ops `OKsynch`: SyChan$[T]$ $\rightarrow$ `MsgS`$[T]$;
`Register`: SyChan$[T]$ $\times$ `MsgS`$[T]$ $\leftarrow$ SyChan$[T]$;
`Reset`: SyChan$[T]$$\leftarrow$ SyChan$[T]$;
vars *msg*: `MsgS`$[T]$; *id*: `Ident`;
axioms
`OKsynch`([*id*, *msg*]) = *msg*;
`Register`([*id*, `nomess`]),*msg*) =([*id*, *msg*]) ;
`Reset`([*id*, *msg*])= [*id*, `nomess`] }

4.7 Sequence Diagram

The semantics of a sequence diagram is an axiom in the system specification that expresses a property over the *glts* we are describing, that is a property over the behaviour of the system described by the `JTN` specification.

We express such axioms using the branching time temporal logic described in the appendix (D).

The general form of the axioms is: $\Phi_{Init}(st) \Rightarrow \mathit{in\_one\_case}(st, \pi)$

where:

- $\Phi_{Init}(st)$ describes the initial state of the system components, as described in the top line of the sequence diagram;
- $\mathit{in\_one\_case}(st,\pi)$ requires that the property π has to be ensured by at least one execution of the system, starting from state st. The path formula π describes the occurrences of some cooperations along a (part of a) system execution, that is it describes that a path must contain the occurrences (in a given order) of the labels describing such cooperations.

Sequence Diagrams

The semantics of the sequence diagrams is a specification, extending the system specification, with an axiom for each sequence diagram.

$$\frac{\mathit{seqdia}_i, \rho \vdash_{\text{Seq}} \mathtt{Ax}_i \quad i = 1, \ldots, n}{\mathit{seqdia}_1; \ldots; \mathit{seqdia}_n, \rho \vdash_{\text{Seqs}} \begin{array}{l} \textbf{spec } \text{CONSTRAINED\_SYSTEM} = \\ \text{SYSTEM} \\ \textbf{then} \\ \mathtt{Ax}_1 \\ \ldots \\ \mathtt{Ax}_n \end{array}}$$

Sequence Diagram

The semantics of the initial states of the components is a formula over the state of the whole system; the semantics of the list of the interactions and of the annotations is a path-formula over at least one path starting from the initial state.

$$\frac{\begin{array}{l} \mathit{inits}, \rho \vdash_{\text{Inits}} \Phi_{state} \\ \mathit{interacts}, \rho \vdash_{\text{Inters}} \pi_1 \\ \mathit{anns} \vdash_{\text{Annots}} \pi_2 \end{array}}{(\mathit{vardecs}, \mathit{inits}, \mathit{coops}, \mathit{anns}), \rho \vdash_{\text{Seq}} \Phi_{state} \Rightarrow \mathit{in\_one\_case}(\mathtt{\_st}, \pi_1 \wedge \pi_2)}$$

Initial State

$$\frac{\mathit{id}_i, T_i, \mathit{exp}_i \vdash_{\text{Init}} \mathtt{Ax}_i \quad i = 1, \ldots, n}{(\mathit{id}_1, T_1, \mathit{exp}_1), \ldots, (\mathit{id}_n, T_n, \mathit{exp}_n) \vdash_{\text{Inits}} \mathtt{Ax}}$$

$$\text{if } \mathtt{Ax} \equiv \begin{array}{l} \mathtt{\_st}{=}(comp_1 \mid \ldots \mid comp_m) \ \wedge \\ (\exists\ comp.\ comp{\in}\{comp_1 \ldots comp_m\} \ \wedge \mathtt{ID}(comp) \ = \mathsf{IdentComp}(id_1,\ \rho) \ \Rightarrow \mathtt{Ax}_1) \wedge \\ \ldots \\ \wedge\ (\exists\ comp.\ comp{\in}\{comp_1 \ldots comp_m\} \ \wedge \mathtt{ID}(comp) \ = \mathsf{IdentComp}(id_n,\ \rho) \ \Rightarrow \mathtt{Ax}_n) \end{array}$$

$$\frac{exp \vdash_{\overline{\text{Exp}}} \mathtt{exp}}{exp \vdash_{\overline{\text{Init}}} comp = \mathtt{exp}}$$

$$\frac{exp_i \vdash_{\overline{\text{Exp}}} \mathtt{exp}_i \ i = 1, \ldots, n}{f_1 = exp_1; \ldots; f_n = exp_n \vdash_{\overline{\text{Init}}} f_1(comp) = \mathtt{exp}_1 \wedge \ \ldots \ \wedge f_n(comp) = \mathtt{exp}_n}$$

Cooperation list

$$\frac{inter_i, \rho \vdash_{\overline{\text{PSubF}}} \pi_i \quad i = 1, \ldots, n}{inter_1; \ldots; inter_n, \rho \vdash_{\overline{\text{Inters}}} \pi_1 \wedge (eventually\ \pi_2 \wedge (\ldots \ \wedge eventually\ \pi_n) \ldots)}$$

The following rules returns the path formulae describing which must be the information content in a point of the system behaviour. Every path formula is the formalization of an interaction of the sequence diagram, thus it requires that one of the following activities is described in the information:

- a component perform a transition modeling an asynchronous read/write operation;
- a component perform a transition modeling a method call;
- two components performs the transitions modeling a synchronization;

Interaction 1

$$\frac{}{\begin{array}{l}(\mathtt{EXT},\ ach\_id(\mathtt{term}),\ pid),\rho \vdash_{\overline{\text{PSubF}}} info\_cond(\mathrm{x} \ \bullet \mathrm{x} \ = \ (\mathtt{pc}, \\ ?\ ach\_id(\mathtt{term}), \mathtt{pc}') \wedge \\ \mathtt{ID}(\mathtt{pc}) = \mathsf{IdentComp}(pid,\rho))\end{array}}$$

Interaction 2

$$\frac{}{\begin{array}{l}(pid,\ ach\_id(\mathtt{term}),\ \mathtt{EXT}),\rho \vdash_{\overline{\text{PSubF}}} info\_cond(\mathrm{x} \ \bullet \mathrm{x} \ = \ (\mathtt{pc}, \\ !\ ach\_id(\mathtt{term}), \mathtt{pc}) \wedge \\ \mathtt{ID}(\mathtt{pc}) \ = \mathsf{IdentComp}(pid,\rho))\end{array}}$$

Interaction 3

$(pid,\ met(\text{parL}),\ obj),\rho \vdash_{\text{PSubF}} info\_cond(\text{x}\ \bullet \text{x} = (\mathtt{pc},\ obj.met(\text{parl}),$
$\mathtt{pc´}) \wedge$
$\mathtt{ID}(\mathtt{pc}) = \mathsf{IdentComp}(pid,\rho))$

Interaction 4

$(pid,\ ach\_id(\mathtt{term}),\ ach\_id),\rho \vdash_{\text{PSubF}} info\_cond(\text{x}\ \bullet \text{x} = (\mathtt{pc},$
$!\ ach\_id(\mathtt{term}),\ \mathtt{pc´}) \wedge$
$\mathtt{ID}(\mathtt{pc}) = \mathsf{IdentComp}(pid,\rho))$

Interaction 5

$(ach\_id,\ ach\_id(\mathtt{term}),\ pid),\ \rho \vdash_{\text{PSubF}} info\_cond(\text{x}\ \bullet \text{x} = (\mathtt{pc},$
$?\ ach\_id(\mathtt{term}),\ \mathtt{pc´}) \wedge$
$\mathtt{ID}(\mathtt{pc}) = \mathsf{IdentComp}(pid,\rho))$

Interaction 6

$(pid_1,\ sch\_id(\mathtt{term}),\ pid_2),\ \rho \vdash_{\text{PSubF}}$
$info\_cond(\text{x}\ \bullet \text{x} = (\mathtt{pc}_1,\ !\ sch\_id(\text{val}),\ \mathtt{pc}'_1),\ (\mathtt{pc}_2,\ ?\ sch\_id(),\mathtt{pc}'_2) \wedge$
$\mathtt{ID}(\mathtt{pc}_1) = \mathsf{IdentComp}(pid_1,\rho) \wedge \mathtt{ID}(\mathtt{pc}_2) = \mathsf{IdentComp}(pid_2,\rho))$

Interaction 7

$(pid_1,\ sch\_id(),\ pid_2),\ \rho \vdash_{\text{PSubF}}$
$info\_cond(\text{x}\ \bullet \text{x} = (\mathtt{pc}_1,\ !\ sch\_id(),\ \mathtt{pc}'_1)\ ,\ (\mathtt{pc}_2,\ ?\ sch\_id(),\mathtt{pc}'_2) \wedge$
$\mathtt{ID}(\mathtt{pc}_1) = \mathsf{IdentComp}(pid_1,\rho) \wedge \mathtt{ID}(\mathtt{pc}_2) = \mathsf{IdentComp}(pid_2,\rho))$

Annotations

$$\frac{bexp_i \vdash_{\text{MExp}} \mathtt{bexp}_i \quad i = 1, \ldots, n}{bexp_1 \ldots bexp_n \vdash_{\text{Annots}} \mathtt{bexp}_1 \wedge \ \ldots\ \wedge \mathtt{bexp}_n}$$

Chapter 5

JTN translation into Java

5.1 Introduction

In this chapter, we define a translation function that returns a running Java program from a `JTN` specification.

Here we briefly sketch out the main ideas of the translation of `JTN` into Java.

- A `JTN` datatype is translated into a Java class, even if in `JTN` it is described by giving its constructors and operations, without using any OO notion. The constructors are implemented by fields and the operations by methods. Many predefined datatypes are obtained by combining Java primitive data (e.g., integer and char) and extending and specializing Java standard classes (e.g., Vectors and Stack).

- A `JTN` object class is translated into a Java class, where the fields implement the fields and the methods are the direct encoding of the corresponding methods specified in the body diagram.

- A `JTN` process class is translated into a Java thread class, where the state constructors are translated into fields of the class; the unique method is `run`, whose code is the translation of the behaviour diagram of the class; the communication channels are fields referencing particular objects that implement synchronous and asynchronous channels. The class has also some fields referencing the objects declared in the interface, that supply methods to the process class.

- A synchronous connector $conn_S$ is translated into an object of a special class that acts as a "synchronizer"; when a process P wants to perform a communication on a channel connected to $conn_S$, it accesses such object and checks whether the communication can be immediately completed, that is if there exists a process P' already suspended, waiting for completing the synchronization. In that case, P reads (or writes) the message, if any, and then resumes P'. Otherwise, P writes its identity (and the message, if it is performing a write operation) and then suspends itself until some process will resume it.

- An asynchronous channel is realized by using some Java streams. Simple communication with the external environment is implemented by an object stream connected with an external file or socket.

  For asynchronous broadcast communications, there are also special threads, acting as connectors, that read a data from the streams of the writing processes and replicate it to the stream of each one of the reading processes. Finally, named asynchronous communication is implemented by adding the identities of the sender and of the receiver to each message. A connector thread is charged to dispatch the message into the correct pipelined stream.

Let us remember that a pre-processing phase orders the `use` relations and eliminates the `inheritance` relations; this greatly simplify the translation phase. Furthermore, we assume that the specification is statically correct.

The translation is formally described as a function applied to the textual counterpart of a `JTN` specification (see Appendix A) that returns Java code. Its definition follows, as much as possible, the structure of the textual presentation, also using some auxiliary functions. The sets in the domains of these functions have the same names of the corresponding non-terminals of the grammar of Appendix A.

The translation of a fragment of a `JTN` specification may depend on its context, thus we need some relevant information (called *environment* from now on) about the translation of other fragments.

In any of the following sections we describe the translation of the various parts of a `JTN` specification describing what is the produced code and if (and how) the environment is modified.

The translation function for the `JTN` constructs of kind Term will be named TJ_{Term} and its functionality will be either

$$\mathrm{TJ}_{Term}\colon \mathsf{Term} \to (\mathrm{Env} \to \mathrm{Java} \times \mathrm{Env})$$

(the environment is modified) or

$$\mathrm{TJ}_{Term}\colon \mathsf{Term} \to (\mathrm{Env} \to \mathrm{Java})$$

(the environment is not modified).

We will denote the translation of a generic construct, say c, of kind Term in the environment e by $\mathrm{TJ}_{Term}(c)[e]$.

In several cases, the translation operates on lists of elements of the same kind (e.g., lists of constructors for datatypes etc ...); this requires two translation functions; one operating on single elements and one operating on lists, the latter simply applies the former to the elements of the list. The general schema is:

$$\mathrm{TJ}_{TermList} \colon \mathsf{Term}^* \to (\mathrm{Env} \to \mathrm{Java} \times \mathrm{Env})$$

$$\mathrm{TJ}_{Term} \colon \mathsf{Term} \to (\mathrm{Env} \to \mathrm{Java} \times \mathrm{Env})$$

$\mathrm{TJ}_{TermList}(t_1 \ ; \ \ldots \ ; \ t_n)[e] = (c_1 \ \ldots c_n, \ e_n)$
where
$(c_1, \ e_1) = \mathrm{TJ}_{Term}(t_1)[e] \ \ldots (c_n, \ e_n) = \mathrm{TJ}_{Term}(t_n)[e_{n-1}]$

Thus, from now on we omit the definition of the function operating on lists, every time it is easy and intuitive to understand how it is defined, giving explicitly only the definition of the function operating over the elements of the list.

5.2 Class Diagram

5.2.1 The Class environment

The class environment (i.e., the information on the context used for the translation of the `JTN` class diagrams) is an n-upla of sets, containing class names, declarations of constructors, operations, fields, methods, channels, local variables, and the encoding of the constructors of the process classes.

CEnv denotes the set of the class environments. It is ranged over by the metavariable ρ.

Below we list the functions used to operate on CEnv.

5.2.1.0.5 To add or update some information

- $\mathsf{AddField} \colon \mathsf{Field}_{\mathsf{id}} \times \mathsf{Type}_{\mathsf{id}} \times \mathrm{CEnv} \to \mathrm{CEnv}$

  adds a field with its type to the information about the current class

- AddCon: $\mathsf{Con}_{\mathsf{id}} \times \mathsf{Arg}^* \times \mathrm{CEnv} \to \mathrm{CEnv}$

  adds a constructor, with its arity, to the information about the current class

- AddOp: $\mathsf{Opn}_{\mathsf{id}} \times \mathsf{Type}_{\mathsf{id}}{}^* \times \mathsf{Type}_{\mathsf{id}} \times \mathrm{CEnv} \to \mathrm{CEnv}$

  adds an operation, with its arity, to the information about the current class

- $\mathsf{AddMethod}$: $\mathsf{Meth}_{\mathsf{id}} \times \mathsf{Type}_{\mathsf{id}}{}^* \times \mathsf{Type}_{\mathsf{id}} \times \mathrm{CEnv} \to \mathrm{CEnv}$

  adds a method, with its arity, to the information about the current class

- $\mathsf{AddVarDecs}$: $\mathsf{VarDec}^* \times \mathrm{CEnv} \to \mathrm{CEnv}$

  adds a list of variable declarations to the information about the current class

- $\mathsf{AddVarArg}$: $\mathsf{Var}_{\mathsf{id}} \times \mathsf{Var}_{\mathsf{id}} \times \mathrm{CEnv} \to \mathrm{CEnv}$

  adds the correspondence between a variable used in a pattern and a variable used as a formal parameter of a method call

- $\mathsf{AddClassName}$: $\mathsf{Type}_{\mathsf{id}} \times \mathrm{CEnv} \to \mathrm{CEnv}$

  adds a new class name, that also becomes the current class

- $\mathsf{AddAChan}$ ($\mathsf{AddSChan}$): $\mathsf{Type}_{\mathsf{id}} \times \mathsf{IOSym} \times \mathsf{Chan}_{\mathsf{id}} \times \mathsf{Type}_{\mathsf{id}} \times \mathrm{CEnv} \to \mathrm{CEnv}$

  adds an asynchronous (synchronous) channel to the information about the current class; it also adds updates the list of field of the current class, recording the field corresponding to the channel.

5.2.1.0.6 To retrieve some information

- $\mathsf{CurrentClass}$: $\mathrm{CEnv} \to \mathsf{Type}_{\mathsf{id}}$

  returns the name of the current class

- $\mathsf{Constructor}$: $\mathsf{Con}_{\mathsf{id}} \times \mathsf{Type}_{\mathsf{id}} \times \mathrm{CEnv} \to \mathsf{Type}_{\mathsf{id}}{}^*$

  given a constructor name and a class, returns the types of its arguments

- $\mathsf{Operation}$: $\mathsf{Opn}_{\mathsf{id}} \times \mathsf{Type}_{\mathsf{id}} \times \mathrm{CEnv} \to (\mathsf{Type}_{\mathsf{id}}{}^* \times \mathsf{Type}_{\mathsf{id}})$

  given an operation name and a class, returns the types of its arguments and its return type

- Method: $\mathsf{Meth}_{\mathsf{id}} \times \mathsf{Type}_{\mathsf{id}} \times \mathrm{CEnv} \to (\mathsf{Type}_{\mathsf{id}}{}^* \times \mathsf{Type}_{\mathsf{id}})$

  given a method name and a class, returns the types of its arguments and its return type

- $\mathsf{CurrentOperation}\colon \mathrm{CEnv} \rightarrow \mathsf{Opn}_{\mathsf{id}}$

  returns the name of the current operation

- $\mathsf{CurrentMethod}\colon \mathrm{CEnv} \rightarrow \mathsf{Meth}_{\mathsf{id}}$

  returns the name of the current method

- $\mathsf{Fields}\colon \mathsf{Type}_{\mathsf{id}} \times \mathrm{CEnv} \rightarrow (\mathsf{Field}_{\mathsf{id}} \times \mathsf{Type}_{\mathsf{id}})^*$

  returns the list of the fields, with their types, of a given class

- $\mathsf{Encoding}\colon \mathsf{Con}_{\mathsf{id}} \times \mathsf{Type}_{\mathsf{id}} \times \mathrm{CEnv} \rightarrow \mathsf{integer}$

  returns the integer value encoding a constructor of a class

- $\mathsf{AChan}\ (\mathsf{SChan})\colon \mathsf{Type}_{\mathsf{id}} \times \mathrm{CEnv} \rightarrow {\mathsf{Chan}_{\mathsf{id}}}^*$

  returns the names of the asynchronous (synchronous) channels declared in the interface of a process class

- $\mathsf{ActPar}\colon \mathsf{Var}_{\mathsf{id}} \times \mathrm{CEnv} \rightarrow \mathsf{Var}_{\mathsf{id}}$

  returns the name of the actual parameter corresponding to a variable name occurring in an operation or method body. If the variable is just a local auxiliary variable, thus it is not the name of a formal parameter, the function returns the variable itself.

- $\mathsf{VarType}\colon \mathsf{Var}_{\mathsf{id}} \times \mathrm{CEnv} \rightarrow \mathsf{Type}_{\mathsf{id}}$

  returns the type of a variable

- $\mathsf{MessageType}\colon \mathsf{Chan}_{\mathsf{id}} \times \mathsf{Type}_{\mathsf{id}} \times \mathrm{CEnv} \rightarrow \mathsf{Type}_{\mathsf{id}}$

  returns the message type of a channel declared inside a process class

- $\mathsf{ReceiverType}\colon \mathsf{Chan}_{\mathsf{id}} \times \mathsf{Type}_{\mathsf{id}} \times \mathrm{CEnv} \rightarrow \mathsf{Type}_{\mathsf{id}}$

  returns the type of the receiver processes of a channel for named communications

5.2.2 Class diagram

A class diagram is a list of datatype, object and process class diagrams, each one containing the interface and the body of the class plus the information about the relations with the other classes.

Given a class diagram, its translation is simply the sequence of Java classes obtained by the translation of each class in the diagram.

$$\mathrm{TJ}_{Class}\colon \mathsf{Class\_dia} \rightarrow \mathrm{Java}$$

$\mathrm{TJ}_{Class}(datatypes,\ objdias,\ pdias) = c_1\ c_2\ c_3$
where
$(c_1,\ \rho_1) = \mathrm{TJ}_{Datas}(datatypes)[\rho_0]$
$(c_2,\ \rho_2) = \mathrm{TJ}_{Objs}(objdias)[\rho_1]$
$(c_3,\ \rho_3) = \mathrm{TJ}_{Procs}(pdias)[\rho_2]$
ρ_0 is the empty environment, that is the n-upla of empty sets.

5.3 Datatype

The translation of a datatype diagram returns a Java class whose name is the name of the datatype and whose body is obtained by composing the fields implementing the (public and private) constructors and the methods implementing the (public and private) operations of the datatype, plus a special method that is the constructor of the class instances.

Let us remark that the public and private constructors are translated in the same way, into private fields of the class; we assume that the `JTN` specification is statically correct, so we do not need to use different access modifiers for the public and the private operations of a class (they are all public).

The inheritance relation is translated with the Java construct `extends`. Notice that, instead, the use relations are ignored: we assume that the pre-processing phase correctly orders the datatypes, thus all the classes in the usage list for a class T have already been translated and can be used by the Java class that implements T.

The variables declared in the diagram are ignored, because they are the variables appearing in the pattern expressions of the operations definitions, thus their types are deduced from the operation functionality and are translated by the formal parameters of the methods implementing the operations. The environment returned by the translation records the constructors and the operations of the datatype.

$$\mathrm{TJ}_{Data}: \mathsf{DataType\_dia} \to (\mathrm{CEnv} \to \mathrm{Java} \times \mathrm{CEnv})$$

- *The datatype inherits from another one*
  $\mathrm{TJ}_{Data}(T,$ `use` $TL,$ `inherit` $T',\ Pu\_cons,\ Pr\_cons,\ Pu\_opndefs,\ Pr\_opndefs)[\rho]=$
  (`class` T `extends` T' $\{c_1\ c_2\ c_3\ \},\ \rho_2)$
  where
  $\rho' = \mathsf{AddClassName}(T,\ \rho)$
  $(c_1,\ \rho_1) = \mathrm{TJ}_{Cons}(Pr\_cons;Pu\_cons)[\rho']$
  $(c_2,\ \rho_2) = \mathrm{TJ}_{Opns}(Pr\_opndefs;Pu\_opndefs)[\rho_1]$
  $c_3 = \mathsf{ClassConstructor}(T,\rho_2)$

The auxiliary function

$\mathsf{ClassConstructor}: \mathsf{Type}_{\mathsf{id}} \times \mathrm{CEnv} \rightarrow \mathrm{Java}$

returns the code for the constructor method of a class, reading the information about the fields in the environment

$$\mathsf{ClassConstructor}(T,\ \rho) = \begin{array}{l} \texttt{static}\ T(T_1\ \texttt{x}_1, \ldots, T_n\ \texttt{x}_n)\{ \\ \texttt{new}\ T(); \\ f_1\texttt{=}\ \texttt{x}_1; \\ \ldots \\ f_n\texttt{=}\ \texttt{x}_n;\} \end{array}$$

where

$\mathsf{Fields}(T,\rho) = (f_1, T_1); \ldots ; (f_n, T_n)$

$\texttt{x}_1, \ldots\ \texttt{x}_n$ are the standard names of the formal parameters of a method.

- *The datatype does not inherit from another one*
  $\mathrm{TJ}_{Data}(T,\ \texttt{use}\ TL,\ \texttt{inherit}\ , Pu\_cons,\ Pr\_cons,\ Pu\_opndefs,\ Pr\_opndefs)[\rho]=$
  $(\texttt{class}\ T\ \{c_1\ c_2\ c_3\ \},\ \rho_2)$
  where
  $\rho' = \mathsf{AddClassName}(T,\ \rho)$
  $(c_1,\ \rho_1) = \mathrm{TJ}_{Cons}(Pr\_cons; Pu\_cons)[\rho']$
  $(c_2,\ \rho_2) = \mathrm{TJ}_{Opns}(Pr\_opndefs; Pu\_opndefs)[\rho_1]$
  $c_3 = \mathsf{ClassConstructor}(T, \rho_2)$

where $\mathsf{ClassConstructor}: \mathsf{Type}_{\mathsf{id}} \times \mathrm{CEnv} \rightarrow \mathrm{Java}$

is the auxiliary function, defined before, that returns the code of the method constructor for a class.

Constructors

We distinguish between 0-ary and n-ary constructors. A 0-ary constructor describes a value, such as the empty list, that does not have a meaningful internal structure. Its translation is a particular instance of the corresponding class contained in a class field, created just once as part of the class initialization, whose name is the same of the constructor. Furthermore, being a stateless object, there is no problem with the concurrent access and updating of its internal state by threads. We can refer the value represented by a 0-ary constructor *con()* of class T from any point in the code of another class just by $T.con$; thus we do not have to create an object and then explicitly pass its reference everywhere it is used.

We implement all the n-ary constructors of a datatype by equipping the corresponding class with one field of type integer (named `kind`) containing an encoding of the constructor name and with other fields that contain the values of the arguments of all the constructors; at any time, the value of the field `kind` determines which fields contain meaningful values.

More precisely, we translate the n-ary constructors con_1, ... , con_m for a datatype T as follows:

- declare a field `kind` of type `int`; the constructors are mapped into integer values following the order in the `JTN` specification;
- Let $\hat{T}_1, \ldots \hat{T}_h$ be the distinct types appearing in the functionalities of $con_1, \ldots, con_m$; for every type $\hat{T}_j$, we calculate, for every constructor con_i, the number $\#(\hat{T}_j, i)$ of arguments of con_i having type $\hat{T}_j$.

  Then, if $k_j = \#(\hat{T}_j)$ is the maximum among $\#(\hat{T}_j, 1), \ldots, \#(\hat{T}_j, n)$, we define just k_j fields `the_Tj_1, ... ,the_Tj_k`$_j$, of type T_j, because every state can be described by using, at most, k_j fields of type T_j.

  When `kind=` i, i.e., when the constructor con_i has $j < k_j$ arguments of type T, only the first j fields `the_T_1`, ... , `the_T_j` contain meaningful values and the other ones are ignored.

More precisely, given $con_1(T_1^1; \ldots; T_{n_1}^1), \ldots, con_m(T_1^m; \ldots; T_{n_m}^m)$, let

$\{\hat{T}_1, \ldots, \hat{T}_h\}$ s.t. for each i and j, $T_i^j \in \{\hat{T}_1, \ldots, \hat{T}_h\}$ is the set of the types used by the constructors, then:

$\#(\hat{T}_j, i) = k(i, j)$ iff

$\exists\ s_1, \ldots, s_{k(i,j)} \in \{1, \ldots, n_i\}.\ T_{s_1}^i = \cdots = T_{s_{k(i,j)}}^i = \hat{T}_j$ and $\not\exists k.k \notin \{s_1, \ldots, s_{k(i,j)}\}$ and $T_k^i = \hat{T}_j$

Finally we define: $k(j) = \#(\hat{T}_j) = max\{k(i,j) \mid i = 1, \ldots, h\}$

$$\mathrm{TJ}_{Cons}: \mathsf{ConDec}^* \to (\mathrm{CEnv} \to \mathrm{Java} \times \mathrm{CEnv})$$

$$\mathrm{TJ}_{Cons}\begin{pmatrix} zcon_1();\ \ldots\ ;\ zcon_p(); \\ con_1(\,T_1^1;\ \ldots\ ;\ T_{n_1}^1); \\ \ldots \\ con_m(\,T_1^m;\ \ldots\ ;\ T_{n_m}^m) \end{pmatrix}[\rho] \;=\; \begin{pmatrix} \texttt{int kind;} \\ \texttt{static } T\ zcon_1 \texttt{ = new } T\texttt{();} \\ \ldots \\ \texttt{static } T\ zcon_p \texttt{ = new } T\texttt{();} \\ \hat{T}_1\ \texttt{the\_}\hat{T}_1\texttt{\_1;} \\ \ldots \\ \hat{T}_1\ \texttt{the\_}\hat{T}_1\texttt{\_k}_1\texttt{;} \\ \ldots \\ \hat{T}_h\ \texttt{the\_}\hat{T}_h\texttt{\_1;} \\ \ldots \\ \hat{T}_h\ \texttt{the\_}\hat{T}_h\texttt{\_k}_h\texttt{;} \end{pmatrix},\ \rho'$$

where ρ'=
$\mathsf{AddCons}((zcon_1, \Lambda);\ \ldots\ ;\ (zcon_p, \Lambda);(con_1, T_1^1;\ \ldots\ ;\ T_{n_1}^1);\ \ldots\ ;\ (con_m, T_1^m;\ \ldots\ ;\ T_{n_m}^m),\rho)$

AddCons is the obvious extension of AddCon to lists of constructors, for $i = 1, \ldots, h$ $\hat{T}_i$ is calculated as described above.

The updated environment ρ' associates with the type T the names and the functionalities of the constructors: we do not pass the integer value that is the encoding of each non-zero constructor as it can be easily derived from the order of the list passed to AddCons (the value of the encoding of con_i is i).

5.3.0.0.7 Operation

We assume, without loss of generality, that the arguments of an operation are ordered so that, if there is at least an argument whose type is the current datatype, such argument must be the first one. Thus, an operation of a datatype T has either of the following two forms:

- opn: $T \times T_1 \cdots \times T_n \to T_r$
- opn: $T_1 \times \cdots \times T_n \to T_r$, $T_i \neq T$ for all $i = 1, \ldots, n$.

In both cases, opn is implemented by a Java method

$T_r\ opn(\,T_1\ \texttt{x}_1\texttt{,}\ \ldots, T_n\ \texttt{x}_n\texttt{);}$

in the class T. In the second case, it is a class method because it has no argument of type T, so it cannot be called on particular instances of class T.

The translation function TJ_{Opn} returns a method whose name is the the operation name and whose return type is the type of the operation codomain. Notice that the translation of the parameter list modifies the environment, because the names of the formal parameters are passed to the function that translates the operation body and this information is discarded after the body translation. Instead TJ_{Opn} updates the environment by adding the functionality of the translated operation.

$$\mathrm{TJ}_{Opn} : \mathsf{OpnDef} \rightarrow (\mathrm{CEnv} \rightarrow \mathrm{Java} \times \mathrm{CEnv})$$

$$\mathrm{TJ}_{Opn}(opn,\ T_1;\ \ldots\ ;\ T_n,\ T_r,\ opnbody)[\rho] =$$
$$\begin{cases} (\texttt{public synchronized}\ T_r\ \ opn(c_1)\{c_2\},\ \rho') & \textbf{if}\ \ T = T_1 \\ (\texttt{public synchronized static}\ T_r\ \ opn(c_1)\ \{c_2\},\ \rho') & \textbf{otherwise} \end{cases}$$
where
$$\rho' = \mathsf{AddOp}(opn, T_1;\ \ldots\ ;\ T_n, T_r,\ \rho)$$
$$T = \mathsf{CurrentClass}(\rho)$$
$$c_1 = \mathrm{TJ}_{Pars}(T_1;\ \ldots\ ;\ T_n)[\rho]$$
$$c_2 = \mathrm{TJ}_{OpnBody}(opnbody)[\rho']$$

5.3.0.0.8 Operation parameters

The translation of the list of the types of the operation parameters returns the formal parameter list of the corresponding Java method. If the first type is the current type, it does not correspond to any parameter of the method, because the first argument of the operation denotes the object receiving the method call.

$$\mathrm{TJ}_{Pars} : \mathsf{Type}_{\mathsf{id}}{}^{*} \rightarrow (\mathrm{CEnv} \rightarrow \mathrm{Java})$$

$$\mathrm{TJ}_{Pars}(T_1;\ \ldots\ ;\ T_n)[\rho] = \begin{cases} T_1\ \mathtt{x}_1,\ \ldots\ ,\ T_n\ \mathtt{x}_n & \textbf{if}\ \ T_1 \neq \mathsf{CurrentClass}(\rho) \\ T_2\ \mathtt{x}_1,\ \ldots\ ,\ T_n\ \mathtt{x}_{n-1} & \textbf{otherwise} \end{cases}$$

5.3.0.0.9 Operation Body

The operation body is a list of conditional rules, each one implemented by an `if` statement whose condition checks if the pattern of the operation is matched by the arguments of the method, whereas the body is a sequence of nested `else` clauses, containing the translations of the conditional expressions associated with the pattern. The environment is not modified.

If the type of the first argument is the datatype we are translating, the argument denotes the object on which the method is called. Thus, in the method body, we refer to this argument by using the `this` keyword.

$$\mathrm{TJ}_{CondRule}: \mathsf{CondRule} \rightarrow (\mathrm{CEnv} \rightarrow \mathrm{Java})$$

$$\mathrm{TJ}_{CondRule}(opn(pts), condexp_1; \ldots ; condexp_k)[\rho] = \begin{array}{l} \texttt{if}(c) \\ \{\mathrm{TJ}_{CondExp}(condexp_1)[\rho'] \\ \texttt{else} \\ \ldots \\ \texttt{else} \\ \mathrm{TJ}_{CondExp}(condexp_k)[\rho'] \ \} \end{array}$$

where
$(c,\rho') = \mathrm{TJ}_{OpnPatt}(opn,\ pts)[\rho]$

5.3.0.0.10 Operation pattern

The translation of a pattern for an operation is a boolean expression obtained by composing the translations of its arguments.

$$\mathrm{TJ}_{OpnPatt}: \mathsf{Opn}_{\mathsf{id}} \times \mathsf{Pattern}^* \rightarrow (\mathrm{CEnv} \rightarrow \mathrm{Java} \times \mathrm{CEnv})$$

$\mathrm{TJ}_{OpnPatt}(opn,\ pt_1;\ \ldots\ ;\ pt_n)[\rho] = ((c_1)\ \texttt{\&\&}\ \ldots \texttt{\&\&}\ (c_n),\ \rho_n)$

where
$T = \mathsf{CurrentClass}(\rho)$

$$\left.\begin{array}{l} (c_1,\ \rho_1) = \mathrm{TJ}_{Patt}(pt_1,\ \texttt{this},\ T)[\rho] \\ \ldots \\ (c_n,\ \rho_n) = \mathrm{TJ}_{Patt}(pt_n,\ \texttt{x}_{n-1},\ T_n)[\rho_{n_1}] \end{array}\right\} \text{if } \mathsf{Operation}(opn, T, \rho) = T;\ \ldots\ ;\ T_n,\ T_r;$$

$$\left.\begin{array}{l} (c_1,\ \rho_1) = \mathrm{TJ}_{Patt}(pt_1,\ \texttt{x}_1,\ T_1)[\rho] \\ \ldots \\ (c_n,\ \rho_n) = \mathrm{TJ}_{Patt}(pt_n,\ \texttt{x}_n,\ T_n)[\rho_n] \end{array}\right\} \text{if } \mathsf{Operation}(opn, T, \rho) = T_1;\ \ldots\ ;\ T_n,\ T_r \text{ and } T \neq T_1$$

5.3.0.0.11 Pattern

The translation of a pattern returns a boolean expression that checks if the value of a field (or the whole object, if the second argument is `this`) matches a pattern.

$$\mathrm{TJ}_{Patt}: (\mathsf{Pattern} \times \mathsf{Field}_{\mathsf{id}} \times \mathsf{Type}_{\mathsf{id}}) \rightarrow (\mathrm{CEnv} \rightarrow \mathrm{Java} \times \mathrm{CEnv})$$

It distinguishes three cases:

- *The pattern is built by a 0-ary construct, thus it denotes an instance of a class and the returned code checks if the variable references such instance*

  $\mathrm{TJ}_{Patt}(con(),id,T)[\rho] = (id\ \texttt{=}\ T.con,\rho)$

- *The pattern is built by a non 0-ary construct, thus the returned code checks whether the values of the fields of the object referenced by the variable match the expressions inside the pattern*

  $\mathrm{TJ}_{Patt}(con(pt_1;\ \ldots\ ;\ pt_n),\ id,\ T)[\rho] =\ ((id\texttt{.kind = i)\&\& (}c_1\texttt{)\&\&}\ \ldots\texttt{\&\& (}c_n\texttt{)},\ \rho_n)$
  where
  $\mathrm{i} = \mathsf{Encoding}(con,\ T,\ \rho)$
  $\mathsf{MeaningfulFields}(con,T,\rho) = (f_1,\ T_1);\ \ldots\ ;(f_n,\ T_n)$
  $(c_1,\ \rho_1) = \mathrm{TJ}_{Patt}(pt_1,\ id.f_1,T_1)[\rho]$
  ...
  $(c_n,\ \rho_n) = \mathrm{TJ}_{Patt}(pt_n,\ id.f_n,T_n)[\rho_{n-1}]$

  The auxiliary function $\mathsf{MeaningfulFields}\colon \mathsf{Con_{id}} \times \mathsf{Type_{id}} \times \mathrm{CEnv} \to \mathsf{Field_{id}}^* \times \mathrm{CEnv}$ returns the list of the fields implementing the arguments of a constructor.

- *The pattern is a variable, thus the boolean expression is simply true and the environment records the correspondence between the variable name and the argument of the operation (needed for the translation of the operation body)*

  $\mathrm{TJ}_{Patt}(x,id,T)[\rho] = (\texttt{true},\ \mathsf{AddVarArg}(id,x,\rho))$

5.3.0.0.12 Conditional expression

The translation of a conditional expression is an `if` statement whose condition translates the boolean expression, while the alternative returns the value denoted by the translation of the expression. The expression is translated in the environment whose current type is the return type of the translated operation.

$$\mathrm{TJ}_{CondExp}\colon \mathsf{CondExpr} \to (\mathrm{CEnv} \to \mathrm{Java})$$

$\mathrm{TJ}_{CondExp}(bexp,\ exp)[\rho] = \texttt{if(}\mathrm{TJ}_{Exp}(bexp,\ T)[\rho]\texttt{)\{return}\ \mathrm{TJ}_{Exp}(exp,\ T_r)[\rho]\texttt{;\}}$
where
$T = \mathsf{CurrentClass}(\rho)$
$\mathsf{CurrentOperation}(\rho) = opn$
$\mathsf{Operation}(opn,\ T,\rho) = (T_1;\ \ldots\ ;\ T_n,T_r)$

5.3.0.0.13 Expression

The translation of an expression is a reference to an object or a method call. The second argument of the TJ_{Exp} function is the type of the expression. Thus, if the expression is just a term describing a class instance (that is a datatype value), the argument denotes the class of the instance that has to be created.

$$\mathrm{TJ}_{Exp}: \mathsf{Expr} \times \mathsf{Type}_{\mathsf{id}} \to (\mathrm{CEnv} \to \mathrm{Java})$$

- *The expression is a 0-ary datatype constructor; in this case the object already exists, thus it is returned the special field of the datatype class that implements such constant value.*
  $\mathrm{TJ}_{Exp}(con(),\ T)[\rho] = T.con\texttt{()}$

- *The expression is a not 0-ary datatype constructor; in this case the corresponding object is created, (its class is the second argument of the translation function).*
  $\mathrm{TJ}_{Exp}(con(exp_1; \ldots; exp_n),\ T)[\rho] =$
  $\mathsf{CallConstructor}(con, T, \mathrm{TJ}_{Exp}(exp_1,\ T_1)[\rho]; \ldots; \mathrm{TJ}_{Exp}(exp_n, T_n)[\rho])$
  where
  $\mathsf{Constructor}(con,\ T,\ \rho) = T_1, \ldots, T_n$ (this means that T_i, for i=1, ...n, is the type of exp_i)

  $\mathsf{CallConstructor}: \mathsf{Con}_{\mathsf{id}} \times \mathsf{Type}_{\mathsf{id}} \times \mathsf{Expr}^* \times \mathrm{CEnv} \to \mathrm{Java} \times \mathrm{CEnv}$

  is an auxiliary function that, given a class name, a constructor and a list of values, returns the Java code performing a call of the class constructor using the values to initialize the new class instance. If there is no value corresponding to a formal parameter, it is passed a null value.

  if $\mathsf{Fields}(T,\rho) = (f_1, T_1); \ldots; (f_m,\ T_m)$, the constructor of a class T is a method

  $T(\mathsf{Integer}\ \mathrm{k},\ T_1\ \mathtt{x}_1, \ldots,\ T_m\ \mathtt{x}_m)$ thus

  $\mathsf{CallConstructor}(con,\ T,\ c_1; \ldots; c_n,\ \rho) = T(\mathsf{Encoding}(con, T, \rho), exp_1; \ldots; exp_m)$ where $\mathsf{Encoding}(con,\ T,\ \rho)$ is the integer value encoding of the constructor con in class T.

  The expression exp_i is defined as follows:

  $exp_i = \begin{cases} c_i \text{if exists } j.\ f'_j = f_i \\ \mathtt{null} \text{ otherwise} \end{cases}$

  where $f_1', \ldots, f_n' = \mathsf{MeaningfulFields}(con,\ T,\ \rho)$ are the fields used to implement the arguments of con.

- *The expression is an operation application translated by an instance method, such method is called on the object denoted by the translation of the first argument of the*

operation.
$\mathrm{TJ}_{Exp}(opn(exp_1; \ldots ; exp_n),\ T_r)[\rho] =$
$\mathrm{TJ}_{Exp}(exp_1, T)[\rho]).opn(\mathrm{TJ}_{Exp}(exp_2, T_2)[\rho], \ldots , \mathrm{TJ}_{Exp}(exp_n, T_n)[\rho])$
where
$\mathsf{CurrentClass}(\rho) = T$
$\mathsf{Operation}(opn, T, \rho) = T; \ldots ; T_n,\ T_r$

- *The expression is an operation application translated by a class method, such method is called on the current class.*
  $\mathrm{TJ}_{Exp}(opn(exp_1; \ldots ; exp_n),\ T_r)[\rho] = T.opn(\mathrm{TJ}_{Exp}(exp_1, T_1)[\rho], \ldots , \mathrm{TJ}_{Exp}(exp_n, T_n)[\rho])$
  where
  $\mathsf{CurrentClass}(\rho) = T$
  $\mathsf{Operation}(opn, T, \rho) = T_1; \ldots ; T_n,\ T_r$
  $T_1 \neq T$

- *The expression is a local method call, it is translated by a call of the corresponding method to the object denoted by* `this`*. The parameters are the translation of the parameters of the* `JTN` *method call.*
  $\mathrm{TJ}_{Exp}(met(exp_1; \ldots ; exp_n),\ T_r)[\rho] = \mathtt{this}.\ met(\mathrm{TJ}_{Exp}(exp_1, T_1)[\rho], \ldots , \mathrm{TJ}_{Exp}(exp_n, T_n))[\rho])$
  where
  $T = \mathsf{CurrentClass}(\rho)$
  $\mathsf{Method}(met, T, \rho) = T_1; \ldots ; T_n,\ T_r$

- *The expression is a variable, it is returned the parameter name corresponding to the variable.*
  $\mathrm{TJ}_{Exp}(x, T)[\rho] = \mathsf{ActPar}(x, \rho)$

5.4 Object class

The translation of an object class is quite intuitive: the fields are trivially translated into Java instance fields; if there is an explicit initial value, the field declaration comes together with the corresponding initialization expression.

The methods are translated into synchronized Java methods, with arguments and return type trivially deduced by the `JTN` method functionality.

The use relations are ignored, whereas, if class T inherits from class T', the class implementing T extends the class implementing T'.

The variable declaration list contains the variables appearing in the patterns used in the conditional rules of the method definition plus some auxiliary variables used in the left part of the assignments. The first ones are not declared as they are translated into the corresponding method arguments; the last ones are declared in the method body, immediately before their use.

5.4.0.0.14 Object class

$$\mathrm{TJ}_{Obj}: \textsf{Obj\_dia} \rightarrow (\mathrm{CEnv} \rightarrow \mathrm{Java} \times \mathrm{CEnv})$$

- *The object class inherits from another one, thus the Java class extends the corresponding one. The returned environment is the one resulting after the field and method translation, updated with the variable declarations.*
  $\mathrm{TJ}_{Obj}(T, (clientships, uses, \texttt{inherit}\ T_1), vardecs, fields, mets_1, mets_2)[\rho]=$
  $(\texttt{class}\ T\ \texttt{extends}\ T_1\ \{c_1\ c_3\ c_2\ \}, \rho_2)$
  where
  $\rho' = \textsf{AddVarDecs}(vardecs, \textsf{AddClassName}(T,\rho))$
  $(c_1, \rho_1) = \mathrm{TJ}_{Fields}(fields)[\rho']$
  $(c_2, \rho_2) = \mathrm{TJ}_{Methods}(mets_1;mets_2)[\rho_1]$
  $c_3 = \textsf{ClassConstructor}(T,\rho_2)$

- *The object class does not inherit from any class*
  $\mathrm{TJ}_{Obj}(T,(clientships, uses, \texttt{inherit}\ ), vardecs, fields, mets_1, mets_2)[\rho]=$
  $(\texttt{class}\ \mathrm{T}\ \{c_1 c_3 c_2\}, \rho_2)$
  where
  $\rho' = \textsf{AddVarDecs}(vardecs, \textsf{AddClassName}(T,\rho))$
  $(c_1, \rho_1) = \mathrm{TJ}_{Fields}(fields)[\rho']$
  $(c_2, \rho_2) = \mathrm{TJ}_{Methods}(mets_1;mets_2)[\rho_1]$
  $c_3 = \textsf{ClassConstructor}(T,\rho_2)$

5.4.0.0.15 Object field

The translation of an object field is the declaration of a Java private field (with the initialization, if there is any initialization expression in the specification).

$$\mathrm{TJ}_{Field}: \textsf{FieldDec} \rightarrow (\mathrm{CEnv} \rightarrow \mathrm{Java} \times \mathrm{CEnv})$$

$\mathrm{TJ}_{Field}(f,T,exp)[\rho] = (\texttt{private}\ T\ f\ \texttt{=}\ \mathrm{TJ}_{Exp}(exp,\ T)[\rho], \textsf{AddField}(f, \textsf{CurrentClass}(\rho), \rho))$

$\mathrm{TJ}_{Field}(f,T,\Lambda)[\rho] = ($`private` $T\ f$, $\mathsf{AddField}(f, \mathsf{CurrentClass}(\rho), \rho))$

5.4.0.0.16 Method definition

All the methods are declared to be synchronized, so that an object cannot be accessed and modified by several concurrent threads at the same time.

$$\mathrm{TJ}_{Method}: \mathsf{MethodDef} \to (\mathrm{CEnv} \to \mathrm{Java} \times \mathrm{CEnv})$$

- *The method has a return type, the translation is a Java synchronized method with a non-void return type.*
  $\mathrm{TJ}_{Method}(met,\ T_1; \ldots ; T_k,\ T_r,\ mbody)[\rho]=$
  $($`public synchronized` $T_r\ met(T_1\ \mathtt{x}_1, \ldots, T_k\ \mathtt{x}_k)\ \{\mathrm{TJ}_{MBody}(mbody)[\rho']\}, \rho')$
  where
  $\rho' = \mathsf{AddMethod}(met,\ T_1; \ldots ; T_k,\ T_r,\ \rho)$

- *The method has no return type, the translation is a Java synchronized method with void return type.*
  $\mathrm{TJ}_{Method}(met,\ T_1; \ldots ; T_k, \mathtt{void}, mbody)[\rho']=$
  $($`public synchronized void` $met(T_1\ \mathtt{x}_1, \ldots, T_k\ \mathtt{x}_k)\{\mathrm{TJ}_{MBody}(mbody)[\rho]\ \}, \rho')$
  where
  $\rho' = \mathsf{AddMethod}(met,\ T_1; \ldots ; T_k, \mathtt{void}, \rho)$

5.4.0.0.17 Method body

The definition of a method is given by an ordered list of method cases, that is of conditional rules with pattern matching. The code of the method is the code obtained by combining the translations of these cases.

The conditional rules are translated in a way similar to the operations on data, using `if` statements to translate the pattern and the boolean conditions; the elements of the assignment lists are translated by Java assignment statements or by method calls.

$$\mathrm{TJ}_{MBody}: \mathsf{MethodBody} \to (\mathrm{CEnv} \to \mathrm{Java})$$

$\mathrm{TJ}_{MBody}(metcase_1; \ldots ; metcase_n)[\rho] =$
$\mathrm{TJ}_{MCase}(metcase_1)[\rho]$ `else` ... `else` $\mathrm{TJ}_{MCase}(metcase_n)$ `return;`

5.4.0.0.18 Method case

The translation of a method case is an `if` statement whose condition contains the translation of the pattern expression, and the body contains the translation of the conditional statements.

$$\mathrm{TJ}_{MCase}\colon \textsf{MethodCase} \to (\mathrm{CEnv} \to \mathrm{Java})$$

$\mathrm{TJ}_{MCase}(met,(pt_1;\ \ldots\ ;\ pt_n),cstms)[\rho] = \texttt{if}(c_1\ \texttt{\&\&}\ \ldots\ \texttt{\&\&}\ c_n)\ \{\mathrm{TJ}_{CondStatList}(cstms)[\rho_n]\}$
where
$(c_1,\rho_1) = \mathrm{TJ}_{Patt}(pt_1,\ \mathtt{x}_1,\ T_1)[\rho]$
$\ldots$
$(c_n,\ \rho_n) = \mathrm{TJ}_{Patt}(pt_n,\mathtt{x}_n,\ T_n)[\rho_{n-1}]$

As we have seen in the translation of the method signature, $\mathtt{x}_1,\ \ldots\ ,\mathtt{x}_n$ are the standard names of the formal parameters of a method having n parameters.

5.4.0.0.19 Conditional Statement List

The translation of a list of conditional statements is a sequence of Java `else` statements implementing the conditional alternatives of the method definition.

$$\mathrm{TJ}_{CondStatList}\colon \textsf{CondStatement}^* \to (\mathrm{CEnv} \to \mathrm{Java})$$

$$\mathrm{TJ}_{CondStatList}(cstm_1;\ldots;\ cstm_n)[\rho] = \mathrm{TJ}_{CondStat}(cstm_1)[\rho]\ \texttt{else}\ldots\texttt{else}\ \mathrm{TJ}_{CondStat}(cstm_n)[\rho]$$

5.4.0.0.20 Conditional Statement

The translation of a conditional statement is an `if` statement whose condition is the translation of the boolean expression and whose body is the translation of the statements.

$$\mathrm{TJ}_{CondStat}\colon \textsf{CondStatement} \to (\mathrm{CEnv} \to \mathrm{Java})$$

$$\mathrm{TJ}_{CondStat}(bexp,\ stm_1;\ \ldots\ ;\ stm_n)[\rho] = \begin{array}{l}\texttt{if}\ (\mathrm{TJ}_{Exp}(bexp,\ \textsf{CurrentClass}(\rho))[\rho])\ \{\\ \mathrm{TJ}_{Stat}(stm_1)[\rho]\ \ldots\mathrm{TJ}_{Stat}(stm_n)[\rho]\ \}\end{array}$$

5.4.0.0.21 Statement

A statement is either an assignment (to an object field, to a variable or to the special variable `res`) or a method call.

$$\mathrm{TJ}_{Stat}: \mathsf{Statement} \rightarrow (\mathrm{CEnv} \rightarrow \mathrm{Java})$$

$\mathrm{TJ}_{Stat}(f = exp)[\rho] = \mathtt{this}.f\ \mathtt{=}\ \mathrm{TJ}_{Exp}(exp,\ T)[\rho]\mathtt{;}$
where
$\mathsf{Fields}(\mathsf{CurrentClass}(\rho),\rho) = (f_1, T_1); \ldots ; (f, T); \ldots ; (f_n, T_n)$

$\mathrm{TJ}_{Stat}(x{=}\ exp)[\rho] = T'\ \mathtt{x;x\ =}\ \mathrm{TJ}_{Exp}(exp,\ T)[\rho]\mathtt{;}$
where
$T = \mathsf{CurrentClass}(\rho),\ \mathsf{VarType}(x,\rho) = T'$

$\mathrm{TJ}_{Stat}(\mathtt{res} = exp)[\rho] = \mathtt{return}\ \mathrm{TJ}_{Exp}(exp,\ T_r)[\rho]\mathtt{;}$
where
$\mathsf{Method}(\mathsf{CurrentMethod}(\rho),\ \mathsf{CurrentClass}(\rho),\ \rho) = T_1; \ldots ; T_n,\ T_r$

$\mathrm{TJ}_{Stat}(met(exp_1; \ldots ; exp_n))[\rho] = \mathtt{this}.met(\mathrm{TJ}_{Exp}(exp_1,\ T_1)[\rho], \ldots , \mathrm{TJ}_{Exp}(exp_n,\ T_n)[\rho])\mathtt{;}$
where
$\mathsf{Method}(met,\ \mathsf{CurrentClass}(\rho),\ \rho) = T_1; \ldots ; T_n,\ T_r$

Example The translation of the DISPLAY object class in `Calculator` is

```
class Display {
  APPLY_LIST Cont = APPLY_LIST.Empty;
  public synchronized APPLY_LIST Read(){
   return Cont;}
  public synchronized void Write(APPLY_LIST x1){
   Cont = x1;}
}
```

5.5 Process Class

A `JTN` process class is translated into a Java class that extends `Thread`. The only method is the `run` method, whose code is supplied by the translation of the behaviour diagram.

$$\mathrm{TJ}_{Proc}: \mathsf{Proc\_dia} \rightarrow (\mathrm{CEnv} \rightarrow \mathrm{Java} \times \mathrm{CEnv})$$

$\mathrm{TJ}_{Proc}((T, relations, interface, beh))[\rho] \; = (\texttt{class}\ T\ \texttt{extends Thread}\ \{c_1\ c_2\}, \rho_2)$
where
$(c_1,\rho_1) = \mathrm{TJ}_{Inter}(interface)[\mathsf{AddClassName}(T,\rho)]$
$(c_2,\rho_2) = \mathrm{TJ}_{Behav}(beh)[\rho_1]$

5.5.1 Interface Diagram

The interface diagram of a process class does not contain methods, but only the the use and clientship relations with the other classes, that are ignored, and the declaration of the objects and of the communication channels, synchronous and asynchronous, used by the process.

We assume that the `JTN` specification is correct, thus the class of the used datatype and of the objects have been already declared and their operations and methods are accessible. The inheritance relation has been also eliminated by the preprocessing phase that entirely rewrites the code of the inherited class into the inheriting one, adding the new transition capabilities.

Thus, the interface diagram is translated into the declaration of the fields containing references to the objects implementing the above channels. The class constructor of a process class creates a new instance and initializes only the fields implementing the object and channel references of the interface diagram, because the constructor is called inside the `main` method, with the correct references of the already created objects and channels. The other fields, translating the arguments of the process state constructors, are initialized in the `run` method, according with the information contained in the behaviour diagram.

$$\mathrm{TJ}_{Inter}: \mathsf{Interface\_dia} \rightarrow (\mathrm{CEnv} \rightarrow \mathrm{Java} \times \mathrm{CEnv})$$

$\mathrm{TJ}_{Inter}(obrefs, synchs, asynchs)[\rho] = (c_1\ c_2\ c_3,\ \rho_3)$
where
$(c_1,\rho_1) = \mathrm{TJ}_{ORefs}(obrefs)[\rho],$
$(c_2,\rho_2) = \mathrm{TJ}_{SRefs}(synchs)[\rho_1]$
$(c_3,\rho_3) = \mathrm{TJ}_{ARefs}(asynchs)[\rho_2]$

5.5.1.0.22 Object reference

The interface diagram contains the names of the instances of the object classes used by the process. These names are trivially translated into fields of the corresponding types; the environment is enriched with the declaration of the new fields.

$$\mathrm{TJ}_{ORef} : \mathsf{ObjRef} \to (\mathrm{CEnv} \to \mathrm{Java} \times \mathrm{CEnv})$$

$\mathrm{TJ}_{ORef}(id\colon\ T)[\rho] = (\texttt{private}\ T\ id;, \mathsf{AddField}(id,\ T,\ \rho))$

5.5.1.0.23 Channel reference

As we will see in Sect.??, the correct implementation of both synchronous and asynchronous communication channels requires some information that are available only in the architecture diagram, because we need to know which is the kind of connector to whom it is attached. More precisely, a synchronous (asynchronous) channel in the interface declaration is translated into a field whose type is a class that specializes the general class for synchronous (asynchronous) communication. The right choice can be performed only during the translation of the architecture diagram. Thus, at this level, a synchronous channel is translated into a field of the general non standard Java class `SChannel`, while an asynchronous input (output) channel is translated in the standard Java class `InputStream` (`OutputStream`) that is the superclass of a very large set of more specialized classes for input (output) communication.

The method `main`, as we will see in the following, is obtained by translating the architecture diagram; it creates every process instance with, for each field of type `SChannel` (`InputStream`, `OutputStream`), a reference to the correct sub-class of `SChannel` (`InputStream`, `OutputStream`). Let us recall that the asynchronous channels may supply simple, broadcast or named communication primitives and the exchanged messages may have any type; thus, there is a plethora of Java classes that implement every kind of asynchronous channel.

Recall that AddAChan updates the list of the fields of a class too (it is not necessary to call AddField as in the object reference case).

The updated environment records that $ach\_id(sch\_id)$ is an asynchronous (synchronous) channel for exchanging messages of type T.

$$\mathrm{TJ}_{ARef} : \mathsf{AsynCh} \to (\mathrm{CEnv} \to \mathrm{Java} \times \mathrm{CEnv})$$

$\mathrm{TJ}_{ARef}(T', ?\ , ach\_id, T)[\rho] = (\texttt{private InputStream}\ ach\_id;, \mathsf{AddAChan}(T', ?\ , ach\_id, T, \rho))$

$\mathrm{TJ}_{ARef}(T', !\ ,ach\_id, T)[\rho] = ($`private OutputStream` $ach\_id;$, $\mathsf{AddAChan}(T', !\ ,ach\_id, T, \rho))$

$$\mathrm{TJ}_{SRef} : \mathsf{SynCh} \rightarrow (\mathrm{CEnv} \rightarrow \mathrm{Java} \times \mathrm{CEnv})$$

$\mathrm{TJ}_{SRef}(io, sch\_id, T)[\rho] = ($`private SChannel` $sch\_id;$, $\mathsf{AddSChan}(io, sch\_id, T, \rho))$

Example The translation of the interface diagram for DISPLAY_DRIVER in `Calculator`is:

```
class DisplayDriver extends Thread{
   Display the_Display; // the display;
   // synchronous communication channels
   SChannel Show;
   SChannel Ok;
   // asynchronous communication channel
   OutputStream OUT;
   OutputStream REFRESH;
}
```

5.5.1.1 Behaviour Diagram

A behaviour diagram is textually represented as a list of state constructors, a list of variable declarations, the expressions defining the initial state and a behaviour graph. The translation function returns the declaration of some fields plus the code for the class constructor (that sets the values of the object and channels references) and for the `run` method, that describes the behaviour of the thread class that implements the process. The variable declarations are ignored. The variables describing the arguments of input communications are declared in the translation of such communications, just before being used. The other ones appear inside the pattern expressions, thus they are not in the code of the `run` method.

$$\mathrm{TJ}_{Behav} : \mathsf{Behaviour\_dia} \rightarrow (\mathrm{CEnv} \rightarrow \mathrm{Java} \times \mathrm{CEnv})$$

$\mathrm{TJ}_{Behav}(cons, vardecs, exps, bgraph)[\rho] = (c_1\ c_3$ `void run()`$\{\mathrm{TJ}_{Init}(exps, \rho_1)\ c_2\}, \rho_2)$
where
$(c_1, \rho_1) = \mathrm{TJ}_{PCon}(cons)[\rho]$
$c_2 = \mathrm{TJ}_{BGraph}(bgraph)[\rho_1]$
$c_3 = \mathsf{ClassConstructor}(\mathsf{CurrentClass}(\rho_1), \rho_1)$

State constructors

The state constructors are explicitly listed, with their names and functionalities, in the textual presentation of the behaviour diagram. We assume that the `initial` constructor is always the first one.

The translation of the state constructors is analogous to the translation of the datatype constructors, except for the 0-ary constructors that are simply encoded into an integer value of field `kind`.

Thread instances are not managed and passed among objects, like datatype values. Thus we do not need to create particular process class instances denoting threads without meaningful field values (because their state is described by a 0-ary construct). Furthermore, a thread can change its internal state and this is rendered simply by modifying the value of the `kind` field and of the other related fields.

The code returned by the state constructors translation completes the declarations of the fields of the process class and also defines the constructor method that will be used within the method `main` to create new instances of that class. The environment is updated by associating to the type T the information about the encoding and functionalities of the constructors: for each constructor we record the integer value that encode this constructor and the types of its arguments.

Here we assume the definitions given in Sect. 5.3 for what concerns the treatment of constructors.

$$\mathrm{TJ}_{PCon}\colon \mathsf{ConDec}^* \to (\mathrm{CEnv} \to \mathrm{Java} \times \mathrm{CEnv})$$

$$\mathrm{TJ}_{PCon}\begin{pmatrix}\texttt{initial}(\,T_1;\ \ldots;\ T_n)\\ con_1(\,T_1^1;\ \ldots;\ T_{n_1}^1);\\ \ldots;\\ con_m(\,T_1^m\ ;\ \ldots;\ T_{n_m}^m)\end{pmatrix}[\rho] = \begin{pmatrix}\texttt{int kind;}\\ \hat{T}_1\ \texttt{the\_}\hat{T}_1\texttt{\_1;}\\ \ldots\\ \hat{T}_1\ \texttt{the\_}\hat{T}_1\texttt{\_k}_1\texttt{;}\\ \ldots\\ \hat{T}_h\ \texttt{the\_}\hat{T}_h\texttt{\_1;}\\ \ldots\\ \hat{T}_h\ \texttt{the\_}\hat{T}_h\texttt{\_k}_h\texttt{;}\end{pmatrix}, \rho'$$

where
$\rho'= \mathsf{AddCons}((\texttt{initial}(\,T_1;\ \ldots;\ T_n));(con_1,\ T_1^1;\ \ldots;\ T_{n_1}^1);\ \ldots;\ (con_m,\ T_1^m;\ \ldots;\ T_{n_m}^m),\ \rho).$

Example The translation of the state constructors for DISPLAY_DRIVER in `Calculator` adds some new lines of code and defines the constructor method:

```
   int kind; // state constructor implementation
   APPLY_LIST dl; // digit list;

DisplayDriver(SChannel x1, SChannel x2, OutputStream x3, OutputStream x4,
              APPLY_LIST x5){
   super();
   kind = the_kind;
   Show = x1;
   Ok = x2;
   OUT = x3;
   REFRESH = x4;
   dl = x5;}
```

5.5.1.1.1 Behaviour graph

A behaviour graph is translated into a `run` method.

The method starts with an initialization section where we set the initial values of the fields as described in the initial state of the behaviour graph (always called `initial` and encoded by the integer value 0): if in the behaviour graph we have `initial`(exp_1; ... ; exp_n), then the field `kind` is set to 0 and the other fields corresponding to the arguments of the state constructor Initial are set to the values described respectively by the expressions exp_1, ... , exp_n.

5.5.1.1.2 Initial state

$$\mathrm{TJ}_{Init} : \mathsf{Expr}^* \to (\mathrm{CEnv} \to \mathrm{Java})$$

$$\mathrm{TJ}_{Init}(exp_1;\ \ldots\ ;\ exp_n)[\rho] = \begin{array}{l} \texttt{kind} = 0\ ; \\ f_1\ \texttt{=}\mathrm{TJ}_{Exp}(exp_1,\ T_1)[\rho]; \\ \ldots \\ f_n\ \texttt{=}\mathrm{TJ}_{Exp}(exp_n,\ T_n)[\rho]; \end{array}$$

where
$\mathsf{MeaningfulFields}(\mathrm{Initial},\ \mathsf{CurrentClass}(\rho),\ \rho) = (f_1, T_1);\ \ldots\ ;\ (f_n, T_n)$

The code for the method `run` is an infinite loop, implemented by a statement `while(true)`. Every execution of the while cycle checks which is the current state of the process; this is realized by a statement `switch`; the value associated with each `case` construct is the encoding of a state constructor and the statements after each `case` are the translation of the node of the behaviour graph that corresponds to that constructor.

Given a list $state\_case_1; \ldots ; state\_case_n$ textually representing the nodes of the behaviour graph, we assume that the pre-processing phase ordered them so that the i-th element describes the activity in the state described by the constructor whose encoding is integer i (we assume that $state\_case_o$ is the node corresponding to state Initial).

$$\mathrm{TJ}_{BGraph} : \mathsf{BehaviourGraph} \rightarrow (\mathrm{CEnv} \rightarrow \mathrm{Java})$$

$$\mathrm{TJ}_{BGraph}(state\_case_o;state\_case_1; \ldots ; state\_case_n) = \begin{array}{l} \texttt{while(true)} \\ \texttt{switch(kind)} \ \{ \\ \texttt{case(0):} \ \{c_0\}\texttt{;break;} \\ \ldots \\ \texttt{case(n):}\{c_n\} \ \texttt{;break;}\} \end{array}$$

where
$c_0 = \mathrm{TJ}_{StateCase}(state\_case_o)[\rho]$
$c_1 = \mathrm{TJ}_{StateCase}(state\_case_1)[\rho]$
...
$c_n = \mathrm{TJ}_{StateCase}(state\_case_n)[\rho]$

5.5.1.1.3 State case

Every node of the behaviour graph, called a state case, contains several state patterns from which there is a list of outcoming transitions, called *transition cases*.

Every `if` checks if the current process state corresponds to one pattern of the alternative and the body is the translation of the corresponding transition list.

$$\mathrm{TJ}_{StateCase} : \mathsf{StateCase} \rightarrow (\mathrm{CEnv} \rightarrow \mathrm{Java})$$

$$\mathrm{TJ}_{StateCase}(transcase_1; \ldots ; transcase_n)[\rho] = \mathrm{TJ}_{TranCase}(transcase_1)[\rho], \ldots , \mathrm{TJ}_{TranCase}(transcase_n)[\rho]$$
$c_1 \ \ldots c_n$

5.5.1.1.4 Transition case

Given a single transition case its translation under environment ρ is an `if` statement whose condition checks if the current state of the process matches the pattern. Inside the body, there are as much nested `if` as the transition alternatives.

$$\mathrm{TJ}_{TranCase} : \mathsf{Con}_{\mathsf{id}} \times \mathsf{Pattern}^* \times \mathsf{TransitionAlt}^* \rightarrow (\mathrm{CEnv} \rightarrow \mathrm{Java})$$

$$\mathrm{TJ}_{TranCase}(con(pts),\ trans) = \texttt{if}(c_1)\{\mathrm{TJ}_{TranList}(con(pts),trans)[\rho_1]\ \}$$
where
$(c_1,\ \rho_1) = \mathrm{TJ}_{Patt}(con(pts),\ \texttt{this},\ \mathsf{CurrentClass}(\rho))[\rho]$

5.5.1.1.5 Transition alternative

Notice that the translation of the transition alternative needs to know the initial state of the transition. This avoid dummy assignments or object creations, as described in more details below, when translating the final state of the transition. The first argument of the TJ_{Tran} function is the starting state of the transitions described by the third argument. The second argument is the boolean expression describing the conditions over the execution of such transitions.

$$\mathrm{TJ}_{Tran} : \mathsf{State} \times \mathsf{Expr} \times \mathsf{Label\&State}^{*} \rightarrow (\mathrm{CEnv} \rightarrow \mathrm{Java})$$

$$\mathrm{TJ}_{Tran}(st,\ bexp,\ labstates)[\rho] = \mathtt{if}(\mathrm{TJ}_{Exp}(bexp, \mathsf{CurrentClass}(\rho)[\rho])\ \{\mathrm{TJ}_{LabStateList}(st, labstates)[\rho]\}$$

5.5.1.1.6 Label states

$$\mathrm{TJ}_{LabStateList} : \mathsf{State} \times \mathsf{Label\&State}^{+} \rightarrow (\mathrm{CEnv} \rightarrow \mathrm{Java})$$

The translation of a sequence of labels and of the corresponding final states depends on their number and kind.

5.5.1.1.7 Single transition capability When there is just one label exiting a state, the returned code is obtained as follows:

- *A method call is translated in the expected way. The name of the method is not changed, the translation is applied to the expressions representing the called object and the arguments of the call.*

  $\mathrm{TJ}_{LabStateList}(st_1,(oid.met(exp_1;\ \ldots\ ;\ exp_n),st_2))[\rho] =$
  $\mathrm{TJ}_{Label}(oid.met(exp_1;\ \ldots\ ;\ exp_n))[\rho]$
  $\mathrm{TJ}_{State}(st_1,st_2)[\rho]$

- *The returned value is assigned to a variable that is declared.*

  $\mathrm{TJ}_{LabStateList}(st_1,(x \leftarrow oid.met(exp_1;\ \ldots\ ;\ exp_n),st_2)[\rho] =$
  $\mathrm{TJ}_{Label}(x \leftarrow oid.met(exp_1;\ \ldots\ ;\ exp_n))[\rho]$
  $\mathrm{TJ}_{State}(st_1,st_2)[\rho]$

- *A read/write operation on a synchronous/asynchronous channel is translated into a call of the corresponding method on the process field that contains the reference to the channel.*
  - $\mathrm{TJ}_{LabStateList}(st_1,(\ !\ sch\_id(exp),st_2)[\rho] =$
    $\mathrm{TJ}_{Label}(\ !\ sch\_id(exp))[\rho]$
    $\mathrm{TJ}_{State}(st_1,st_2)[\rho]$
    where
    $sch\_id \in \mathsf{SChan}(\mathsf{CurrentClass}(\rho))$

  - $\mathrm{TJ}_{LabStateList}(st_1,(\ ?\ sch\_id(x),st_2)[\rho] =$
    $\mathrm{TJ}_{Label}(\ ?\ sch\_id(x))[\rho])$
    $\mathrm{TJ}_{State}(st_1,st_2)[\rho]$
    where
    $sch\_id \in \mathsf{SChan}(\mathsf{CurrentClass}(\rho))$

  - $\mathrm{TJ}_{LabStateList}(st_1,(\ !\ ach\_id(exp),st_2))[\rho] =$
    $\mathrm{TJ}_{Label}(\ !\ ach\_id(exp))$
    $\mathrm{TJ}_{State}(st_1,st_2)[\rho]$
    where
    $ach\_id \in \mathsf{AChan}(\mathsf{CurrentClass}(\rho))$

  - *A read operation on an asynchronous channel returns an object that is casted to the correct type.*
    $\mathrm{TJ}_{LabStateList}(st_1,(\ ?\ ach\_id(x),st_2))[\rho] =$
    $\mathrm{TJ}_{Label}(\ ?\ ach\_id(x)[\rho]$
    $\mathrm{TJ}_{State}(st_1,st_2)[\rho]$
    where
    $ach\_id \in \mathsf{AChan}(\mathsf{CurrentClass}(\rho))$

  - *The channel supplies named communication, an object containing the sender identity and the message is created and passed to the method that write it on the asynchronous channel.*
    $\mathrm{TJ}_{LabStateList}(st_1,oexp\ !\ ach\_id(exp),st_2)[\rho] = \mathrm{TJ}_{Label}(oexp\ !\ ach\_id(exp))[\rho]$
    $\mathrm{TJ}_{State}(st_1,st_2)[\rho]$
    where
    $ach\_id \in \mathsf{AChan}(\mathsf{CurrentClass}(\rho))$

  - *The channel supplies named communication, an object containing the sender identity and the message is created and assigned by the method that reads objects from the asynchronous channel. A casting is required.*

$\mathrm{TJ}_{LabStateList}(st_1,y\ ?\ ach\_id(x),st_2)[\rho]\ =\ \mathrm{TJ}_{Label}(y\ ?\ ach\_id(x))[\rho]$
$\mathrm{TJ}_{State}(st_1,st_2)[\rho]$
where
$ach\_id \in \mathsf{AChan}(\mathsf{CurrentClass}(\rho))$

The variables used in the body must be declared; the list of variables in the behaviour diagram contains the declarations of the variables used in the pattern expressions (that are translated by process fields) and in the input communications that, consequently, are used in the code returned by the translation and must be declared before their use. In the following, the translations of method calls with returned values and of input communications contain the declaration of the local variable used to store the received values.

$$\mathrm{TJ}_{Label} : \mathsf{Label} \times \mathsf{Type_{id}} \rightarrow (\mathrm{CEnv} \rightarrow \mathrm{Java})$$

$\mathrm{TJ}_{Label}(oid.met(exp_1;\ \ldots\ ;\ exp_n))[\rho]\ =$
$oid.met(\mathrm{TJ}_{Exp}(exp_1,\ T_1)[\rho],\ \ldots\ ,\ \mathrm{TJ}_{Exp}(exp_n,\ T_n)[\rho])$;
where
$\mathrm{In}((oid, T), \mathsf{Fields}(\mathsf{CurrentType}(\rho),\ \rho))$
$\mathsf{Method}(met,\ T,\ \rho)\ =\ T_1;\ \ldots\ ;\ ,T_n,\ T_r$

$\mathrm{TJ}_{Label}(x\text{<- }oid.met(exp_1;\ \ldots\ ;\ exp_n))\ =\ \ T'$`x;x =` $\mathrm{TJ}_{Label}(oid.met(exp_1;\ \ldots\ ;\ exp_n),\ T)[\rho]$;
where
$\mathsf{VarType}(x,\rho)=\ T'$

$\mathrm{TJ}_{Label}(\ !\ sch\_id(exp))[\rho]\ =\ sch\_id$`.write(`$\mathrm{TJ}_{Exp}(exp,\ T)[\rho]$`);`
where
$T = \mathsf{MessageType}(sch\_id,\ \mathsf{CurrentClass}(\rho),\ \rho)$

$\mathrm{TJ}_{Label}(\ ?\ sch\_id(x))[\rho]\ =\ T'$ `x;x = (`T'`)`sch_id`.read();`
where
$\mathsf{VarType}(x,\rho)=\ T'$

$\mathrm{TJ}_{Label}(\ !\ ach\_id(exp))[\rho]\ =\ ach\_id$`.writeObject(`$\mathrm{TJ}_{Exp}(exp,\ T)[\rho]$`);`
where
$T = \mathsf{MessageType}(ach\_id,\ \mathsf{CurrentClass}(\rho),\ \rho)$

$\mathrm{TJ}_{Label}(\ ?\ ach\_id(x))[\rho]\ =$
T' `x;`

```
x = (T')ach_id.readObject();
```
where
T'= VarType(x, ρ)

$\mathrm{TJ}_{Label}(oid\,!\ ach\_id(exp))[\rho]$ = `NamedMess sm = NamedMess(this,` oid, $\mathrm{TJ}_{Exp}(exp, T)$`);`
ach_id`.writeObject(sm);`
where
T= MessageType(ach_id, CurrentClass(ρ), ρ)

$\mathrm{TJ}_{Label}(y\ ?\ ach\_id(x))[\rho]$ = T_1 `x;`
T_2 y;
`NamedMess` z;
z = `(NamedMess)`ach_id`.readObject();`
`x =` z`.Message;`
y = z`.Sender;`
where
T_1= MessageType(ach_id, CurrentClass(ρ), ρ)
T_2= ReceiverType(ach_id, CurrentClass(ρ), ρ)

We use instances of a non-standard class `NamedMess` (defined in Sect. 5.6.5)to manage messages with the identities of the sender and of the receiver.

5.5.1.1.8 Several transition capabilities

There are two cases in which the behaviour graph defines several alternatives (labelled by $l_1; \dots ; l_h$), under the same condition:

- when the process tries to complete a synchronous communication over one among h channels ($h \geq 1$ and for each i, $1 \leq i \leq h$, l_i= $!\ sch\_id_i(\ \dots)$ or l_i= $?\ sch\_id_i(\ \dots)$); the alternative is implemented by h exclusive `if` statements, whose condition tests if there is a process already waiting on that channel and, if it is so, the execution of the block performs the read/write operation and sets the new state according to the final state of the transition (see below);
- when the process continuously tests h asynchronous channels until it finds an available message ($h \geq 1$ and for each i, $1 \leq i \leq h$, l_i= $?\ ach\_id_i(\ \dots)$ or l_i= $x_i\ ?\ ach\_id_i(\ \dots)$); the alternative is implemented by a `while` cycle with h exclusive `if` statements inside. Every `if` has a condition testing if there is a message available on the channel and, if it is the case, the block performs the read operation, sets the new state according to the final state of the transition (see below) and makes false the `while` condition.

In both cases, the last label l_h could be the special label `else`.

The synchronous communication case is modified by adding a final `else` clause whose statements changes the process state according to the final state of the `else` transition.

In the asynchronous communication case, the channels are tested just one time and, if no message is available, it is performed the else transition. Thus, the translation does not use the `while` cycle; there are $h-1$ exclusive `if` statements implementing the checks for a message and a final `else` clause whose statements set the process state according to the final state of the `else` transition.

Several label alternatives, with no `else` label are translated as follows:

5.5.1.1.9 Multiple transition capabilities with synchronous communications

$\mathrm{TJ}_{LabStateList}(st,(io_1\ sch\_id_1(exp_1),st_1);\ \ldots\ ;\ (io_n sch\_id_n(exp_n),st_n))[\rho]) =$
$\texttt{if}(sch\_id_1.opn_1(\mathrm{TJ}_{Exp}(exp_1,\ T_1)[\rho]))$
  $\{\mathrm{TJ}_{State}(st,st_1)[\rho];\}$
`else`
$\ldots$
`else`
$\texttt{if}(sch\_id_{n-1}.opn_{n-1}(\mathrm{TJ}_{Exp}(exp_{n-1},\ T_{n-1})[\rho]))$
  $\{\mathrm{TJ}_{State}(st,st_{n-1})[\rho];\}$
$\texttt{else}\{sch\_id_n.opn_n(\mathrm{TJ}_{Exp}(exp_n,\ T_n)[\rho]);$
$\mathrm{TJ}_{State}(st,st_n)[\rho];\}$
where
for $i=1,\ \ldots\ ,\ n-1$
$T_i = \mathsf{MessageType}(sch\_id_i,\ \mathsf{CurrentClass}(\rho),\ \rho)$
if $T_i \neq$ `void`
$$opn_i \equiv \begin{cases}\texttt{readAvailable} \text{ if } io_i = \ ?\\ \texttt{writeAvailable} \text{ if } io_i = \ !\end{cases}$$
$$opn_n \equiv \begin{cases}\texttt{read} \text{ if } io_n = \ ?\\ \texttt{write} \text{ if } io_n = \ !\end{cases}$$
if $T_i =$ `void`
$opn_i \equiv$ `synchAvailable`
$opn_n \equiv$ `synch`

The Java `readAvailable` and `writeAvailable` methods test if the communication can be immediately completed and, in that case, return the read value or write the passed value.

The Java `read` (`write`) methods are the blocking versions, thus a call to `read` (`write`) does not return until the synchronization is not completed.

If the communication is just a signal exchange, the `synchAvailable` method returns **true** if the synchronization can immediately complete, while `synch` returns only when the synchronization has completed.

If there is an `else` label, the last operation is a non-blocking one and the final `else` clause changes the state according to the final state reached by the else transition.

$\mathrm{TJ}_{LabStateList}(st,(io_1 sch\_id_1(exp_1),st_1);\ \ldots\ ;\ (io_n sch\_id_n(exp_n),st_n);(\mathtt{else},\ st_{n+1}))[\rho]) =$
$\mathtt{if}(sch\_id_1.opn_1(\mathrm{TJ}_{Exp}(exp_1,\ T_1)[\rho]))$
$\{\mathrm{TJ}_{State}(st,st_1)[\rho]\,;\}$
$\mathtt{else}$
$\ldots$
$\mathtt{else}$
$\mathtt{if}(sch\_id_{n-1}.opn_{n-1}(\mathrm{TJ}_{Exp}(exp_{n-1},\ T_{n-1})[\rho]))$
$\{\mathrm{TJ}_{State}(st,st_{n-1})[\rho]\,;\}$
$\mathtt{else}$
$\mathtt{if}(sch\_id_n.opn_n(\mathrm{TJ}_{Exp}(exp_n,\ T_n)[\rho])$
$\mathrm{TJ}_{State}(st,st_n)[\rho];$
$\mathtt{else}\{\mathrm{TJ}_{State}(st,st_{n+1})[\rho]\,;\}$
where
for $i = 1,\ \ldots\ ,\ n-1$
$T_i = \mathsf{MessageType}(sch\_id_i,\ \mathsf{CurrentClass}(\rho),\ \rho)$
if $T_i \neq \mathtt{void}$
$$opn_i \equiv \begin{cases} \mathtt{readAvailable} \text{ if } io_i = \ ? \\ \mathtt{writeAvailable} \text{ if } io_i = \ ! \end{cases}$$
$$opn_n \equiv \begin{cases} \mathtt{read} \text{ if } io_n = \ ? \\ \mathtt{write} \text{ if } io_n = \ ! \end{cases}$$
if $T_i = \mathtt{void}$
$opn_i \equiv \mathtt{synchAvailable}$
$opn_n \equiv \mathtt{synch}$

5.5.1.1.10 Multiple transition capabilities with asynchronous simple communications

For simplicity, we consider the case in which the asynchronous communications are not named. It easy to extend the translation below to the case in which some of the alternatives are named communications.

$\mathrm{TJ}_{LabStateList}(st,(\ ?\ ach\_id_1(x_1),st_1);\ \ldots\ ;\ (\ ?\ ach\_id_h(x_h),st_h))[\rho] =$
`bool exit = false;`
`while (!exit)` $\{$
c_1
$\ldots$
$c_h\}$

where
T_i= VarType($\mathtt{x}_i$, ρ) for $i = 1, \ldots, h$
c_i= T_i $\mathtt{x}_i$;
synchronized(ach_id_i){
if(ach_id_i.available() $>$ 0)
$\mathtt{x}_i$= (T_i)ach_id_i.readObject();
$\mathrm{TJ}_{State}(st,st_i)[\rho]$;
exit = true;}

where the method `readObject(x)` assigns a message to `x` (the read message if any, otherwise the special "null" message), `x.available()` returns the number of available bytes on the channel.

If there is an else label, the translation is:

$\mathrm{TJ}_{LabStateList}(st,(\ ?\ ach\_id_1(x_1),st_1);\ \ldots\ ;\ (\ ?\ ach\_id_n(x_n),st_n),(\mathtt{else},\ st_{n+1}))[\rho]$ =
bool exit = false;
while (!exit) {
c_1
else
...
else
c_n
else
$\mathrm{TJ}_{State}(st,st_{n+1})[\rho]$}
where
T_i= VarType($\mathtt{x}_i$, ρ) for $i = 1, \ldots, h$
c_i= T_i $\mathtt{x}_i$;
synchronized(ach_id_i){
if(ach_id_i.available() $>$ 0)
$\mathtt{x}_i$= (T_i)ach_id_i.readObject();
$\mathrm{TJ}_{State}(st,st_i)[\rho]$;
exit = true;}

5.5.1.1.11 Final state

The final state of a transition (described by a term $con(exp_1, \ldots, exp_n)$) is translated into a sequence of assignments: one that sets the value of the `kind` field to the coding of the final state constructor (con), and the others that set the values of the fields that are meaningful for that value of `kind`, to the values described by $exp_1, \ldots, exp_n$. The final state is compared with the starting one to avoid "dummy" assignments and object creations if the value of a field is not changed.

$$\mathrm{TJ}_{State} : \textsf{State} \times \textsf{State} \to (\mathrm{CEnv} \to \mathrm{Java})$$

- $\mathrm{TJ}_{State}(st,\ st) =$;

- $\mathrm{TJ}_{State}(con(exp_1;\ \ldots\ ;\ exp_n),\ con(exp_1{}';\ \ldots\ ;\ exp_n{}')) = \begin{array}{l} fieldass_1 \\ \ldots \\ fieldass_n \end{array}$

  where
  $\textsf{MeaningfulFields}(con,\ \textsf{CurrentClass}(\rho),\ \rho) = (f_1, T_1);\ \ldots\ ;\ (f_n, T_n)$
  if $exp_1 \neq exp_1{}'$ then $fieldass_1 \equiv f_1$`=`$\mathrm{TJ}_{Exp}(exp_1{}',\ T_1)[\rho]$`; else;`
  ...
  if $exp_n \neq exp_n{}'$ then $fieldass_n \equiv f_n$`=`$\mathrm{TJ}_{Exp}(exp_n{}',\ T_n)[\rho]$`; else;`

- $\mathrm{TJ}_{State}(con'(exps),\ con(exp_1;\ \ldots\ ;\ exp_n)) = \begin{array}{l} \texttt{kind = }\textsf{Encoding}(con,\ \rho)\texttt{;} \\ f_1\texttt{= } \mathrm{TJ}_{Exp}(exp_1,\ T_1)[\rho]\texttt{;} \\ \ldots \\ f_n\texttt{= } \mathrm{TJ}_{Exp}(exp_n,\ T_n)[\rho]\texttt{;} \end{array}$

  where
  $con' \neq con$, $\textsf{MeaningfulFields}(con, \textsf{CurrentClass}(\rho), \rho) = (f_1, T_1);\ \ldots\ ;\ (f_n, T_n)$

Example The translation of the alternatives for the `KEYBOARD_DRIVER` from state Check(k,b) (see fig. C.3) is:

```
switch(kind) {

 ....

case 2 :{
  if (k = COMMAND.N)
   Newline.put();
  else
   if (k = COMMAND.P)
    {Newline.put();
      Print.put();
      Ok.get();
```

```
    }
  else
   if (KEY.IsDigit(k))
   kind = 3;
  else
   Pass.write(k);
   break;}
 case 3 : { ... }

   ..
}
```

where 2 e 3 are the values encoding the states Check and Taking.

The translation of the behaviour graph for the `DISPLAY_DRIVER` in `Calculator` is:

```
switch (kind) {
case 1 :{
  Show.get();
  kind = 2;
  break;
  }

case 2 :{
 dl = the_Display.Read();
 kind = 3;
 break;
 }

case 3 :{
 Ok.get();
 kind = 4;
 break
 }

case 4:{
 REFRESH.write();
 kind = 5;
 break;
 }
```

```
case 5: {
 if (dl = APPLY_LIST.Empty)
  {kind = 0;
    break:}
 else
  if(dl.head =!= null and dl.tail =!= null)
   {
    OUT.Write(dl.head);
    kind = 5;
    dl = dl.tail();
    break;
   }
 }

}
```

5.6 Architecture diagram

The architecture diagram supplies two kinds of information: the type and the number of the instances that implement the system components and the type and the number of the connectors. Thus, the information that the translation function deduces from the architecture diagram is:

- how many instances have to be created for each object and process class;
- which and how many instances of channel and connector classes have to be created;
- which processes share the same channel instances.

This results in the code of the classes implementing the channels, the connectors, the exchanged messages and the class containing the `main` method.

5.6.1 Architecture diagram environment

The environment used by the translation of the architecture diagram is a 5-upla: the two sets containing the identifiers of the object and process instances declared in the architecture diagram, the two sets containing the descriptions of the object and the communication cooperations and a set containing the identifiers of the stream and connectors instances that have to be created to implement the cooperations. In this last set we also put the information about the processes that must refer these instances.

The set of the architecture environment is denoted by AEnv and it is ranged over by the metavariable α.

Again, we define some operations to store/retrieve information in/from an architecture environment.

To add information

- $\mathsf{AddObjectCoops}$: $(\mathsf{Comp}_{\mathsf{id}} \times \mathsf{Comp}_{\mathsf{id}} \times \mathsf{Comp}_{\mathsf{id}} \times \mathrm{AEnv})^* \rightarrow \mathrm{AEnv}$

  add some object cooperations

- $\mathsf{AddNewInstances}$: $(\mathsf{Comp}_{\mathsf{id}} \times \mathsf{Type}_{\mathsf{id}} \times (\mathsf{Comp}_{\mathsf{id}})^* \times (\mathsf{Comp}_{\mathsf{id}} \times \mathsf{Comp}_{\mathsf{id}})^*)^* \times \mathrm{AEnv} \rightarrow \mathrm{AEnv}$

  adds a list of new instances implementing channels. For each instance it is added the identity, the type, the identities of the pipelined channels (if any) or of the external entity it is connected to and the list of process instances that use it, each one with the local name of the channel.

To retrieve information

- $\mathsf{ObjectRefs}$: $\mathsf{Comp}_{\mathsf{id}} \times \mathsf{Type}_{\mathsf{id}} \times \mathrm{AEnv} \rightarrow {\mathsf{Comp}_{\mathsf{id}}}^*$

  returns the list of the object references needed to create a process instance

- $\mathsf{AchanRefs}$ ($\mathsf{SchanRefs}$): $\mathsf{Comp}_{\mathsf{id}} \times \mathsf{Type}_{\mathsf{id}} \times \mathrm{AEnv} \rightarrow {\mathsf{Chan}_{\mathsf{id}}}^*$

  returns the list of the asynchronous (synchronous) channel references needed to create a process instance

- $\mathsf{ChanInstances}$: $\mathrm{AEnv} \rightarrow (\mathsf{Comp}_{\mathsf{id}} \times \mathsf{Type}_{\mathsf{id}} \times (\mathsf{Comp}_{\mathsf{id}} \cup\ ))^*$

  returns the list of the channel instances that have to be created, each one with the identity, the type and the identity of the pipelined channels (if any)

- $\mathsf{ConnInstances}$: $\mathrm{AEnv} \rightarrow (\mathsf{Comp}_{\mathsf{id}} \times \mathsf{Type}_{\mathsf{id}} \times (\mathsf{Comp}_{\mathsf{id}})^+)^*$

  returns the list of the connector instances that have to be created, each one with the identity, the type and the identity of the arguments.

5.6.1.0.12 Auxiliary functions

- CreateChannels: AEnv $\rightarrow$ Java

  reads in the architecture environment which are the channels instances to be created and returns the corresponding code

  CreateChannels(α) = CreateChannel($inst_1$) ... CreateChannel($inst_k$) ... CreateConnector($inst'_1$) ... CreateConnector($inst'_m$) ...

  if ChanInstances(α) = ($inst_1$; ... ; $inst_k$) and ConnInstances(α)= ($inst'_1$; ... ; $inst'_m$)

- Since channels must supply methods to read and write objects, a piped channel is implemented by an instance of `ObjectInputStream`(`ObjectOutputStream`) created over an instance of the `PipedInputStream`(`PipedOutputStream`) class.

  CreateChannel: $\mathsf{Comp_{id}} \times \mathsf{Type_{id}} \times (\mathsf{Comp_{id}} \cup\ ) \rightarrow$ Java

  - CreateChannel(id, `PipedInputStream`, id') =
    `PipedInputStream` aux_id = `new PipedInputStream(`id'`)` ;
    `ObjectInputStream` id= `new ObjectInputStream(`aux_id`)` ;
  - CreateChannel(id, `PipedOutputStream`, id') =
    `PipedOutputStream` aux_id = `new PipedOutputStream(`id'`)` ;
    `ObjectOutputStream` id= `new ObjectOutputStream(`aux_id`)` ;
  - CreateChannel(id, `InputStream`,) =
    `InputStream` aux_id = `new InputStream()` ;
    `ObjectInputStream` id= `new ObjectInputStream(`aux_id`)` ;
  - CreateChannel(id, `PipedOutputStream`,) =
    `OutputStream` aux_id = `new OutputStream()` ;
    `ObjectOutputStream` id= `new ObjectOutputStream(`aux_id`)` ;

- CreateConnector: $\mathsf{Comp_{id}} \times \mathsf{Type_{id}} \times \mathsf{Chan_{id}} \times \mathsf{Comp_{id}}^* \rightarrow$ Java

  CreateConnector(id, T, ach_id,(ach_id_1,id_1); ... ; (ach_id_n id_n))=
  Vector out_chans;
  CreateVector(out_Chans, (ach_id_1,id_1); ... ; (ach_id_n id_n));; T id= `new` T(ach_id, out_chans);
  id.`start()`;

CreateVector is a function returning the Java code that creates a vector whose elements are couples of channel reference and thread identifier (the identifier of the thread connected to the channel).

5.6.2 Architecture diagram

The environment structure used by the architecture diagram contains the information, collected by the translation of the class diagram, about the structure and the constructor methods of each process class.

Given an architecture diagram with cooperation list *coops* and instance list *comps*, first we consider *coops*; its translation returns some code plus the information about the nature and the number of channels that have to be created and passed to the process instances in the *comps*. The translation of the *comps* returns the code of the `main` method that is mainly devoted to create and initialize the system components.

Given the lists of the components and cooperations, their translation under the environment ρ is the code of the classes implementing the connectors, the synchronous channels (whereas the classes implementing the asynchronous channels are Java standard classes) plus the code of the class `Program` containing the `main` method.

$$\mathrm{TJ}_{Arch} : \mathsf{Arch\_dia} \to \mathrm{Java}$$

$$\mathrm{TJ}_{Arch}(comps,\ coops) = \begin{array}{l} c_1; \\ \texttt{class Program}\ \{ \\ \quad \texttt{public static void main(String argc[])}\ \{c_2\} \\ \} \end{array}$$

where
$(c_1,\alpha) = \mathrm{TJ}_{Coperations}(coops)$
$c_2 = \mathrm{TJ}_{Comps}(comps)[\alpha]$

5.6.3 Cooperations

The translation of the cooperations performs two task:

- collects the information (stored into the architecture environment) about which object or channel instances have to be passed to each process when created inside the main method
- returns the code for the auxiliary classes

$$\mathrm{TJ}_{Coperations} : \mathsf{Cooperations} \to (\mathrm{Java} \times \mathrm{AEnv})$$

$\mathrm{TJ}_{Coperations}(objcoops,\ Acoops,\ scoops) = (c_1\ c_2,\ \alpha_2)$
where
$\alpha = \mathrm{TJ}_{ObjCoops}(objcoops)$
$(c_1,\alpha_1) = \mathrm{TJ}_{AsynCoops}(Acoops)[\alpha]$
$(c_2,\alpha_2) = \mathrm{TJ}_{SynCoops}(scoops)[\alpha_1]$

5.6.3.0.13 Object Cooperations

The translation of an asynchronous cooperation between an object *oid* and a process *pid* does not produce code but it modifies the environment, recording that *pid* needs a reference to *oid*. This information will be used during the translation of the component lists, that creates the process instances. Let us recall that the list of the methods of an object used by a process is ignored in the translation.

$$\mathrm{TJ}_{ObjCoops}\colon \mathsf{ObjectCoop}^* \rightarrow \mathrm{AEnv}$$

$\mathrm{TJ}_{ObjCoops}((oid_1,pid_1,ln_1,mets_1);\ \ldots\ ;\ (oid_n,pid_n,ln_n\,mets_n)) = \alpha_1$
where
α_0 is the empty initial architecture environment
$\alpha_1 = \mathsf{AddObjectCoops}((pid_1,ln_1,oid_1);\ \ldots\ ;\ (pid_n,ln_n,oid_n),\alpha_0)$

5.6.3.0.14 Asynchronous cooperations

The cooperations among several processes that communicate in an asynchronous way are represented by particular icons called connectors.

The `java.io` package supplies a wide range of specialized classes that vary the form of the transmitted data as well as the type of the resource at the end of the stream. So we use a different stream class depending on the kind of resource we want to communicate with.

An asynchronous channel can be used to connect threads inside the same program; in the more general case, a data can be written by one or more threads and read by other threads.

Java supplies the class `PipedInputStream` and `PipedOutputStream` for pipelined communications among threads.

Since a read operation consumes the data, a broadcast communication requires to reply the message; this operation is performed by a particular entity, acting as a *connector*.

The `Thread` subclass `BConnector` implementing the broadcast connector has a field referencing an instance of `PipedInputStream`, a field referencing an array of instances of

`PipedOutputStream` and a field referencing the message. The method `run` is just an infinite loop, that continuously checks if there is a message to be read and replicates it on the output channels.

Let us point out that the thread classes produced by the translation of the process classes have, for each asynchronous channel, a field of a generic type `InputStream` or`OutputStream`. Thus, the main method, when creating the processes, can pass them only instances of subclasses of `InputStream` or`OutputStream`.

The implementation of the asynchronous named communication requires pipelined streams and a connector again. The threads acting as senders share an output stream that manages objects having two fields: one containing a reference to a `Thread` and one containing the message. Such stream is pipelined with an input stream of the connector that reads every message and put it into the pipelined input stream of the corresponding receiver thread.

Every receiver process has an input stream that manages messages and references to threads (denoting the senders of the messages).

In `JTN` we can specify the particular type of the data exchanged on a communication channel but we do not have to implement the channels by classes specialized for a particular message type, as we assume that the `JTN` specification to translate is correct. Furthermore, on this stream we can wrap a more specialized stream to read-write particular datatypes (bytes, characters, strings, objects, ...). Two object stream classes, `ObjectInputStream` and `ObjectOutputStream`, are specialized to read and to write objects using *object serialization*; these library classes do not make any check on the type of the data sent down the channel.

Thus, a channel of type T is implemented by an instance of a Java standard class as `ObjectInputStream` or the `ObjectOutputStream` constructed over instances of `InputStream`, `OutputStream` or `PipedInputStream` and `PipedOutputStream` (in case of object exchanged among processes). Explicit casting operations on the result of a `read` operation is required to receive an object of the correct type.

The implementation of a connector for asynchronous communication strictly depends on how this channel is used to communicate, that is we need to know which are the other parties of the communication, whether the message is sent to any, exactly one or all the processes using the connector: in other words, we need to know which is the kind of connections among this channel and the other entities. This information is contained in the architecture diagram, this is the reason because we cannot declare all the classes of the Java program in the translation of the class diagram and because we can create the process instances only when we exactly know which channel instances are referred by the process fields.

Thus, the translation of the asynchronous cooperations returns the code of the connector classes `BConnector` and `NConnector` plus the new environment resulting by the translation of each one of the asynchronous cooperations.

We can distinguish some cases, each one returning a new environment recording the instances that have to be created and the references that have to be passed to each process participating in the cooperation.

In the following, we describe the content of the new environment for each different case, graphically depicted in fig. 2.11. Let us suppose that the ach_id_i and $ach\_id'_i$ are the local names of the channels and that `EXT` denotes an external entity.

Simple communication The connector is an input or output stream class, where the data are enqueued following the order of the writing operations and the read operations consume the data. Notice that there is no need to implement the connector by a process. We can identify three sub-cases, depending on the components using the asynchronous connector:

- **(1a)** α records that all the fields implementing the $OC_1 \ldots OC_n$ refer the same instance of the output stream class `ObjectOutputStream`;
- **(1b)** α records that all the fields implementing the $IC_1 \ldots IC_m$ refer the same instance of the input stream class `ObjectInputStream` connected with the external world;
- **(1c)** α records that the fields implementing the $OC_1 \ldots OC_n$ refer the same instance os of of the piped output stream class `PipedOutputStream`, the fields implementing the $IC_1 \ldots IC_m$ refer the same instance is of the piped input stream class `PipedInputStream`; and that is and os are connected.

Broadcast communication The connector is implemented by a special thread class that reads a message from a piped input stream object and writes it over all the m piped output stream.

- **(2a)** α records that the fields implementing the $OC_1 \ldots OC_n$ refer the same instance os of `PipedOutputStream`, whereas the fields implementing the $IC_1 \ldots IC_m$ refer m different instances is_i of `PipedInputStream`, and that an instance bc of the broadcast connector class `BConnector` is connected with os and all the is_i.
- **(2b)** For each message read on the input channel, an instance of `BConnector` sends m copies on its piped output streams connected to the input streams of the processes.

  α records that the fields implementing the $IC_1 \ldots IC_m$ refer different instances $is_1 \ldots is'$of the class `PipedInputStream` and that an instance of the broadcast connector class `BConnector` is connected with all the is_i and with an instance of the input stream class `ObjectInputStream` (used by the connector to read from the external world);

Named communication Named communication is used to send a message to a particular process, or to receive a message with the identity of the sender, thus the connector cannot be used by the external world.

The connector is implemented by an instance of class `NConnector` that reads a triple (*sender, receiver, message*) from the input channels connected to the $IC_1 \ldots IC_m$ and writes a couple (*sender, message*) into the input channel of the process denoted by *receiver*.

α records that the fields implementing $OC_1 \ldots OC_n$refer the same instance of `PipedOutputStream` and that the fields implementing $IC_1 \ldots IC_m$refer m different instances of `PipedInputStream`.

The translation of the asynchronous connectors returns the code for the non-standard Java classes for the connectors described above (and precisely defined in Sect. 5.6.5) and the new class environment resulting from the translation of each one of the cooperations.

$$\mathrm{TJ}_{AsynCoops} : \mathsf{AsynhcCoop}^* \rightarrow (\mathrm{AEnv} \rightarrow (\mathrm{Java} \times \mathrm{AEnv}))$$

$\mathrm{TJ}_{AsynCoops}(Acoop_1; \ldots ; Acoop_n)[\alpha] = (\texttt{AIOCode}, \alpha_n)$
where
$\alpha_1 = \mathrm{TJ}_{AsynCoop}(Acoop_1)[\alpha]$
$\ldots$
$\alpha_n = \mathrm{TJ}_{AsynCoop}(Acoop_n)[\alpha_{n-1}]$

5.6.3.0.15 Asynchronous cooperation

On its turn, the translation of an asynchronous connector distinguishes among connectors for simple, broadcast and named communication:

$$\mathrm{TJ}_{AsynCoop} : \mathsf{AsynhcCoop} \rightarrow (\mathrm{AEnv} \rightarrow \mathrm{AEnv})$$

$\mathrm{TJ}_{AsynCoop}(id, T, \texttt{s}, comps_1, comps_2)[\alpha] = \mathrm{TJ}_{SAsynchCoop}(id, T, \texttt{s}, comps_1, comps_2)[\alpha]$

$\mathrm{TJ}_{AsynCoop}(id, T, \texttt{b}, comps_1, comps_2)[\alpha] = \mathrm{TJ}_{BAsynchCoop}(id, T, \texttt{b}, comps_1, comps_2)[\alpha]$

$\mathrm{TJ}_{AsynCoop}(id, T, \texttt{n}, comps_1, comps_2)[\alpha] = \mathrm{TJ}_{NAsynchCoop}(id, T, \texttt{n}, comps_1, comps_2)[\alpha]$

5.6.3.0.16 Simple asynchronous communication

$$\mathrm{TJ}_{SAsynchCoop}: \mathsf{AsynhcCoop} \to (\mathrm{AEnv} \to \mathrm{AEnv})$$

- *The external world sends messages to the processes of the cooperation. The returned environment records that there is an instance of* `ObjectInputStream` *shared by all the processes.*
  $\mathrm{TJ}_{SAsynchCoop}(id, T, \mathtt{s}, (\mathtt{EXT}, extid), (pid_1, ln_1); \ldots ; (pid_n, ln_n))[\alpha] = \alpha_1$
  where
  $\alpha_1 = \mathsf{AddNewInstances}(id, \mathtt{ObjectInputStream}, (pid_1, ln_1); \ldots ; (pid_n, ln_n), \alpha)$

- *Analogous to the previous case, if the processes send messages to the external world.*
  $\mathrm{TJ}_{SAsynchCoop}(id, T, \mathtt{s}, (pid_1, ln_1); \ldots ; (pid_n, ln_n), (\mathtt{EXT}, extid))[\alpha] = \alpha_1$
  where
  $\alpha_1 = \mathsf{AddNewInstances}((id, \mathtt{ObjectOutputStream}, extid, (pid_1, ln_1); \ldots ; (pid_n, ln_n)), \alpha)$

- *Two groups of processes cooperate; the processes in $comps_1$ act as senders, the ones in $comps_2$ act as receivers. The translation returns the environment recording that there is an instance of* `PipedOutputStream` *shared by the processes in $comps_1$ and an instance of* `PipedInputStream` *shared by the processes in $comps_2$ that have to be connected in a pipeline.*
  $\mathrm{TJ}_{SAsynchCoop}(id, T, \mathtt{s}, comps_1, comps_2)[\alpha] = \alpha_1$
  where
  $comps_1 = (pid_1, ln_1); \ldots ; (pid_n, ln_n)$
  $comps_2 = (pid_1', ln_1'); \ldots ; (pid_m', ln_m')$
  $\alpha_1 = \mathsf{AddNewInstances}((id_{IN}, \mathtt{PipedInputStream}, (pid_1, ln_1); \ldots ; (pid_n, ln_n));$
  $(id_{OUT}, \mathtt{PipedOutputStream}(id_{IN}), (pid_1', ln_1'); \ldots ; (pid_m', ln_m')), \alpha)$

5.6.3.0.17 Cooperations for broadcast communications

$$\mathrm{TJ}_{BAsynchCoop}: \mathsf{AsynhcCoop} \to (\mathrm{AEnv} \to \mathrm{AEnv})$$

- *The translation returns the architecture environment recording that the connector reads from the input stream connected to an external entity and sends the message copies over n output streams that are the input streams $is_1, \ldots, is_n$ of the receiving threads.*

$\mathrm{TJ}_{BAsynchCoop}(conn, T,\mathsf{b},(\mathtt{EXT}, extid),(pid_1,ln_1);\ \ldots\ ;\ (pid_n,ln_n))[\alpha]=\alpha_2$
where
$\alpha_1=$ $\mathsf{AddNewInstances}((is,\ \mathtt{ObjectInputStream},\Lambda),$
$;\ \ldots\ ;\ (is_1,\mathtt{PipedInputStream},(pid_1,ln_1));\ \ldots\ ;$
$(is_n,\mathtt{PipedInputStream},(pid_n,ln_n)),\ \alpha)$
$\alpha_2=$ $\mathsf{AddConnector}((conn,\ \mathtt{BConnector}, is,(pid_1,is_1);\ \ldots\ ;\ (pid_n,is_n),\ \alpha_1)$

- *If the broadcast communication involves two groups of processes, the translation is analogous, but the connector instance connects the output stream instances os shared by the k sending processes to the input streams of the receivers.*
  $\mathrm{TJ}_{BAsynchCoop}(conn, T,\mathsf{b},comps_1,comps_2)[\alpha]=\alpha_2$
  where
  $comps_1 = (pid_1,ln_1);\ \ldots\ ;\ (pid_n,ln_n),$
  $comps_2 = (pid_1',ln_1');\ \ldots\ ;\ (pid_m',ln_m'),$
  $\alpha_1 =$ $\mathsf{AddNewInstances}((os,\mathtt{PipedOutputStream},(pid_1,ln_1);\ \ldots\ ;\ (pid_n,ln_n));$
  $(os_1,\mathtt{PipedInputStream}(os),(pid_1',ln_1'))\ldots,$
  $(os_m,\mathtt{PipedOutputStream}(os),(pid_m',ln_m'));,\alpha)$
  $\alpha_2=$ $\mathsf{AddConnector}(conn,\ \mathtt{BConnector}, os;(pid_1',os_1)\ \ldots(pid_m' os_m),\ \alpha_1)$

5.6.3.0.18 Cooperations for named communications

$$\mathrm{TJ}_{NAsynchCoop} : \mathsf{AsynhcCoop} \rightarrow (\mathrm{AEnv} \rightarrow \mathrm{AEnv})$$

The translation is analogous to the previous case, except that the Java classes support operations to manage the names of the senders and of the receivers of the messages.
$\mathrm{TJ}_{NAsynchCoop}(conn, T,\mathsf{n},comps_1,comps_2)[\alpha]=\alpha_2$
where
$comps_1 = (pid_1,ln_1);\ \ldots\ ;\ (pid_n,ln_n),$
$comps_2 = (pid_1',ln_1');\ \ldots\ ;\ (pid_m',ln_m'),$
$\alpha_1=$ $\mathsf{AddNewInstances}((os,\ \mathtt{PipedOutputStream},(pid_1,ln_1);\ \ldots\ ;\ (pid_n,ln_n));$
$(is_1,\mathtt{PipedInputStream}(os),(pid_1',ln_1'))\ldots(is_m,\mathtt{PipedInputStream}(os),(pid_m',ln_m')),\alpha)$
$\alpha_2=$ $\mathsf{AddConnector}(conn,\ \mathtt{NConnector}(os,\ is_1;\ \ldots\ ;\ is_m))$

5.6.3.0.19 Synchronous cooperations

The synchronous communication is realized by rendez-vous channels that connect two threads at a time: one acting as a sender and one acting as a receiver. These channels are not buffered and allow to pass just one data at a time.

Cardinality	Signal exchange	Typed Message Exchange
One to One	`One_Sig_Chan`	`One_Msg_Chan`
Many to Many	`Many_Sig_Chan`	`Many_Msg_Chan`

Table 5.1: Classes implementing synchronous channels

When a thread performs a blocking read or write operation, it calls the corresponding method of the object implementing the channel. This method checks if the other partner is already waiting to complete the communication. If it so, it is notified (remember that there is at most one process waiting to complete the communication) and the method returns. Otherwise, the calling process becomes waiting to complete the communication.

The access to the channel methods is synchronized to ensure that no write action can be started while some thread is trying to read and vice versa. In the case of a channel used by many threads, an additional synchronization is required, to avoid that a reader tries to read on a channel while, for example, another one is suspended on the same channels, waiting for a message. The overhead due to the additional synchronization mechanisms is quite expensive, thus we distinguish between a class that implements a channel for one-to-one communication and a class for a channel used by many readers and many writers.

The most generic class is `Channel` that allows to exchange generic `Object` instances. In Table 5.1 we summarize the Java `Channel` sub-classes implementing the various kinds of synchronous channels (the complete code is in 5.6.5).

For the one-to-one communication: `One_Msg_Chan` implements rendez-vous channels similar to the ones used in CCS. If the threads need just to synchronize, without exchanging data, the channel is implemented by a `One_Sig_Chan` instance.

`Many_Sig_Chan` and `Many_Msg_Chan` are the equivalent classes that implement channels used by many users at the same time.

The analysis of the synchronous connectors determines which kind of channels correctly translate the cooperation. We can distinguish some cases described in fig. 2.12, depending on the number of involved processes and on the presence of messages in the communication. Their translation returns some code (the code of the classes `One_Msg_Chan` etc..) plus the environment resulting from the translation of the synchronous cooperations.

The translation of a single synchronous cooperation:

$(id, T, (pid_1, ln_1) \ldots (pid_n, ln_n), (pid_1', ln_1') \ldots (pid_m', ln_m'))$

modifies the environment as follows:

(b1) if $n = 1$, $m = 1$ and $T \neq$ `void`

α_1 records that all the ln_i, ln_i' refer the same instance of `One_Msg_Chan`;

(b2) if $n = 1$, $m = 1$ and T= `void`:

α_1 records that ln_i, ln_i' refer the same instance of `One_Sig_Chan`;

(b3) if $n > 1$, $m > 1$ and T≠`void`:

α_1 records that ln_i, ln_i' refer the same instance of `Many_Msg_Chan`;

(b4) if $n > 1$, $m > 1$ and T= `void`:

α_1 records that ln_i, ln_i' refer the same instance of `Many_Sig_Chan`.

$$\mathrm{TJ}_{SynCoops} : \mathsf{SynchCoop}^* \to \mathrm{AEnv} \to (\mathrm{Java} \times \mathrm{AEnv}))$$

The translation of the list of the synchronous cooperations returns the code for the classes described above (`SIOCode`*) plus the new environment resulting from the translation of each cooperation.*

$\mathrm{TJ}_{SynCoops}(scoop_1; \ldots ; scoop_n)[\alpha]$= (`SIOCode`, α_n)

where

α_1= $\mathrm{TJ}_{SynCoop}(scoop_1)[\alpha]$

...

α_n= $\mathrm{TJ}_{SynCoop}(scoop_n)[\alpha_{n-1}]$

5.6.3.0.20 Synchronous cooperation

$$\mathrm{TJ}_{SynCoop} : \mathsf{SynchCoop} \to \mathrm{AEnv} \to \mathrm{AEnv}$$

- *Two processes communicate by message exchange in a synchronized way. The returned environment records that there is an instance of* `One_Msg_Chan` *shared by the two processes.*

  $\mathrm{TJ}_{SynCoop}(id, T, (pid_1, ln_1), (pid_2, ln_2))[\alpha]$= α_1

  where

  α_1= $\mathsf{AddNewInstances}((id,$`One_Msg_Chan`$,(pid_1,ln_1);(pid_2,ln_2)),\alpha)$

- *Several processes communicate by message exchange in a synchronized way. The returned environment records that there is an instance of* `Many_Msg_Chan` *shared by all the processes.*

  $\mathrm{TJ}_{SynCoop}((id, T, (pid_1, ln_1) \ldots (pid_n, ln_n), (pid_1', ln_1') \ldots (pid_m', ln_m')))[\alpha]$= α_1

  where

  α_1= $\mathsf{AddNewInstances}((id,$`Many_Msg_Chan`$,(pid_1,ln_1) \ldots (pid_m',ln_m')),\alpha)$

- *Two processes communicate by signal exchange in a synchronized way. The returned environment records that there is an instance of* `One_Sig_Chan` *shared by the two processes.*
  $\mathrm{TJ}_{SynCoop}(id, \texttt{void}, (pid_1, ln_1), (pid_2, ln_2))[\alpha] = \alpha_1$
  where
  $\alpha_1 = \mathsf{AddNewInstances}((id, \texttt{One\_Sig\_Chan}, (pid_1, ln_1); (pid_2, ln_2)), \alpha)$

- *Several processes communicate by signal exchange in a synchronized way. The returned environment records that there is an instance of* `Many_Sig_Chan` *shared by all the processes.*
  $\mathrm{TJ}_{SynCoop}((id, \texttt{void}, (pid_1, ln_1) \ldots (pid_n, ln_n), (pid_1{}', ln_1{}') \ldots (pid_m{}', ln_m{}')))[\alpha] = \alpha_1$
  where
  $\alpha_1 = \mathsf{AddNewInstances}((id, \texttt{Many\_Sig\_Chan}, (pid_1, ln_1) \ldots (pid_m{}', ln_m{}')), \alpha)$

Example The code for the `main` method of the `Calculator` is:

```
public class Main {
 public static void main(String args[]){
  One_Sig_Chan Show = new One_Sig_Chan();
  One_Sig_Chan Pass = new One_Sig_Chan();
  One_Sig_Chan Print = new One_Sig_Chan();
  One_Sig_Chan NewLine = new One_Sig_Chan();

  Many_Sig_Chan Ok = new Many_Sig_Chan();

  ObjectOutputStream OUT = new ObjectOutputStream;
  ObjectOutputStream OUT_P = new ObjectOutputStream;
  ObjectOutputStream NEW_LINE = new ObjectOutputStream;
  ObjectOutputStream REFRESH = new ObjectOutputStream;

  ObjectInputStream IN = new ObjectInputStream;

  Display d = new Display();

  DisplayDriver dd = new DisplayDriver(Show,Ok,OUT, REFRESH,d);
  KeyboardDriver kd = new KeyboardDriver(Ok, Pass, NewLine, Print, IN, d);
  ComputingUnit cu = new ComputingUnit(Show,Pass,Ok,d);
  PrinterDriver pd = new PrinterDriver(Ok, NewLine,Print, OUT_P, NEW_LINE,d)
```

```
  dd.start();
  kd.start();
  cu.start();
  pd.start();
  }
}
```

5.6.4 Components

The translation of the component list creates the channels, the object and the process instances. The information about which objects and processes have to be created is contained in the environment; for simplicity, we assume to have ordered the components, so that all the objects precede the processes. and then the list of the component has the general form:

$(\mathtt{ob},\ oid_1, T_1{}^O);\ \ldots\ ;\ (\mathtt{ob},\ oid_k, T_k{}^O);(\mathtt{pr},\ pid_1, T_1{}^I);\ \ldots\ ;\ (\mathtt{pr},\ pid_n, T_n{}^I)$

Its translation returns a sequence of objects and processes and does not modify the environment. The call of a process constructor method contains the references to the objects and the channels used by the process. Every process instance is started immediately after the creation.

$$\mathrm{TJ}_{Comps}\colon \mathsf{Component}^+ \rightarrow ((\mathrm{CEnv} \times \mathrm{AEnv}) \rightarrow \mathrm{Java})$$

$$\mathrm{TJ}_{Comps}(comp_1;\ \ldots\ ;\ comp_n)[\alpha] = \begin{array}{l} \mathsf{CreateChannels}(\alpha) \\ \mathrm{TJ}_{Comp}(comp_1)[\alpha] \\ \ldots \\ \mathrm{TJ}_{Comp}(comp_n)[\alpha] \end{array}$$

5.6.4.0.21 Component

$$\mathrm{TJ}_{Comp}\colon \mathsf{Component} \rightarrow \mathrm{AEnv} \rightarrow \mathrm{Java}$$

- $\mathrm{TJ}_{Comp}(\mathtt{ob},\ oid, T)[\alpha] = oid$ `= new` T`();`

- $\mathrm{TJ}_{Comp}(\mathtt{pr},\ pid, T)[\alpha] = pid$ `= new` $T(oids, ach\_ids, sch\_ids)$`;` pid`.start();`
  where
  $\mathsf{ObjectRefs}(pid, T, \alpha) = oids$
  $\mathsf{AchanRefs}(pid, T, \alpha) = ach\_ids$
  $\mathsf{SchanRefs}(pid, T, \alpha) = sch\_ids$

5.6.5 Code of the predefined Java classes

In the following we give the code of the following non standard Java classes used to translate synchronous and asynchronous communications:

- `BConnector`
- `NConnector`
- `One_Sig_Chan`
- `One_Msg_Chan`
- `Many_Sig_Chan`
- `Many_Msg_Chan`

5.6.5.0.22 BConnector The class `BConnector` is a thread subclass with a field referencing an instance of `PipedInputStream`, a field referencing an array of instances of `PipedOutputStream`The method `run` is just an infinite loop, that continuously checks if there is a message to be read and replicates it on the output channels.

```
class BConnector extends Thread{

 private ObjectInputStream inChan;
 private Vector outChanVect = new Vector();

 BConnector(PipedOutputStream process_out, Vector process_in){

  PipedInputStream p_in = PipedInputStream(process_out);
  inChan = new ObjectInputStream(p_in);

  int size = process_in.size();

  for (i=0;i<size; i++){
    PipedOutputStream p_out = PipedOutputStream(process_in.elementAt(i));
    outChan = new ObjectOutputStream(p_out);
    outChanVect.addElement(outChan);
   }
```

```
 }
 public static void run() {
  Object mess;
  while(true){
   mess = inChan.readObject();
   for (i=0;i<size; i++){
    outChanVect.elementAt(i).writeObject(mess);
   }
  }
 }
}
```

5.6.5.0.23 NamedMess

The class `NamedMess` implements messages (instances of the class Object) enriched with the identities of the sender and the receiver.

```
class NamedMess{
 private Object the_message;
 private Thread the_sender;
 private Thread the_receiver

 NamedMess(Thread sender, Thread receiver, Object mess){
  super();
  the_sender = sender;
  the_receiver = receiver;
  the_message = message;
  }

 public Thread Sender(){return the_sender;}
 public Thread Receiver(){return the_receiver;}
 public Object Message(){return the_message;}

}
```

5.6.5.0.24 NConnector

The class `NConnector` is a thread subclass with a field referencing an instance of `ObjectInputStream` and a field referencing a vector of instances of `PipedOutputStream`The method `run` is just an infinite loop, that continuously checks if there is a message to be read and replicates it on the output channels.

```
class NConnector extends Thread{

 private ObjectInputStream inChan;
 private Vector outChanVect = new Vector();

 NConnector(PipedOutputProcess process_out, Vector process_in){

  PipedInputStream p_in = PipedInputStream(process_out);
  inChan = new ObjectInputStream(p_in);

  int size = process_in.size();
  for (i=0;i<size; i++){
    PipedOutputStream p_out = PipedOutputStream(process_in.elementAt(i));
    outChan = new ObjectOutputStream(p_out);
    outChanVect.addElement(outChan);
   }
 }

 public static void run() {
  Object mess;
  while(true){
   for (i=0; i<in_size; i++){
   if (inChan.available() >0)
    mess = (NamedMess)inChan.readObject();
    String sender = mess.sender();
    String receiver = mess.receiver();
    for (j=0; j<out_size; j++){
     if (outChanVect.elementAt(j).Identity() = receiver)
       outChanVect.elementAt(j).writeObject(mess);
    }
   }
  }
 }
}
```

5.6.5.0.25 One_Sig_Chan

The class One_Sig_Chan implements a synchronous channel shared by only two process to exchange signals. It supplies a method (`synchAvailable`) for checking if a synchronization can immediately be completed, plus a method (`synch`) for the blocking version of the synchronization.

```
class One_Sig_Chan{

 public boolean sign_ready = false;

 public boolean synchAvailable(){ // non-blocking get operation
   synchronized(this){
    if (sign_ready) {   // second to the rendez-vous;
      notify();  // wake-up the waiting writer
      sign_ready = false;
      return true;
    }
    else return false
  }
 }


 public void synch(){ // blocking get operation
   synchronized(this){         // synchronization among reader and writer
    if (!sign_ready) {   // first to the rendez-vous;
     try{
      sign_ready = true;
      wait();
     }
     catch(InterruptedException e){
     }
    else{ sign_ready = false; //second to the rendez-vous
     notify();  // wake-up the waiting writer
    }
   }
  }
 }


}// end of One_Sig_Chan class
```

5.6.5.0.26 One_Msg_Chan

Equivalent to the previous case, but the read and write operation allow to exchange messages.

```
public class One_Msg_Chan{

 Object msg;

 boolean ready = false;

 Object reader_monitor = new Object();
 Object writer_monitor = new Object();

 public Bool readAvailable(Object x){
   synchronized(this){
   // synchronization among reader and writer
    if ( msg_ready) {
    // second to the rendez-vous; can read the message an return immediately
     msg_ready = false; //second to the rendez-vous
     notify();  // wake-up the waiting writer
     x = msg;
     return(true);
    }
    else return(false);
   }
  }


 public void writeAvailable(){
   synchronized(this){
   // synchronization among reader and writer
    if (ready) {   // second to the rendez-vous;
     ready = false; //second to the rendez-vous
     notify();  // wake-up the waiting writer
     msg = the_msg;
     return(true);
    }
    else return(false);
   }
  }
```

```
 public Object read(){
   synchronized(this){         // synchronization among reader and writer
    if (! ready) {   // first to the rendez-vous;
     try{
      ready = true;
      wait();
     }
     catch(InterruptedException e){
      }
    else{ ready = false; //second to the rendez-vous
     notify();  // wake-up the waiting writer
    }
    return msg;
   }
  }
 }

 public void write(Object the_msg){
   synchronized(this){         // synchronization among reader and writer
    msg = the_msg;
    if (!ready) {   // first to the rendez-vous;
     try{
      ready = true;
      wait();
     }
     catch(InterruptedException e){}
    else{
     ready = false; //second to the rendez-vous
     notify();  // wake-up the waiting writer
    }
   }
  }
 }

}// end of One_Msg_Chan class
```

5.6.5.0.27 ManySigChan

The class Many_Sig_Chan implements a synchronous channel shared by several processes to exchange signals. It supplies one method (`synchAvailable`) for checking if a send/get operation can immediately be completed, plus one method (`synch`) for the blocking version of the synchronization.

```
class Many_Sig_Chan{

 public boolean sign_ready = false;

 Object reader_monitor = new Object();
 Object writer_monitor = new Object();

 public boolean synchAvailable(){
  synchronized(writer_monitor){
   synchronized(this){
    if (sign_ready) {   // second to the rendez-vous: it can immediately get;
      notify();  // wake-up the waiting writer
      sign_ready = false;
      return = true;
    }
    else return false;
   }
  }
 }


 public void synch(){
  synchronized(reader_monitor){ //synchronization among several readers
   synchronized(this){         // synchronization among reader and writer
    if (!sign_ready) {   // first to the rendez-vous;
     try{
      sign_ready = true;
      wait();
     }
     catch(InterruptedException e){
     }
    else{ sign_ready = false; //second to the rendez-vous
     notify();  // wake-up the waiting writer
    }
   }
  }
```

```
 }

}// end of Many_Sig_Chan class
```

5.6.5.0.28 Many_Msg_Chan

Analogous to the class above, but the methods allow to exchange messages.

```
public class Many_Msg_Chan{

 Object msg;

 boolean msg_ready = false;

 Object reader_monitor = new Object();
 Object writer_monitor = new Object();

 public Object readAvailable(){
  synchronized(reader_monitor){ //synchronization among several readers
   synchronized(this){
   // synchronization among reader and writer
    if ( msg_ready) {
    // second to the rendez-vous; can read the message an return immediately
     msg_ready = false; //second to the rendez-vous
     notify();  // wake-up the waiting writer
    }
    return msg;
   }
  }
 }

 public void writeAvailable(){
  synchronized(writer_monitor){ //synchronization among several writers
   synchronized(this){         // synchronization among reader and writer

    msg = the_msg;
    if (msg_ready) {   // second to the rendez-vous;
     msg_ready = false; //second to the rendez-vous
     notify();  // wake-up the waiting reader
```

```
   }
  }
 }
}


public Object read(){
 synchronized(reader_monitor){ //synchronization among several readers
  synchronized(this){         // synchronization among reader and writer
   if (! msg_ready) {   // first to the rendez-vous;
    try{
     wait();
    }
    catch(InterruptedException e){
     }
   else{ msg_ready = false; //second to the rendez-vous
    notify();  // wake-up the waiting writer
   }
   return msg;
  }
 }
}

public void write(Object the_msg){
 synchronized(writer_monitor){ //synchronization among several writers
  synchronized(this){         // synchronization among reader and writer

   msg = the_msg;
   if (!msg_ready) {   // first to the rendez-vous;
    try{
     msg_ready = true;
     wait();
    }
    catch(InterruptedException e){
     }
   else{
    msg_ready = false; //second to the rendez-vous
    notify();  // wake-up the waiting reader
   }
  }
 }
```

```
 }

}// end of Many_Msg_Chan class
```

5.7 JTN specification

The translation of a JTN specification returns a sequence of Java classes: those returned by the translation of the class diagram and those returned by the translation of the architecture diagram (the class with the main method, plus the classes of the connectors and the communication channels, not explicitly described within the class diagram). The execution diagrams do not supply any new information to the translation.

$$\mathrm{TJ} : \mathsf{JTN} \rightarrow \mathrm{Java}$$

$\mathrm{TJ}(\mathit{cldia},\ \mathit{archdia},\ \mathit{seqdias}) = \mathrm{TJ}_{\mathit{Class}}(\mathit{cldia})\ \mathrm{TJ}_{\mathit{Arch}}(\mathit{archdia})$

Chapter 6

Java semantics

6.1 Introduction

In this section we present a formal semantics of the subset of the Java language that we use to translate the `JTN` specifications.

There exist some works by other authors on formal semantics of some parts of the Java language but, for our purpose, we need to give our own semantics to the whole Java language. The reasons are essentially two:

- deeply understanding Java, even if, w.r.t. other languages, it is much more easy to use and has a clear informal description (see [J.G96]); in order to define the translation of a `JTN` specification into the correct Java program implementing it, we need a familiarity and deep understanding of the mechanisms and features of the language: the activity of giving a semantics to Java obviously leads us to correctly grasp its features;
- having a formal base for checking the correctness of the final step of the method, that transforms a design specification into Java code; this requires a model that can be easily compared with the model of the `JTN` specifications.

We are interested in a formal semantics covering the Java sublanguage that we use to translate the `JTN` specifications: *class* and *object* fields and methods, *concurrent threads*, *local variables* and *I-O operations*; for the moment we do not need *exception handling* and *Remote Method Invocation*, while, for simplicity, we skip the semantics of mathematical binary operators, control structures (`for, while, do, break`, pre and post-fix operators (`++, --`). For the same reasons, we are not interested in the semantics of applets neither of the Java security mechanisms.

Thus, we need a semantics that is simple and intuitive and gives a formal counterpart to the specification of the Java language described in [J.G96], by using the simplest mathematical theory and techniques: An operational semantics seems to be the correct way. Furthermore, we also describe the model of a Java program in a way very similar to the model associated with a `JTN` specification by the semantics described in chapter 4. Our semantics is given in the SOS style of Plotkin and Milner's CCS([Plo81, Mil80]).

6.1.1 Questionable properties of the Java programs

In [EG99] we have given an operational semantics to multi-threaded Java by following the informal specification given in [J.G96]. This work had pointed out some questionable properties of the Java programs (see [EG98]). Such properties concern the way concurrent threads can interact with shared objects. Java semantics allows the access to shared objects to occur in a very loose way and this sometimes has very strange effects and makes it impossible or, at least, very hard to reason about concurrent programs.

To give an idea of such properties, we want to stress that if two threads share an object and if one of them modifies the object state, the second one is not guaranteed to read the newly assigned value, even if it is forced to use the object after the first one had performed the assignment (Java gives the mechanism for coordinating the activity of concurrent threads, e.g., to force that a thread waits until another one has terminated).

So, we have restricted our interest to subclasses of Java programs having acceptable properties guaranteeing a reliable use of shared objects. It is emerged that such class is very meaningful and wide enough to cover the great part of the Java uses. We have formally characterized one of such classes, based on the idea that the mechanism ensuring such properties is the use of *synchronization*, that forces the updating of the thread working memory before using shared variables and the flushing of the assigned value into the main memory.

The good point, on the contrary, is that the restriction to the class of Java programs allows the use of a simpler operational semantics (called *abstract semantics*), and, consequently, simpler observations and equivalence notion. Finally, all the programs produced by the automatic translation of a `JTN` specification fall into this class: thus, in the following we will present only the *abstract semantics*.

6.1.2 Main memory and working memory

Java supports concurrent programming by means of threads that independently execute code. Such code operates on Java values and objects residing in a shared *main memory*. In order to make memory access more efficient, [J.G96] says that every thread owns a private *working memory* in which it keeps *working copies* of all the variables it uses or assigns.

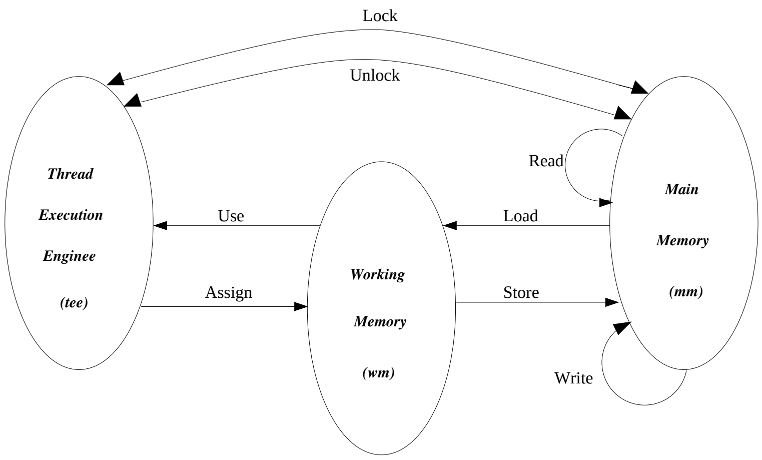


Figure 6.1: Working and main memory interaction

When a thread uses a local copy of a variable, it performs a sequence of actions, some of which, for example when it *assigns* or *uses* a value, directly operate on variables. These actions are *low level* actions that do not correspond to Java statements (in the following, *low-level actions* refers to this kind of actions). The *master copies* of variables are kept into *main memory*. Some rules (in [J.G96]) describe when a thread is permitted to update its own local copy of a variable.

Moreover, Java implements *monitors* that allow one single thread at a time to execute protected parts of code. For that purpose, for each object and class there exists a *lock* kept into the main memory. Every thread has to compete with the other ones in order to acquire a lock.

Different actions (see fig.6.1) are performed by the *thread execution engine*, by the *working memory* and by the *main memory* (just *tee*, *wm* and Mm, respectively, in the following).

A *tee* can `use` and `assign` a variable. A `use` transfers the content of a variable from the *wm* into the *tee*; an `assign` transfers a value of a variable from the *tee* into the *wm*; the *wm* can `load` a value from Mm or `store` it into the Mm, that, conversely, has the capability to send a value to the *wm* by a `read` action or to receive a value from *wm* by a `write` action. Every `read` by the Mm is followed some time later by a corresponding `load` performed by the *wm*. Every `store` performed by *wm* is followed some time later by the corresponding `read` performed by the Mm. Moreover, both *tee* and Mm can perform (in a tightly coupled way) `lock` and `unlock` actions.

6.1.3 A too loose semantics

From our point of view, Java semantics is too loose w.r.t. shared memory management. Constraints ruling the asynchronous interaction in [J.G96] between Mm and *wm* leave a certain degree of freedom to the implementor. If, from one side, this relaxation allows designers to improve speed and throughput, by the other side "strange" and undesired behaviors can come from the free interactions of threads on shared variables.

Such a situation is described in the following **Example 1** (presented in [PRM98]):

(t_1) $synchronized(p)\{p.y = 2;\}a = p.x; b = p.y; c = p.y;$

(t_2) $synchronized(p)\{p.y = 3; p.y = 100;\}p.x = 1;$

where t_1 and t_2 are the names of two threads executing the two pieces of code.

The key point, intuitively described by the example, is that from the moment a `read` operation for a value of an object field is performed by Mm, to the moment such value is actually loaded on the thread *wm* and used, some other thread can change it (thus, even if it has been loaded just before being used, it is not the most recent one).

When a value is loaded into the *wm*, it could have been read by the Mm in any moment within the past, so that it could not be the most recent one. This can sound normal in a usual concurrent programming context: if we do not ensure some ordering in the access to a field object, then it can be modified by any concurrent thread in any order and, in particular, it is possible that a thread reads a value after another one had assigned it but the Mm has not been already changed.

What sounds a bit more strange is that if we know the execution order of the reading and writing operations on an object field, i.e., we use a "naive" coordination mechanism, for example by imposing that a thread can start only after another one has terminated the assignment execution, we can have undesired results the same.

In the previous example, even if we force the thread t_1 to execute after the thread t_2 had terminated, t_1 may read the old value 3 for y, even if t_2 assigned 100 to y before terminating. This situation is represented in fig.6.2, depicting one execution of the whole program. A label a_i, where $i = \{1, 2\}$ describes the occurrence of action a performed by the thread i, while a_M describes the occurrence of action a performed by Mm. In fig.6.2, even if transition Load(y,3)$_1$ occurs after t_2 has terminated, the corresponding `read` occurrence Read(y,3,t1)$_M$ precedes Assign(y,100)$_2$.

What seems worthwhile noting, from our point of view, is that the asynchrony of communication between Mm and *wm* can cause "unexpected" and unpredictable behaviors that need to be avoided in truly concurrent programming.

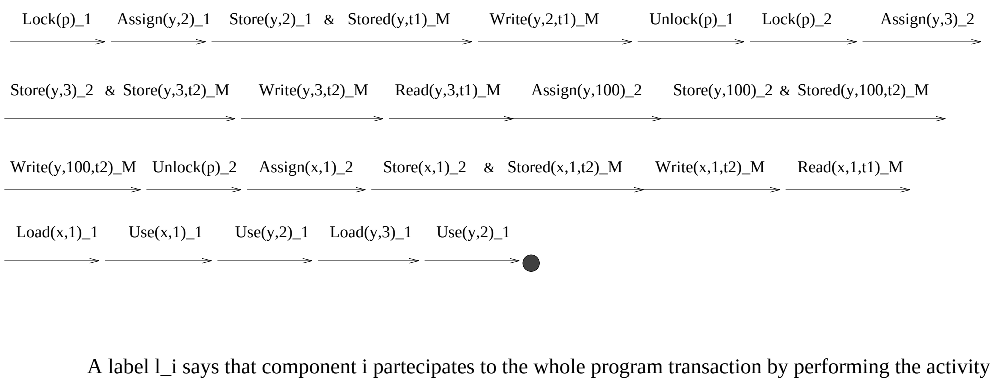


Figure 6.2: An execution for Example 1

If, in the previous example, t_1 assigns value 1 to a, and we are sure that t_1 executes after t_2 has completed its computation, then we reasonably expect that t_1 will assign to b and c the same value, since no other thread is modifying the value of y. From our point of view, it seems hard to predict that one possible computation assigns 100 to a but two different values to b and c.

Situations like the classical producer-consumer schema as well as the case when a variable is used for example as a counter, that can be updated and read by several components and must ensure that it always contains the correct values, are very frequent cases, where the "strange" executions can cause problems.

We aim to formally characterize a sensible subset of Java programs in which we are guaranteed to have banished computations like the one depicted in fig.6.2.

We describe the subset of good programs by means of a property, informally described below(see [EG99] for a formal presentation, using temporal logic). In the following o_{id}.`f`denotes a field of a shared object.

P) If sometime in the past a thread tc_1 assigned a value `v` *to o_{id}.*`f`*, no other thread assigned subsequently a value to o_{id}.*`f`*, tc_1 has terminated and another thread tc_2 reads o_{id}.*`f`*, then tc_2 reads* `v`*.*

Goods is defined to be the set of all Java programs satisfying the requirement expressed by P and, then, ensure a predictable use of the shared memory.

Java gives some synchronization mechanisms to avoid that threads interfere when accessing shared resources. So, the easiest solution to have completely reliable communications by using shared objects is to synchronize every access. Moreover, it is obvious that not all the shared variables have to be used in a synchronized block, since variables that are read only

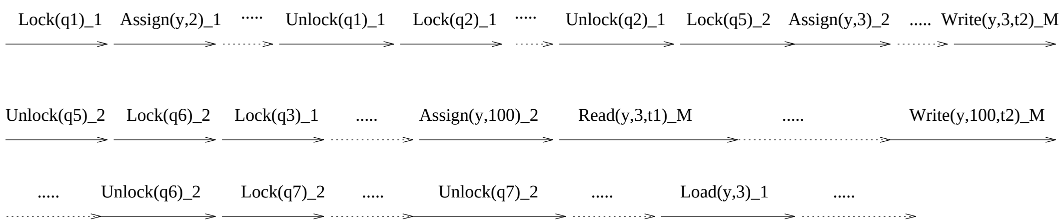


Figure 6.3: A computation for Ex.3 not satisfying *P1*

or write only by all threads sharing them can be accessed without restriction. The problem is only to guarantee that a thread reads correctly a variable that could be modified by some other thread.

We can observe that forcing any access to a shared variable to be synchronized on the same lock can be too strong a constraint.

If we only impose that an assign or a load operation on a variable referencing a shared object occurs inside a synchronized block, no matter which is the lock, we have a larger set $Goods_1 \supset Goods$, whose elements do not satisfies property *P1* but the less restrictive *P1'*. **Example 3** is:

(t_1) $\{p.y = 2;\}_{q1}\{a = p.x;\}_{q2}\{b = p.y;\}_{q3}\{c = p.y;\}_{q4}$

(t_2) $\{p.y = 3;\}_{q5}\{p.y = 100;\}_{q6}\{p.x = 1;\}_{q7}$

Fig.6.3 presents a computation of Ex.3 that does not satisfy *P1* since Read(y,3,t1)_M occurs after Assign(y,100)_2 but before Write(y,100,t2)_M.

Starting from these observations, the restriction that forces any access to a shared object to happen within a synchronized block, seems to be natural for having concurrent programs satisfying *P1*.

Example 2 below gives an example of a program in *Goods* (the set of programs satisfying *P1*) obtained by Ex.1 by extending the use of synchronization to all the accesses to shared variable y (we use the short notation $\{p.y = 2;\}_l$ for $synchronized(l)\{p.y = 2;\}$):

(t_1) $\{p.y = 2;\}_{ly}\{a = p.x;\}_{lx}\{b = p.y;\}_{ly}\{c = p.y;\}_{ly}$

(t_2) $\{p.y = 3;\}_{ly}\{p.y = 100;\}_{ly}\{p.x = 1;\}_{lx}$

Informally, the proof that the program satisfies *P1* is based on two considerations: an unlock operation forces any assignment occurring inside the synchronized block to be copied in the Mm and any read operation to occur after the locking operation.

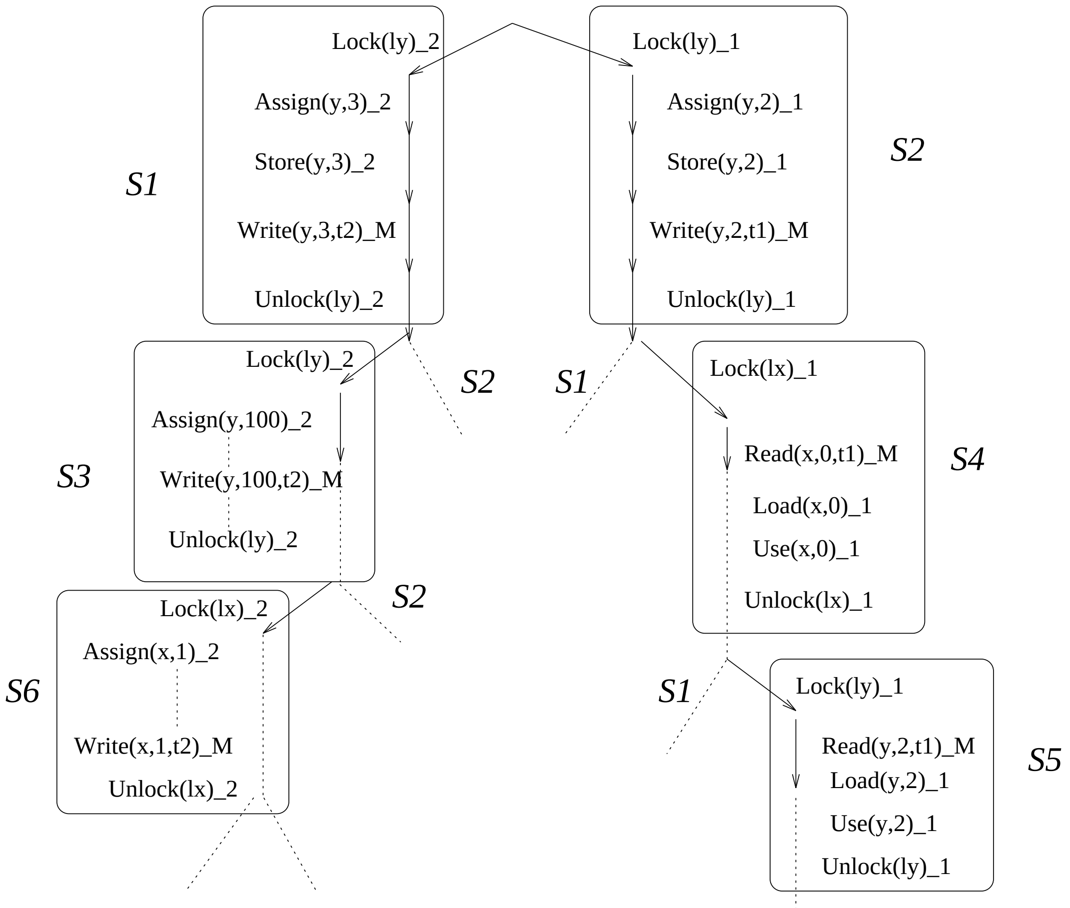


Figure 6.4: The execution model of Example 2

A more detailed reasoning is: y in $\{b = p.y;\}_{ly}$ is synchronized and if $\{b = p.y;\}_{ly}$ is performed after $\{p.y = 100;\}_{ly}$, then the reading of y for t_1 that is coupled with the loading action is forced to occur after the lock on l_y by t_1 and, consequently, after the unlock on l_y of t_2 and then, after the writing of value 100 for y in wm. In fig. 6.4 we can see that if an `assign` occurs inside a block labeled by Si, Si also contains the corresponding `write`. Analogously, if a `load` occurs inside a block labeled by Si, Si contains the corresponding `read`.

It is easy to see that all the programs that are returned by the translation of a `JTN` specification, as defined in Chap.5.1 fall in the *Goods* set. In fact threads can exchange values along channels only by calling read and write methods of standard or predefined classes: they are all synchronized methods.

Furthermore, the fields of the shared objects can be updated or read only by synchronized methods. The interaction of threads with Mm (described in Sect. 6.1.2) involves the modeling of another entity (the main memory manager, Mmng) that may asynchronously read a value for a thread from an object state and, later, send it to the *wm* of a thread or write a value received from the thread by modifying the corresponding object: it is not just a passive component of the program and needs to be modeled by an *lts*.

As we have seen, one of the most tricky aspects of Java semantics is to describe the interaction among threads, *wm* and Mm. What does it happen if we abstract from the distinction among a Mm and several *wm*s, if all the interactions involved by the reading or writing of a variable are performed synchronously, if an assigned value is assumed to be written immediately into Mm, if a variable is directly read from the Mm every time it is used? The semantics becomes fairly simple and easy to understand.

First of all, the program state has a less complex representation since it loses the Mmng component. Thread states lose working memories and there is no need of an lts that models the memory manager. Moreover, the number of transitions both for thread and program lts is reduced, since the low level actions on memories disappear.

These considerations moved us to define a new, more abstract semantics (see [ER99]), called *abstract* (*Abs*) and to compare it with the original *concrete* (*Con*) described in [EG99].

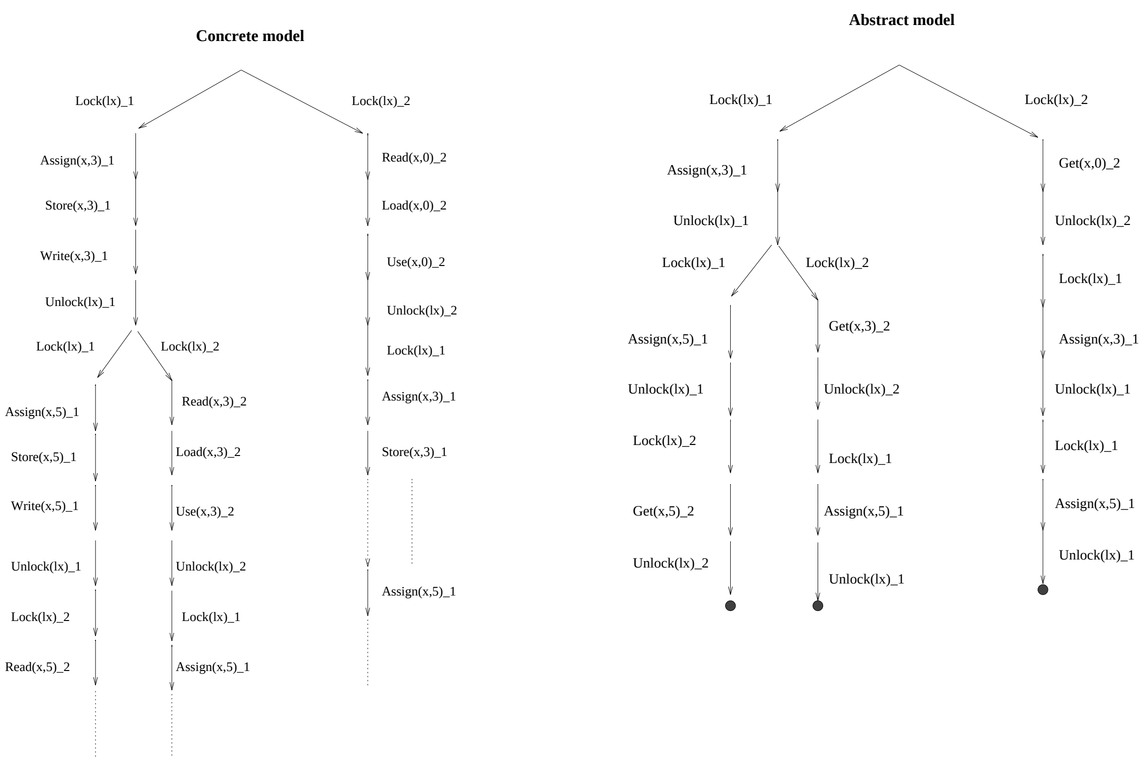
Fig.6.5 shows how much simpler is the model given by *Abs* w.r.t. the one given by *Con*.

6.1.3.1 The Model

In our approach, multi-threaded Java programs are modeled by a labelled transition system (*lts*, see sect.D); given the *lts*, we can associate with each initial state a *transition tree* that is a labelled tree whose nodes are decorated by states and whose edges by labels. Thus, an edge of the tree describes the execution of an atomic step in the computation, the branching of the tree models the non-determinism of the program. Inductive rules describe the transitions of the model and give the semantics of the language in a simple and intuitive way.

Since it is not possible to simply associate a result with a multi-threaded Java program, it is absolutely necessary to have a correct observational semantics and, consequently, an equivalence notion to test if two programs are equivalent.

We take advantage of the model defined by the operational semantics: as we will see in Chap.7, a labelled transition system not only gives an operational semantics, but also describes what information about a program execution is observable from outside. We have analysed the most important communication means implemented in Java; in sec. 6.7 we give a semantics of some of these mechanisms, such as communication with an external



Example 4: t1: $\{x = 3;\}_{lx}$ $\{x = 5;\}_{lx}$
t2: $\{a = x;\}_{lx}$

Figure 6.5: The concrete and the abstract models of a simple example

user by standard input-output, read/write operations over files, use of sockets, etc. The whole amount of data exchanged by these operations is observable from an external user, thus, it must be described by our semantics: the transitions modelling communication operations are labelled with the communicated data.

Once we have described how a model associated with a Java program gives us information about interactions with the external environment, the equivalence relation is intuitive: roughly speaking, two programs are equal if they allow the same observations.

Obviously, we need to have a semantics as simple as possible, that abstracts from some aspects of the language, like, for example, priorities associated with threads. Every time it seems reasonable and correct, we try to avoid the description of some language aspects that could make the presentation of the semantics more "tricky" and less easy to be understood.

Let us recall that the semantics is given having in mind the translation function and, thus, the subset of Java programs we are interested in: we will skip most of the Java features that are not used in these programs.

Anyway, we expect to enrich the `JTN` language and, consequently, to modify the translation function; this will require to enrich the Java subset used in the translation and to modify its semantics.

To minimize the impact of these changes on this semantics, we follow a modular approach. A good way, in our opinion, is to choose an initial subset of Java (called Java_1) that we consider interesting and useful enough and give it a semantics. It contains all the object-oriented Java features; the main restriction is that Java_1 programs are just programs without concurrency synchronization and I/O operations. When these other features are considered, they result in a richer sub-language (Java_2) whose semantics is given modifying as less as possible the preceding one. Java_2 introduces the mechanisms for thread synchronization. Finally, in Java_3, we also consider I/O operations using files and streams. We omit only the Java features that are peculiar to distributed programming, like Remote Method Invocation and, consequently, Object Serialization and mechanism for interaction by means of graphical interfaces, like event-handling.

6.1.3.1.1 A comparison with other semantics

If we compare our semantics for Java with the work of other authors, we can say that we essentially focus on the *dynamic* semantics of multi-threaded Java.

In the case of pure sequential programs and for the static part of the semantics, we just apply the idea given in [S.D97] and other works, even if here we extend the type system and the functions enriching syntactic terms with type information, as introduced in [S.D97], to cover the whole Java sub-languages we have chosen. For a more detailed comparison of our static semantics with the one in [S.D97], see sec.6.4.2.2.

Our pragmatic approach to Java semantics is also different from the one adopted in [PBM97, PRM98, EB98, Rus98]. In [PBM97, PRM98] the use of *event structures* allows the semantics to get rid of many details about the execution order of certain operations, while we chose to be as descriptive as possible. While in [PBM97, PRM98], the semantics of operations on working and main memories is given by axiomatizing the ordering rules given in [J.G96], we explicitly describe such interaction in a detailed way, subsequently proving (see Chap. 7) that such rules are satisfied.

The semantics given by using ASM in [EB98] covers a great part of Java language, also focusing on the possibility to distinguish a subset of Java programs whose semantics could abstract from the detailed description of memory interaction. However, again, in [EB98] the models proposed for the Java programs do not help in finding an equality notion.

Besides, in [PBM97, PRM98], operational semantics does not give any information about what can be observed from a program execution and, consequently, which could be a sensible notion of semantic equivalence between Java programs, that is essential from our point of view and that we are currently exploring (see [ER99]).

6.2 Static and dynamic semantics for Java

We give a semantics that describes the execution and gives a model of each Java program; we can call it the dynamic semantics of a Java program.

It is worthwhile to stress that our interest is not on the static semantics of Java, i.e., in a semantics that statically analyses a Java program and calculates information about the well-formedness and the types of the terms. However, as we will see in the following, a dynamic semantics for Java cannot be given without some information collected by a previous phase of static analysis.

But to develop a static semantics from scratch is behind the scope of our work, so we refer to [S.D97] that is mainly focused on the type semantics of Java. More precisely, we extend the sublanguage in [S.D97] with: local variables, class methods and fields, `super` expression and thread synchronization.

6.2.0.1.2 An operational, structural, observational approach to dynamic semantics

We need a semantics that is simple and intuitive and gives a formal counterpart to the Java specification in [J.G96], by using the simplest mathematical foundations and tools. The reason is that we want to give a formal description of Java that must be easily understood and discussed by many people, that is a formal base for proving the correctness of the coding step of the method and, perhaps, a formal support to the use of tools within the development method.

A Java program is a sequence of classes; each class consists of a declarative part, describing which is the superclass and which are its components (fields and methods), and a computational part describing the bodies of the methods. A particular class contains a special method (`main`) that is automatically executed when the program starts.

Our interest is on the semantics of the execution of this method, so we separate the computational part (consisting of the main method and the other method bodies) from the declarative part of the program (consisting of declarations of classes, fields, methods and variables) containing static information that cannot be modified during the execution. The declarative part (called the *environment*) must be enriched with the declarations of all the standard predefined classes of Java, described in [J.G96].

The computational part (called the *program*) needs some pieces of information that are necessary, for example, to evaluate field access expressions and method calls (see 6.3.1). So we define a function A (for *Annotating*) that annotates the program with information deduced by using the type system. Then, the (dynamic) semantics of the computational part takes this *annotated* code and returns a transition system modeling the program (see fig.6.4.2).

While the static semantics is just an extension of the one developed by other authors in [S.D97], the *dynamic* one is largely different. This semantics can be described as *operational*, *structural* and *observational*.

We have chosen an *operational* approach, in the style of Plotkin's SOS [Plo81], that models a Java program by a labeled transition system (*lts* shortly, see D).

6.3 Peculiar aspects of Java

Even if Java is quite a simple language, which borrows many aspects and features from well-known OO programming languages, it is necessary to pay attention to some peculiarities that heavily impacts on the semantics we present in the following.

First, we briefly describe how we evaluate the expressions to access fields and to invoke methods. The mechanism is not as simple, since Java has instance as well as class fields and methods with overriding, thus the same name denotes different code in different classes; late binding is allowed only for instance methods.

6.3.1 Field Access and Method Invocation

Here we briefly describe how a field value or the code of a method can be accessed in Java and which restriction we impose to simplify the semantics; this justifies the information that the static semantics annotates on a program.

6.3.1.0.3 Field access semantics As we do not consider interfaces, an object cannot have an instance field, say `f`, multiply inherited by a superclass and by an interface. While in Java we can use expressions like `C.f` and `I.f` to access two different incarnations of `f`, one inherited by the superclass `C` and one by the interface `I`, here we assume that a subclass can only hide a field `f` of its superclass. A field `f` is hidden in a subclass by the declaration of a field with the same name; it can be accessed in any subclass instance by the field access expression `super.f`. So, an object can have two fields with the same name. In order to access the right one, the object has two distinct identifiers for the field `f`: simply `f` and `super.f`. Note that `super.super.f` is not a well-formed Java term.

The result of the expression `e.f` is computed as follows: if `f` is a class field, then the result is the class variable in the class that is the type of `e`; if `f` is an instance field, then the result is the specified instance variable, having the same type as `e`, in the object referenced by the value of `e`.

It is important to note that the field is determined only by the type of the expression, not by the class of the object referenced at run time by th expression value. There is no dynamic look-up for field access in Java; late binding is only available for instance methods, not for instance fields. If a field `f` of a class `S` is hidden in a subclass `T`, then any object of class `T` has two fields named `f`. If the evaluation of an expression `e` returns an object of class `T` but its type is `S`, then the value of the field access `e.f` is the value of `f` declared in `S`, hidden by the declaration of `T`.

6.3.1.0.4 Method call semantics Determining the method invoked by a method invocation expression is more complicated than accessing a field, because of method overloading and instance method overriding. The main problems are to figure out which class has to be considered for the definition of the method and which is the most specific method, since there may be several declarations for the same method name.

Given the environment, together with a class to start from, a method name and a typed list of actual parameters, the static semantics automatically determines the most applicable method, that is the method whose declaration can handle that method invocation without causing compile-time error. The terms representing method calls are annotated by this information. The problem we have to solve is to figure out the correct class in which start searching the method. Java uses static method binding for class methods while it allows dynamic method binding for instance method.

A method invocation expression has one of the possible forms listed below:
MethodInvocation:

`this`.*MethodName*(*ArgumentList*$_{opt}$)
Primary. *Identifier*(*ArgumentList*$_{opt}$)
`super`.*Identifier*(*ArgumentList*$_{opt}$)

The method determination involves several steps, as schematized in the following:

1. Calculate the type of the expression preceding the method name and use it to identify the compile-time class.

2. Search the compile-time class for method declarations and choose the most specific one [1].

   These two steps can be performed at compile-time, so the static semantics can add this information to the term.

3. If the most specific method declaration is `static` (a class method), then overriding is not allowed and the most specific method is the one to be invoked. Otherwise, if the most specific method is an instance method, then evaluate the expression and its value is the *target object*. If the method invocation is of the form `super`.*Identifier*(*ArgumentList*$_{opt}$), then the target object is the value of `this`. Then a *dynamic method lookup* is used.

   (a) Starting from the class of the target object (or from its superclass if the method invocation used the keyword `super`), search for a declaration of the method with the given number and type of parameters and the given return type.

   (b) If no method is found, then continue the search in the superclass of the class until finding a method declaration.

   The last two steps are performed at run-time, so the right choice of the method can be completed by the dynamic semantics.

It is worthwhile to note that fields and methods overriding are different. Shadowed variables are maintained in every object of the given class and each one can be accessed using, for

[1]A method declaration is *applicable* to a method invocation iff

- the number of parameters in the declaration equals the number of argument expressions in the method invocation
- the type of each argument can be converted to the type of the corresponding parameter.

A method declaration is *more specific* than another one if any invocation handled by the first method can be passed to the other one without any compile-time type error (see also Sect. 6.4.2.1.3).

example, casting expression. On the contrary, overridden methods cannot be called by instances. Only the last method declaration for an overridden method can be used. There is no way to refer to the other ones in the super-class hierarchy.

6.4 Java_1

Java_1 is the subset of Java consisting of the following constructs: class declarations and inheritance, class fields and methods, instance fields and methods, primitive types, object creation (by the `new` method), multi-threaded programming and thread management, plus the usual statements (assignment, conditional, return and concatenation). Exception handling, arrays, interfaces and applets are the main features that are not included in Java_1 neither in the other sublanguages. Method synchronization, on the contrary, will be analysed in Java_2.

6.4.1 Syntax

For simplicity, the complete syntax of Java_1, as given in [J.G96], is reported in Appendix 6.6.

We found that some simplifications do not change the expressiveness of Java but helps to make the semantics easier.

First of all, since we consider the "declarative" part to be completely disjoint from the computational one, the assignment expressions used for the initialization of variables and fields are not allowed. Furthermore, class field access is allowed only by class names and not by instance references. Then we assume that the `this` expression is always explicitly indicated when accessing instance fields or methods (so we do not have an assignment of the form `f=e`, but `this.f=e`). Finally, the execution of a method-call is rendered by a textual substitution of the method invocation statement with the body of the method. When the method returns, the computation goes on by executing the statement immediately following the method invocation. If the `return` instruction within a method body is not the last one, then it would be necessary to skip all the code after the return and before this instruction. So, for simplicity, we assume that the `return` statement is always the last statement in a method body.

6.4.2 Static Semantics

We consider separately the static (compile time) semantics and the dynamic one. The first one looks for errors that can be detected by using a sound type-system, before executing the program. In our case it is also used to decorate well-formed terms with the type information needed by the dynamic semantics (to evaluate, for example, field access and method invocation expressions).

6.4.2.1 The type system and the annotation function

In Java_1 it is possible to keep separate the type information (deduced by the declarations of variables, classes and methods) from the code. The *environment* contains the type information, while the *program* contains the code.

6.4.2.1.1 Environment An environment is a sequence of class declarations. A class declaration contains the definition of the inheritance relation for the class plus the field and method declarations. Method declarations contain access modifiers and parameter types. In the following, `ENV` is the set of all the possible environments, defined by

Env:
  StandardEnv
  Env, Decl
StandardEnv:
  `class` *Exception* `ext` *Objects,...,* ρ_Φ[2]
Decl:
  ClassDeclaration
  VariableDeclaration
ClassDeclaration:
  ClassExtDec$\{$*ClassDeclarations*$_{opt}\}$
ClassExtDec:
  `class` *ClassName* `ext` *ClassName*

Type:
  ClassName
  `bool`
  `char`
  `int`
ClassDeclarations:
  ClassDeclarations ClassDecl
  ClassDecl
ClassDecl:
  ClassMemberDeclaration

[2]Standard Environment is essentially a non-empty list of the declarations of all the predefined classes of Java

ClassMemberDeclaration:
  FieldDeclaration
  MethodDeclaration
FieldDeclaration:
  $FieldModifier_{opt}$ *VariableDeclaration*
VariableDeclaration:
  Type VariableDeclaratorId
VariableDeclaratorId:
  Identifier
FieldModifier:
  `st`
MethodDeclaration:
  MethodHeader
MethodHeader:
  $MethodModifier_{opt}$*ResultType MethodDeclarator*
ResultType:
  Type
  `void`

MethodDeclarator:
  Identifier($FormalParameterList_{opt}$)
FormalParameterList:
  FormalParameterList,FormalParameter
FormalParameter:
  Type VariableDeclaratorId
MethodModifier:
  `st , nst`

Thus an environment is a list of declarations and a declaration has the form

`ClassId ext ClassId {vardecs methdefs}`

where `vardecs` is a list of variable declarations (`x:T`) and `methdefs` is a list of method declarations (`met:`$T_1\times\ \ldots\times\ T_n\rightarrow$ `T`). It is ranged over by the metavariable ρ (ρ_Φ denotes the empty environment, that is the empty list of class declarations).

6.4.2.1.2 Programs A program is a sequence of class bodies, which, in turn, are sequences of method bodies. Each method body contains: the method identifier, the types of the arguments and of the result (which identify a method) plus the block of method code with its local environment. PROG is the set (ranged over by the metavariable `p`.) of Java_1 programs defined by (the complete definition is in the appendix 6.6):

Programs:
  ClassBodies$_{opt}$
ClassBodies:
  ClassBody$_{opt}$
  ClassBodies ClassBody
ClassBody:
  ClassName `ext` *ClassName MethodBodies*
MethodBodies:
  MethodBody$_{opt}$
  MethodBodies MethodBody
MethodBody:
  Identifier(ResultType $<--$ TypeList) Block
  ;
TypeList:
  Type
  Identifier : Type
  TypeList, Type
Block:
  { *BlockStatements*}
BlockStatements:
  BlockStatement
  [VariableDeclarations] BlockStatements
BlockStatement:
  Statement
VariableDeclarations:
  VariableDeclaration$_{opt}$
  VariableDeclarations VariableDeclaration

The static semantics, which assigns types to terms, needs only the information in the environment. In a similar way, we would like to use only the program for giving the dynamic semantics, discarding the environment. Unfortunately, some information calculated by the static semantics are used by the dynamic one too. The solution (see fig.6.4.2) is to give the dynamic semantics on a "transformed" program, in which the original code is substituted by the equivalent annotated one, obtained by decorating some terms with type information.

The annotation function:

$$A\text{: Java}_1\times \mathsf{ENV}\rightarrow \text{Java}_{1_A}.$$

returns the annotated Java program.

A type system associates a type with each Java_{1_A} terms; it is described by an inference system to prove assertions of the form $\rho \vdash t$: T with the meaning that terms t has type T in the environment ρ; at the same time, these rules define the function A.

6.4.2.1.3 Auxiliary definitions We define the following syntactic and semantics sets for the metavariable we will use in the type system and in the following:

- CL_ids: class names, ranged over by `C`;
- TYPE= CL_ids$\cup$ { int, bool, char }: types (as defined in 6.6), ranged over by metavariable `T`;
- FIELD: field names, ranged over by `f`;
- METHOD: method names, ranged over by `m`;
- METHOD_TYPE: method types (functional types over TYPE), ranged over by `MT`;
- IDENTIFIER: identifiers, ranged over by `id`;
- Modifiers= {`st`,`nst`}: access modifiers, ranged over by `acc`;
- STATEMENT: Java_1 statements (as defined in 6.6); ranged over by `stm`;
- INTEGER: integer values; ranged over by i;
- BOOL= {true, false}: boolean values; ranged over by b;
- CHAR: char values; ranged over by c;
- PRIMITIVE=INTEGER$\cup$ BOOL$\cup$ CHAR: primitive type values; ranged over by pv;
- FORMAL_PAR_LIST= List(IDENTIFIER): lists of parameters; ranged over by `plist`;

In the following, if `v` is a metavariable ranging over a generic set `S`, then $\mathtt{v}_1,..,\mathtt{v}_n$, $\mathtt{v}'$, $\mathtt{v}''$, ... are metavariables implicitly ranging over `S` too.

The main difference w.r.t. [S.D97] is that we consider local variables too. Thus we use a local environment when calculating the type of terms within a method block. The function A, when calculated on a method block, uses an environment of the form $\rho_l;\rho$, where ρ is the program environment and ρ_l is the local environment of the block (; is the concatenation operation between environments, as described in 6.6), that is discarded at the end of the block.

The name of a local variable can appear both in ρ_l and in ρ, so we define a table lookup function $LookUp(\rho,\mathtt{x})$, which returns the type of `x` in ρ. If `x` is declared twice, then the type of the first declaration is returned (in the following we use `cldecs` as an abbreviation for a list of fields and method declarations $\{\mathtt{mod}_1\mathtt{f}_1\!:\ \mathtt{T}_1, \ldots, \mathtt{mod}_n\mathtt{f}_n\!:\ \mathtt{T}_n, \mathtt{mod}'_1\mathtt{m}_1\!:\ \mathtt{MT}_1, \ldots, \mathtt{mod}'_k\mathtt{m}_k\!:\ \mathtt{MT}_k\}$.

6.4.2.1.4 Environment Given an environment ρ, *LookUp*(ρ,x) is defined as follows:

- *LookUp*(ρ,x) = T *if* ρ= x: T, ρ'
- *LookUp*(ρ,x) = ρ'(x) *if* ρ= x$'$: T$'$, $\rho'$and x$\neq$ x$'$
- *LookUp*(ρ,x) = `undef` *if* ρ= ρ_Φ
- *LookUpCl*(ρ,C) = C ext C$'$cldecs *if* ρ= C ext C$'$cldecs, ρ'
- *LookUpCl*(ρ,C) = *LookUpCl*(ρ,C) *if* ρ= C$'$ ext C$''$cldecs, ρ' and C$\neq$ C$'$
- *LookUpCl*(ρ_Φ,C) = `undef`

6.4.2.1.5 Auxiliary operations We use auxiliary operations to extract information from the environment:

- for a method type MT:$\mathtt{T}_1\times \ldots \times \mathtt{T}_n \rightarrow$ T the argument types are defined by: *Args*(MT) = $\mathtt{T}_1 \times \cdots \times \mathtt{T}_n$; the result type is: *Res*(MT) = T
- given an environment ρ, a class identifier C *st* *LookUpCl*(C) = C ext C$'\{$ $\mathtt{mod}_1\mathtt{f}_1$: $\mathtt{T}_1$, $\ldots$, $\mathtt{mod}_n\mathtt{f}_n$: $\mathtt{T}_n$,$\mathtt{mod}_1\mathtt{m}_1$:$\mathtt{MT}_1$, $\ldots$,$\mathtt{mod}_k\mathtt{m}_k$:$\mathtt{MT}_k\}$ and a field name f:

  FDec: ENV$\times$ CL_ids$\times$ FIELD$\rightarrow$ CL_ids$\times$ CL_ids$\times$ Modifiers: *FDec*(ρ, C,f) returns the nearest superclass of C (possibly C itself) which contains a declaration of f, its type and its access modifiers (st or nst).

  - *FDec*(ρ,C,f) = (C, $\mathtt{T}_j$, $\mathtt{mod}_j$) iff f=$\mathtt{f}_j$;
  - *FDec*(ρ,C,f) = *FDec*(ρ, C$'$,f) iff f$\neq$ $\mathtt{f}_j$, $\forall j \in \{1, \ldots, k\}$

  MDecs:ENV$\times$ CL_ids$\times$ METHOD$\rightarrow$ Set(CL_ids$\times$ METHOD_TYPE) returns all the classes and the signatures of the declarations of a method.

  MDecs(ρ, C,m) = $\{$(C,$\mathtt{MT}_j$) : m= $\mathtt{m}_j\}$ $\cup$ $\{$(C$''$,MT$''$):(C$''$,MT$''$)$\in$ *MDecs*(ρ,C$'$,m), and

  $\forall j \in \{1, \ldots, k\}$ m= $\mathtt{m}_j \Rightarrow$ *Args*($\mathtt{MT}_j$) = *Args*(MT$''$)$\}$
- A value of type T can be assigned to a variable of type T$'$ if T is a subclass of type T$'$ (written T $\sqsubseteq$ T$'$, where $\sqsubseteq$ is the subclass relation).

  The definition of the subclass $\sqsubseteq$ relation is:

– $$\frac{}{\rho\vdash\mathtt{C}\sqsubseteq\mathtt{C},\rho\vdash\mathtt{C}\sqsubseteq\mathtt{C}'}$$ **if** $\rho=\rho', \mathtt{C}\ \text{ext}\ \mathtt{C}' \ldots, \rho''$

– $$\frac{\rho\vdash\mathtt{C}\sqsubseteq\mathtt{C}' \quad \rho\vdash\mathtt{C}'\sqsubseteq\mathtt{C}''}{\rho\vdash\mathtt{C}\sqsubseteq\mathtt{C}''}$$

– $$\frac{}{\vdash\mathtt{Object}\sqsubseteq\mathtt{Object}}$$

- Given an environment ρ, variable types $\mathtt{T}$ and $\mathtt{T}_i$, $i\in\{1,\ldots,n+1\}$, and identifier $\mathtt{m}$, the *most special declaration* is defined as follows:
  - $\mathit{ApplMeths}(\rho,\mathtt{m},\mathtt{T},\mathtt{T}_1\times\ldots\times\mathtt{T}_n)=\{(\mathtt{T}',\mathtt{MT}')\mid(\mathtt{T}',\mathtt{MT}')\in\mathit{MDecs}(\rho,\mathtt{T},\mathtt{m})$ and $\mathtt{MT}'=\mathtt{T}_1{}',\ldots,\mathtt{T}_n{}'\rightarrow\mathtt{T}'_{n+1}$ and $\rho\vdash\mathtt{T}_i\sqsubseteq\mathtt{T}'_i$ for $i\in\{1,\ldots,n\}\}$
  - $(\mathtt{T},\ \mathtt{T}_1\times\ldots\times\mathtt{T}_n\rightarrow\mathtt{T}_{n+1})$ is *more special* than $(\mathtt{T}',\ \mathtt{T}_1{}'\times\ldots\times\mathtt{T}_n{}'\rightarrow\mathtt{T}'_{n+1})$ iff $\rho\vdash\mathtt{T}\sqsubseteq\mathtt{T}'$ and $\forall i\in\{1,\ldots,n\}$, $\rho\vdash\mathtt{T}_i\sqsubseteq\mathtt{T}_i{}'$
  - $\mathit{MostSpec}(\rho,\mathtt{m},\mathtt{T},\mathtt{T}_1\times\ldots\times\mathtt{T}_n)=$
    $\{(\mathtt{T}',\mathtt{MT}')\in\mathit{ApplMeths}(\rho,\mathtt{m},\mathtt{T},\mathtt{T}_1\times\ldots\times\mathtt{T}_n)$ and if $(\mathtt{T}'',\mathtt{MT}'')\in\mathit{ApplMeths}(\rho,\mathtt{m},\mathtt{T},\mathtt{T}_1\times\ldots\times\mathtt{T}_n)$ and
    $(\mathtt{T}'',\mathtt{MT}'')$ is *more special* than $(\mathtt{T}',\mathtt{MT}')$ then $\mathtt{T}''=\mathtt{T}'$ and $\mathtt{MT}''=\mathtt{MT}'\}$

The well-formedness relation on environments ($\rho\diamond$) requires that a class as well as a variable are declared just once, every class has a superclass, within a class declaration, two methods cannot have the same name and the same arguments and that widening and subclass relations are acyclic.

The well-formedness relation is not formally given (see [S.D97]).

6.4.2.2 Type Rules

In [S.D97], Java$_S$ is a subset of Java that lacks some important features, including: class variables and methods, local variables, `super` and concurrency that are included in Java$_1$.

Information that annotate terms of Java$_{1_A}$ are more complex that the ones used in Java$_{SE}$(in [S.D97]), since in Java$_1$ we consider also local variables, class fields and methods. For example, when evaluating a method call, it is necessary to know if the *most specific method* calculated at compile time is a class or an instance method. In the first case, also the information about the class of the most specific method must be available. Moreover, in

Java_1 there is also the `super` construct, that can be used both in field access and in method invocation expressions.

Type rules are given in the following, with a little comment for each one.

Base *Primitive values and variables are not annotated.*

$$\frac{}{\rho \vdash \texttt{null} : \mathsf{nil}, \rho \vdash \texttt{true} : \ \mathsf{bool}, \rho \vdash \texttt{false} : \ \mathsf{bool}, \rho \vdash \mathsf{i} : \ \mathsf{integer}, \rho \vdash \mathsf{c} : \ \mathsf{char}, \rho \vdash \texttt{id} : \mathit{LookUp}(\rho, \texttt{id})} \atop A\{\texttt{z}, \rho\} = \texttt{z}$$

if $\texttt{z}$ is i or c or $\texttt{id}$ or $\texttt{false}$ or $\texttt{true}$ or $\texttt{null}$

BaseClass *The type of a class identifier is the class name if it has been declared; the term is not annotated.*

$$\frac{}{\rho \vdash \texttt{C} : \texttt{C}} \atop A\{\texttt{C}, \rho\} = \texttt{C} \qquad \textbf{if} \;\; \mathit{LookUpCl}(\texttt{C}) = \texttt{C} \text{ ext } \texttt{C}'$$

Ass *An expression of type* $\texttt{T}$ *can be assigned to a variable of type* $\texttt{T}'$ *if* $\texttt{T}'$*is a subclass of* $\texttt{T}$*; the assignment expression has void type, the term is not annotated.*

$$\frac{\rho \vdash \texttt{t} : \texttt{T}, \rho \vdash \texttt{e} : \texttt{T}'}{\rho \vdash \texttt{t} = \texttt{e} : \texttt{void}} \atop A\{\texttt{t} = \texttt{e}, \rho\} = (A\{\texttt{t}, \rho\} = A\{\texttt{e}, \rho\}) \qquad \textbf{if} \;\; \rho \vdash \texttt{T}' \sqsubseteq \texttt{T}$$

Ret1 *A return statement has void type if it does not return any expression; the term is not annotated.*

$$\frac{}{\rho \vdash \texttt{return} : \texttt{void}} \atop A\{\texttt{return}, \rho\} = \texttt{return}$$

Ret2 *A return statement returning an expression has the type of the returned expression.*

$$\frac{\rho \vdash \texttt{e} : \texttt{T}}{\rho \vdash \texttt{return e} : \texttt{T}} \atop A\{\texttt{return e}, \rho\} = \texttt{return } A\{\texttt{e}, \rho\}$$

Seq *A statement sequence has the type of the last statement.*

$$\frac{\rho \vdash \texttt{stm} : \texttt{void}, \;\; \rho \vdash \texttt{stm} : \texttt{T}}{\rho \vdash \texttt{stm}; \texttt{stm} : \texttt{T}} \atop A\{\texttt{stm}; \texttt{stm}, \rho\} = A\{\texttt{stm}, \rho\}; A\{\texttt{stm}, \rho\}$$

New *A new object creation expression has the type of the class of the new object; the information needed when creating a new instance of a given class are recorded in the class* *itself, so the annotation function does not add information to a creation term.* $\rho \vdash \mathtt{C} \sqsubseteq \mathtt{C}$ *denotes that* C *is defined as a class in the environment.*

$$\frac{}{\begin{array}{c}\rho \vdash \mathtt{new\ C} : \mathtt{C}\\ A\{\mathtt{new\ C}, \rho\} = \mathtt{new\ C}\end{array}} \quad \textbf{if} \ \ \rho \vdash \mathtt{C} \sqsubseteq \mathtt{C}$$

Cond *A conditional statement has void type, since both the statement lists have void type.*

$$\frac{\rho \vdash \mathtt{e} : \mathsf{bool}, \rho \vdash \mathtt{stm} : \mathtt{void}, \ \ \rho \vdash \mathtt{stm}' : \mathtt{void}}{\begin{array}{c}\rho \vdash \mathtt{if\ e\ stm\ else\ stm}' : \mathtt{void}\\ A\{\mathtt{if\ e\ stm\ else\ stm}'\} = \mathtt{if}\ A\{\mathtt{e}, \rho\}\ A\{\mathtt{stm}, \rho\}\ \mathtt{else}\ A\{\mathtt{stm}', \rho\}\end{array}}$$

InstanceFieldAcc *A term* t *must have a class type* T *that defines or inherits from a superclass the field* f *with type* T′*(the type of the field access expression); a field access by an expression is correct only if the field is declared to be non static (*nst*); the annotation function adds the class in which the field is declared and about the access modifier*

$$\frac{\rho \vdash \mathtt{t} : \mathtt{C}}{\begin{array}{c}\rho \vdash \mathtt{t.f} : \mathtt{T}'\\ A\{\mathtt{t.f}, \rho\} = A\{\mathtt{t}, \rho\}.[\mathtt{C}', \mathtt{nst}]\mathtt{f}\end{array}} \quad \textbf{if} \ \ \mathit{FDec}(\rho,\mathtt{C},\mathtt{f}) = (\mathtt{C}',\mathtt{T}, \ \mathtt{nst})$$

ClassFieldAcc *Similar to previous case, but a field access by a class identifier is correct only if the field in the class (or inherited by the class) is declared to be static (*st*); the annotation function adds the class in which the field is declared and about the access modifier. Let's remember that class field access can be possible only by using class names, not by instance identifiers, thus a class field access expression has the form:* C.f

$$\frac{\rho \vdash \mathtt{C} : \mathtt{C}}{\begin{array}{c}\rho \vdash \mathtt{C.f} : \mathtt{T}\\ A\{\mathtt{C.f}, \rho\} = \mathtt{C}.[\mathtt{C}', \mathtt{st}]\mathtt{f}\end{array}} \quad \textbf{if} \ \ \mathit{FDec}(\rho,\mathtt{C},\mathtt{f}) = (\mathtt{C}',\mathtt{T}, \ \mathtt{st})$$

MethCall *a method can be applied to its parameters if the types of the actual parameters can be widened to the types of the corresponding formal ones; the annotation function adds a triple (*C*, Args(*MT*),*mod*) where* C *is the class, Args(*MT*) are the argument types and* mod *is the access modifier of the most specific method*

$$\frac{\rho \vdash \mathtt{e}_i : \mathtt{T}_i \ \ i = 1, \ldots, n}{\begin{array}{c}\rho \vdash \mathtt{e}_1.\mathtt{m}(\mathtt{e}_2, \ldots, \mathtt{e}_n) : \mathit{Res}(\mathtt{MT})\\ A\{\mathtt{e}_1.\mathtt{m}(\mathtt{e}_2, \ldots, \mathtt{e}_n), \rho\} = A\{\mathtt{e}_1, \rho\}.[\mathtt{T}, \mathit{Args}(\mathtt{MT}), \mathtt{mod}]\mathtt{m}(A\{\mathtt{e}_2, \rho\}, \ldots, A\{\mathtt{e}_n, \rho\})\end{array}}$$

if $n \geq 1, \mathit{MostSpec}(\rho,\mathtt{m},\mathtt{T}_1,\mathtt{T}_2\times \ \ldots \times\mathtt{T}_n) = \{ \ (\mathtt{T},\mathtt{MT}, \ \mathtt{mod})\}$

MethBody *The annotation function adds the local environment of the method to the program environment* ρ

$$\frac{\begin{array}{c}\rho \vdash \mathtt{this} : \mathtt{C} \\ \rho_l ; \rho, \mathtt{z}_1 : \mathtt{T}_1 \ldots \mathtt{z}_n : \mathtt{T}_n \vdash \mathtt{stm}[\mathtt{z}_1/\mathtt{x}_1 \ldots \mathtt{z}_n/\mathtt{x}_n] : \mathtt{T}'\end{array}}{\begin{array}{c}\rho \vdash \mathtt{mBody} : \mathtt{T}_1 \times \cdots \times \mathtt{T}_n \rightarrow \mathtt{T} \\ A\{\mathtt{mBody}, \rho\} = \mathtt{m}(\mathtt{T} < - - \mathtt{x}_1 : \mathtt{T}_1, \ldots, \mathtt{x}_n : \mathtt{T}_n) A\{\{[\rho_l]\mathtt{stm}\}, \rho_l ; \rho\}\end{array}}$$

if $\rho \vdash \mathtt{T}' \sqsubseteq \mathtt{T}$, $\mathtt{mBody} = \mathtt{m}(\mathtt{T} < \, -- \, \mathtt{x}_1 : \mathtt{T}_1, \ldots, \mathtt{x}_n : \mathtt{T}_n)\{[\rho_l] \; \mathtt{stm}\}$, $\mathtt{x}_i \neq \mathtt{this}$, $i \in \{1 \ldots n\}$, $\mathtt{z}_1 \ldots \mathtt{z}_n$ are new variables in $\rho_l;\rho$

ClassBody *A class is well-formed if it implements all the methods in its declaration. The annotation function adds the type of* `super` *to the local environment; the type of* `this` *and* `super` *can be calculated by applying the* `Base` *rule*

$$\frac{\rho ; \mathtt{this} : \mathtt{C} ; \mathtt{super} : \mathtt{S} \vdash \mathtt{mBody}_i : \mathtt{MT}_i \quad i = 1 \ldots m}{\begin{array}{c}\rho \vdash \mathtt{cBody} : \mathit{LookUpCl}(\rho, \mathtt{C}) \\ A\{\mathtt{cBody}, \rho\} = \mathtt{C} \ \mathtt{ext} \mathtt{C}' \{ A\{\mathtt{mBody}_1, \rho ; \mathtt{this} : \mathtt{C} ; \mathtt{super} : \mathtt{S}\} \ldots A\{\mathtt{mBody}_m, \rho ; \mathtt{this} : \mathtt{C} ; \mathtt{super} : \mathtt{S}\}\}\end{array}}$$

if $\rho \vdash \rho \diamond$, $n \geq 0, k \geq 0, m \geq 0$,
$\mathit{LookUpCl}(\rho, \mathtt{C}) = \mathtt{C} \ \mathtt{ext} \ \mathtt{C}' \ldots \mathtt{cBody} = \mathtt{C} \ \mathtt{ext} \mathtt{C}' \{ \mathtt{mBody}_1, \ldots, \mathtt{mBody}_m \}$,
$\mathtt{mBody}_i = m_i$ is implem_i, $\mathit{LookUp}(\rho, \mathtt{this}) = \mathtt{undef}$, $\mathit{LookUp}(\rho, \mathtt{super}) = \mathtt{undef}$, $\mathit{Super}(\mathtt{C}, \rho) = \mathtt{S}$

ProgramBody *a program is well-formed if it contains a well-typed class body for each class in its declaration*

$$\frac{\rho \vdash \mathtt{cBody}_i : \mathit{LookUpCl}(\rho, \mathtt{C}_i) \quad i = 1, \ldots, n}{\begin{array}{c}\rho \vdash \mathtt{p} \diamond \\ A\{\mathtt{p}, \rho\} = A\{\mathtt{cBody}_1, \rho\} \ldots A\{\mathtt{cBody}_n, \rho\}\end{array}}$$

if $n \geq 0$, $\mathtt{C}_1, \ldots, \mathtt{C}_n$ are all the classes defined in `p`, $\mathtt{p} = \mathtt{cBody}_1, \ldots, \mathtt{cBody}_n$, $\mathtt{cBody}_i = \mathtt{C}_i \mathtt{ext} \ldots$

We think that it is possible to give an operational semantics to Java_1 that does not need this static annotating semantics. Indeed, it is possible to give rules that "calculate" the added type information "on the fly", by means of the information stored in the global environment. In such case it would not be necessary to use two separate syntax for simple and annotated terms even if, as a counterpart, the environment cannot be discarded in the dynamic semantics. This approach "mixes" rules defining the type system together with the rules for the operational semantics. Premises and side conditions for one rule of this "mixed" semantics are the union of the premises and the side conditions of a single operational rule and one or more type system rules.

However, it easy to see that this approach makes more difficult to prove the correctness of the semantic rules, since it is less modular, while following the approach in [S.D97] we can check separately the correctness of type association to terms and the correctness of operational semantics.

There are other positive aspects in the separate approach. The first one is that the annotations do not heavily transform the program, since it is not necessary to calculate type information for every term, but only the information used for field accesses and method invocations. The other one is that the evaluation of the annotated program does not refer anymore to declarations, so the environment is used only in the static semantics and forgotten in the run-time one.

6.4.3 Dynamic Semantics

We have chosen an operational approach that models a Java program by a labeled transition system (*lts* shortly). *lts*s are described in sect.D, here we briefly recall that it is a triple $(State, Label, \rightarrow)$ where *State* is the set of the meaningful intermediate states of the computations of the program, *Label* is the set of the labels describing information exchanged with the external environment during the computation and $\rightarrow \subseteq State \times Label \times State$ is the transition relation describing how the computation proceeds: $s \xrightarrow{l} s'$ means that the program in the state s may perform a transition with the information exchange to the external environment described by l ending in the state s'.

The transitions of the *lts* model not just the execution of single statements, but also the low level actions that are involved in the execution of complex statements. For example, entering a synchronized block implies the execution of locking operations, which do not correspond to Java statements.

The choice of the information put on the labels of the system is crucial for the semantics, since it also determines what is *observable* from outside the program and, consequently, whether two programs are equivalent. The observation of the data exchange involved in the input-output operations performed over files and sockets seems to be a good choice.

From our point of view, the Java support for non-sequential programming is one of the most interesting feature of this language, mainly because our method is addressed to the development of concurrent systems. So, we want a semantics that put a particular emphasis on describing the role of separate concurrent components inside a Java program. In a multi-threaded Java program we can identify several computation flows, associated with "particular" objects, which are instances of *thread classes* (with the term *thread class* we denote the class `Thread` and all its subclasses). These objects have a particular `run` method, whose execution runs concurrently with the other ones, so we can call these objects *active components* of the program in contrast with the *passive components*, (classes and instances of *non thread* classes), that do not perform an independent activity and whose state is modified only by the action of the active components.

Our semantics reflects this *structure*. The passive components are modeled as a memory function that can be updated by the active components. The active components, in turn, are modeled by labelled transition systems. The activity of the whole program is than defined by combining the concurrent behaviour of the active components and by the status of the passive ones.

In our structural approach, classes have a semantics too, since classes have states (given by the value of class fields) and methods, as well as objects. Moreover, an explicit description of the class states is useful to record the identities of the instances created for a class.

Let us remember that the program contains the bodies of the methods and thus it is necessary to give the dynamic semantics: the transition relations for the *lts*' in the following are annotated with a metavariable p denoting the program to be modeled.

We use the following sets of semantic values for (active and passive) components identifiers, with the associated metavariables:

- $\mathsf{TH\_ids}$: thread *identifiers*; ranged over by t_{id};
- $\mathsf{OB\_ids}$: (non-thread) object identifiers; ranged over by o_{id};
- $\mathsf{COMP\_ids} = \mathsf{TH\_ids} \cup \mathsf{OB\_ids} \cup \mathsf{CL\_ids}$: generic component identifiers; ranged over by g_{id};
- $\mathsf{NonCL\_ids} = \mathsf{TH\_ids} \cup \mathsf{OB\_ids}$: instance identifiers; ranged over by nc_{id}.
- $\mathsf{VALUE} = \mathsf{COMP\_ids} \cup \mathsf{PRIMITIVE} \cup \{\mathtt{null}\}$: all the possible values for an identifier; ranged over by v (null denotes a reference to a special "null" object).

Thread and object identifiers contain also information about the class they are instances of; such information can be extracted by the function $Class() : \mathsf{NonCL\_ids} \rightarrow \mathsf{CL\_ids}$.

6.4.4 Thread Semantics

The semantics of threads is given by the *lts*: $\mathsf{T\_LTS}_1$=($\mathsf{T\_states}_1$, $\mathsf{T\_labels}_1$, $--->_{\mathtt{p}}{}^{1}$)

6.4.4.0.1 States

The state of a thread in each instant, during the program execution, can be modeled by the identity, the class, the code to be executed, the values of the local variables, the contents of its working memory and the values of its instance fields. Furthermore, it is necessary to know the *status* of the thread, that can be started, stopped or suspended.

6.4.4.0.2 $\mathsf{T\_states}_1 = \mathsf{TH\_ids} \times \mathsf{STATEMENT} \times \mathsf{Status}_1 \times \mathsf{MEMORY} \times \mathsf{Stacks}$,
where :

- STATEMENT
  is the set of statements, as defined in appendix 6.6, ranged over by `stm`;

  `skip`;`stm` and `stm`;`skip` are simply denoted by `stm`, while {`skip`} is abbreviated in `skip`.

- $\mathsf{MEMORY} = \mathsf{FIELD} \times \mathsf{TYPE} \rightarrow_{\mathcal{P}} \mathsf{VALUE}$[3]

  is the set of memories, ranged over by m;

  $m[(\mathtt{f},\mathtt{T}) \rightarrow \mathsf{v}]$ denotes the updating of m that associates v to $(\mathtt{f},\mathtt{T})$ and behaves as m on the other couples. We will use the same notation for updating operations on other functions.

- Stacks= Stacks(Locals) and $\mathsf{Locals} = \mathsf{IDENTIFIER} \rightarrow_{\mathcal{P}} \mathsf{VALUE}$;

  is the set of stacks of the local memories ranged over by lmS; emptyst denotes the empty stack of local memories.

- Status=$\{New, PreStop, Run, Stop, Susp\}$ is the set of all the the possible status a thread can assume:

  - *New*: not yet started; it has been created but not yet started by some other thread.
  - *PreStop*: stopped but not started; it has been stopped while it was waiting to be started.
  - *Run*: runnable; it has already been started and it is executing its run method.
  - *Stop*: it either has terminated the execution of its run method or another thread has stopped it;
  - *Susp*: suspended; it has been suspended by some other thread while it was executing; it will remain in such state until another thread resumes it.

- $\mathsf{MET\_str} = \mathsf{FORMAL\_PAR\_LIST} \times \mathsf{STATEMENT} \times \mathsf{Locals}$; method structures; ranged over by mst.

[3] In the following, we use $f : D \rightarrow_{\mathcal{T}} C$ to denote the class of the total functions from D in C, and $f : D \rightarrow_{\mathcal{P}} C$ to denote the class of the partial ones.

6.4.4.0.3 Auxiliary operations:

we define some operations to access memory stacks and local memories.

- $Top, Pop, Push$: Stacks $\rightarrow$ Locals

  are the obvious operations on a stack;

- $CurEnt$: Stacks $\rightarrow$ COMP_ids

  defined by: $CurEnt(lmS) = Top(lmS)(\mathtt{this})$,
  returns the current object on the local environment lm, on the top of the stack;

- $Eval$: Stacks $\times$ IDENTIFIER $\rightarrow$ VALUE

  defined by: $Eval(lmS, \mathtt{id}) = \begin{cases} Eval(Pop(lmS), \mathtt{id}) \text{ if } Top(lmS)(\mathtt{id}) = \mathtt{undef}, \\ Eval(lmS, \mathtt{id}) = Top(lmS)(\mathtt{id}) \text{ otherwise,} \end{cases}$

  returns the value of a local variable `id`in the local memory on the top of the stack;

- $ModifyMem$: TYPE $\times$ FIELD $\times$ PRIMITIVE $\times$ MEMORY $\rightarrow$ MEMORY

  defined by: $ModifyMem(\mathtt{T},\mathtt{f},\mathsf{v},m) = m[(\mathtt{f},\mathtt{T}) \rightarrow \mathsf{v}]$

  returns the new memory obtained by associating a new value to a field;

- $UpdateMem$: IDENTIFIER $\times$ VALUE $\times$ Stacks $\rightarrow$ Stacks

  defined by: $UpdateMem(\mathtt{id}, \mathsf{v}, lmS) = Push(lm[\mathtt{id} \rightarrow \mathsf{v}], Pop(lmS))$

- $MethodDec$: METHOD $\times$ $Args$ $\times$ TYPE $\times$ PROG $\rightarrow$ MET_str

  given the method name, the signature and the program, it returns the correct method structure found in the program, that contains the list of the formal parameters, the list of statements and the local environment of the method call.

  To access the component of the structure, we use the auxiliary operations:

  - $Code$: MET_str $\rightarrow$ STATEMENT

    defined by $Code((\mathtt{plist}, \mathtt{stm}, lm)) = \mathtt{stm}$ returns the code in the method declaration mst;

  - $Param$: MET_str $\rightarrow$ FORMAL_PAR_LIST

    defined by $Param((\mathtt{plist}, \mathtt{stm}, lm)) = \mathtt{plist}$ returns the list of formal parameters in the method declaration mst;

  - Loc: MET_str $\times$ $List(Value)$ $\times$ COMP_ids $\rightarrow$ Locals characterized by

    $Loc(\mathsf{mst}, \{v_1, .., v_n\}, g_{id}) = lm \implies$ if $Param(\mathsf{mst}) = \mathtt{this}, \mathtt{id}_1, .., \mathtt{id}_n$ then $lm(\mathtt{this}) = g_{id}$ and $lm(\mathtt{id}_i) = v_i$ returns the local memory for the method described by mst: $\{v_1,..,v_n\}$ and g_{id} are used for initializing, respectively, the value of formal parameters and of `this` reference.

6.4.4.0.4 Labels

$\mathsf{T\_labels}_1 = \mathsf{AccLabs} \cup \mathsf{MCallLabs} \cup \mathsf{ThreadControlLabs} \cup \{\tau\}$

There are four different kinds of labels, describing respectively: access to class and object fields(AccLabs), method calls (MCallLabs), thread state control actions (ThreadControlLabs) and a generic internal activity (τ). The labels are n-uples, but we write $label(x_1, \dots, x_n)$ instead of $(label, x_1, \dots, x_n)$.

We briefly analyse every label (in the following $\mathsf{Args} \equiv \mathsf{COMP\_ids} \times \mathsf{TYPE} \times \mathsf{FIELD}$):

- $\mathsf{AccLabs} = (\{AssO, GetO\} \times \mathsf{NonCL\_ids} \times \mathsf{TYPE} \times \mathsf{FIELD} \times \mathsf{VALUE}) \cup$ $(\{AssC, GetC\} \times \mathsf{CL\_ids} \times \mathsf{TYPE} \times \mathsf{FIELD} \times \mathsf{VALUE})$
  - $AssO(nc_{id}, \mathtt{T}, \mathtt{f}, \mathtt{v})$: a thread assigns the value v to the field f of type T in the object nc_{id};
  - $AssC(\mathtt{C}, \mathtt{f}, \mathtt{v})$: a thread assigns the value v to the field f in the class C;
  - $GetO(nc_{id}, \mathtt{T}, \mathtt{f}, \mathtt{v})$: a thread gets the value v of the field f of type T from the object nc_{id};
  - $GetC(\mathtt{C}, \mathtt{T}, \mathtt{f}, \mathtt{v})$: a thread gets the value v of the field f of type Tfrom class C;
- $\mathsf{MCallLabs} = (\{New\} \times \mathsf{CL\_ids} \times \mathsf{NonCL\_ids}) \cup$

  $(\{CallO\} \times \mathsf{NonCL\_ids} \times \mathsf{METHOD} \times \mathsf{Args} \times \mathsf{MET\_str}) \cup$

  $(\{CallC\} \times \mathsf{CL\_ids} \times \mathsf{METHOD} \times \mathsf{Args}) \cup$

  $(\{SelfCall\} \times \mathsf{METHOD} \times \mathsf{Args})$;
  - $New(\mathtt{C}, nc_{id})$: a thread performs a new instance creation call to class C that returns the identifier nc_{id};
  - $CallO(nc_{id}, \mathtt{m}, \mathtt{args}, \mathsf{mst})$: a thread calls a method m of the object nc_{id} with the argument types args and receives the method declaration structure mst.
  - $CallC(\mathtt{C}, \mathtt{m}, \mathtt{args})$: a thread calls a method m of the C class with arguments types args; the correct method declaration structure can be constructed from the information given by static semantics;
  - $SelfCall(\mathtt{m}, \mathtt{args})$: a thread calls one of its methods m, with argument types args;
- $\mathsf{ThreadControlLabs} = \{Send, Receive\} \times \{StartT, Stop, Suspend, Resume\} \times \mathsf{TH\_ids}$;
  - $Send(StartT, t_{id}), Send(Stop, t_{id}), Send(Suspend, t_{id}), Send(Resume, t_{id})$: a thread starts, stops, suspends or resumes another thread t_{id}.
  - $Receive(StartT, t_{id}), Receive(Stop, t_{id}), Receive(Suspend, t_{id}), Receive(Resume, t_{id})$: a thread t_{id} receives the start, stop, suspend or resume signal;
- Internal action: τ (usually omitted in the rules below).

6.4.4.0.5 Transition relation

$--->_{\mathtt{p}}{}^{1}$: $\mathsf{T\_states}_1 \times \mathsf{T\_labels}_1 \times \mathsf{T\_states}_1$

defined by the inductive rules below. In the rules, all the non-meaningful components of the thread states are omitted. For example, $<$`stm`, $lmS>$ stands for $<t_{id}$,`stm`,$tSt,m,lmS>$. When the identifier and the status are omitted, we assume that their value is, respectively, t_{id} and Run.

When evaluating method calls and assignments, it is necessary to distinguish when a term is ground: we distinguish between r_ground term (term that are on the right part of an assignment and cannot be rewritten) and l_ground term (term on the left part).

A term `t` is said:

- r_ground if `t` is a primitive value or `t` $\equiv g_{id}$ for some identifier g_{id};
- l_ground if `t` is r_ground or `t` $\equiv$ `id` for some identifier `id` or `t` $\equiv$ `y.f` for some reference variable `y` and field name `f`.

Furthermore, we say that a list of terms is r_ground if every element is a r_ground term.

6.4.4.0.6 Expression Semantics

The evaluation of an expression returns a value. An expression is evaluated from left to right.

- Field Access

The value of a local identifier is in the local memory; the value of an object or class field is read from the memory of the corresponding object.

TField1 *Evaluation of a local identifier.*

$$\frac{}{<\mathtt{id}, lmS> \overset{\mathcal{T}}{--->}_{\mathtt{p}}{}^{1} < Eval(lmS, \mathtt{id}), lmS>}$$

TField2 *Evaluation of* `this`*.*

$$\frac{}{<\texttt{this}> \overset{\tau}{---\!>}_{\texttt{p}}{}^{1} < CurEnt(lmS) >}$$

TField3 *Evaluation of an object field: the thread accesses the object memory.*

$$\frac{}{< o_{id}.[\texttt{T},\texttt{mod}]\texttt{f} > \overset{AssO(o_{id},\texttt{T},\texttt{f},\texttt{v})}{---------\!>}_{\texttt{p}}{}^{1} < \texttt{v} >}$$

if $t_{id} \neq o_{id}$

TField4 *Evaluation of a class field: the thread looks for its value in the corresponding class or object.*

$$\frac{}{< \texttt{C}.[\texttt{C},\texttt{st}]\texttt{f} > \overset{GetC(\texttt{C},\texttt{f},\texttt{v})}{--------\!>}_{\texttt{p}}{}^{1} < \texttt{v} >}$$

TField5 *Evaluation of a field accessed by* `super`.

$$\frac{}{< \texttt{super}.[\texttt{T},\texttt{mod}]\texttt{f}, lmS > \overset{AssO(g_{id},\texttt{T},\texttt{f},\texttt{v})}{---------\!>}_{\texttt{p}}{}^{1} < \texttt{v}, lmS >}$$

if $CurEnt(lmS) = g_{id}$

TField6 *evaluation of a thread field: the thread looks for its value in its own memory.*

$$\frac{}{< t_{id}.[\texttt{T},\texttt{mod}]\texttt{f}, m > \overset{\tau}{---\!>}_{\texttt{p}}{}^{1} < m(\texttt{f},\texttt{T}) >}$$

TField8 *evaluation of a thread field accessed by* `super`.

$$\frac{}{< \texttt{super}.[\texttt{T},\texttt{mod}]\texttt{f}, m, lmS > \overset{\tau}{---\!>}_{\texttt{p}}{}^{1} < m(\texttt{f},\texttt{T}), m, lmS >}$$

if $CurEnt(lmS) = t_{id}$

- Creation Expression

TNew *A thread receives the reference for a new object.*

$$\frac{}{<\texttt{new C}> \xrightarrow{New(\mathtt{C},g_{id})}{}_{\mathtt{p}}^{1} < g_{id} >}$$

• Method Call Expressions

A method call is handled by concatenating the code of the method to the code that the thread has to execute; a new local memory is pushed into the stack.

In the following, v is a r_ground value, vals is a list of r_ground terms and args∈ Argsis a list of types of arguments for a method, while `in` is an abbreviation for (T,args,mod).

TCall1 *First we evaluate the expression denoting the component that receives the method call.*

$$\frac{< \mathsf{e} > \xrightarrow{l}{}_{\mathtt{p}}^{1} < \mathsf{e}' >}{< \mathsf{e}.[\mathtt{in}]\mathtt{m}(\mathtt{p\_list}) > \xrightarrow{l}{}_{\mathtt{p}}^{1} < \mathsf{e}'.[\mathtt{in}]\mathtt{m}(\mathtt{p\_list}) >}$$

TCall2 *Then we evaluate the parameters (from left to right).*

$$\frac{< \mathsf{e}_1 > \xrightarrow{l}{}_{\mathtt{p}}^{1} < \mathsf{e}_1' >}{< g_{id}.[\mathtt{in}]\mathtt{m}(\mathsf{v}_1,\ldots,\mathsf{v}_i,\mathsf{e}_1,\ldots,\mathsf{e}_n), m, lmS > \xrightarrow{l}{}_{\mathtt{p}}^{1} < g_{id}.[\mathtt{in}]\mathtt{m}(\mathsf{v}_1,\ldots,\mathsf{v}_i,\mathsf{e}_1',\ldots,\mathsf{e}_n), m', lmS' >}$$

if $\mathsf{v}_1,\ldots,\mathsf{v}_i$ are r_ground

TCall3 *When the method is static, i.e., there is no dynamic binding, it is not necessary to know the class of the object and the method code is calculated using the annotations on the program.*

$$\frac{}{< g_{id}.[\mathtt{T},\mathtt{args},\mathtt{st}]\mathtt{m}(\mathsf{vals}), lmS > \xrightarrow{CallC(\mathtt{T},\mathtt{m},\mathtt{args})}{}_{\mathtt{p}}^{1} < \{\mathtt{stm}\}, Push(lm, lmS) >}$$

if $MethodDec(\mathtt{m},\mathtt{args},\mathtt{T},\mathtt{p}) = \mathsf{mst}, \mathtt{stm} = Code(\mathsf{mst}),\ lm = Loc(\mathsf{mst},\mathsf{vals},g_{id})$

TCall4 *If the method is not static, it is necessary to ask the object, whose class is not known, to send the correct code for the method.*

$$\frac{}{< nc_{id}.[\mathtt{T}, \mathtt{args}, \mathtt{nst}]\mathtt{m}(\mathsf{vals}), lmS > \xrightarrow{CallO(nc_{id}, \mathtt{m}, \mathtt{args}, \mathsf{mst})}_{\mathtt{p}}{}^{1} < \{\mathtt{stm}\}, Push(lm, lmS) >}$$

if $nc_{id} \neq t_{id}$, $\mathtt{stm} = Code(\mathsf{mst})$, $lm = Loc(\mathsf{mst}, \mathsf{vals}, nc_{id})$

TCall5 *A self method call; it is not just an internal action, because when introducing synchronization (see Java$_2$), it is necessary to distinguish this action w.r.t. other internal ones.*

$$\frac{}{< t_{id}.[\mathtt{T}, \mathtt{args}, \mathtt{nst}]\mathtt{m}(\mathsf{vals}), lmS > \xrightarrow{SelfCall(\mathtt{m}, \mathtt{args})}_{\mathtt{p}}{}^{1} < \{\mathtt{stm}\}, Push(lm, lmS) >}$$

if $\mathtt{C} = Class(t_{id})$, $\mathsf{mst} = MethodDec(\mathtt{m}, \mathtt{args}, \mathtt{C}, \mathtt{p})$, $\mathtt{stm} = Code(\mathsf{mst})$, $lm = Loc(\mathsf{mst}, \mathsf{vals}, t_{id})$

TCall6 *A call of a method of the superclass; there is only one rule, since static and dynamic type of the expression are the same. The value of the* current object*, that is the object whose method is executed, is the value of* `this` *in the new local memory pushed on the stack.*

$$\frac{}{< \mathtt{super}.[\mathtt{T}, \mathtt{args}, \mathtt{mod}]\mathtt{m}(\mathsf{vals}), lmS > \xrightarrow{CallC(\mathtt{T}, \mathtt{m}, \mathtt{args})}_{\mathtt{p}}{}^{1} < \{\mathtt{stm}\}, Push(lmS, lm) >}$$

if $t_{id} = CurEnt(lmS)$, $MethodDec(\mathtt{m}, \mathtt{args}, \mathtt{T}, \mathtt{p}) = \mathsf{mst}$, $\mathtt{stm} = Code(\mathsf{mst})$, $lm = Loc(\mathsf{mst}, \mathsf{vals}, t_{id})$

6.4.4.0.7 Statements Semantics

- Assignment

The assignments are evaluated from left to right. The left-hand side is evaluated until it is l_ground, then the right-hand is evaluated, till obtaining a primitive value or a reference to an object.

TAss1 *These two rules ensure that expressions like* `x` *or* g_{id}`.f` *are not further evaluated when are on the left-hand side of an assignment.*

$$\frac{< \mathsf{e}_1 > \overset{l}{--->}_{\mathrm{p}}{}^{1} < \mathsf{e}_1{}' >}{< \mathsf{e}_1 = \mathsf{e}_2 > \overset{l}{--->}_{\mathrm{p}}{}^{1} < \mathsf{e}_1{}' = \mathsf{e}_2 >}$$

if e_1 *not* l_ground

TAss2 ...

$$\frac{< \mathsf{e} > \overset{l}{--->}_{\mathrm{p}}{}^{1} < \mathsf{e}' >}{< \mathsf{l} = \mathsf{e} > \overset{l}{--->}_{\mathrm{p}}{}^{1} < \mathsf{l} = \mathsf{e}' >}$$

if l is l_ground

TAss3 *The assignment of a local identifier modifies the local memory.*

$$\frac{}{< \mathtt{id} = \mathsf{v}, lmS > \overset{\tau}{--->}_{\mathrm{p}}{}^{1} < \mathtt{skip}, UpdateMem(\mathtt{id}, \mathsf{v}, lmS) >}$$

if $lm = Top(lmS)$

TAss4 *The assignment of a class field modifies it.*

$$\frac{}{< \mathtt{C}.[\mathtt{C}', \mathtt{st}]\mathtt{f} = \mathsf{v} > \overset{AssC(\mathtt{C},\mathtt{C}',\mathtt{f},\mathsf{v})}{--------->}_{\mathrm{p}}{}^{1} < \mathtt{skip} >}$$

TAss5 *The assignment of an object field modifies it.*

$$\frac{}{< ncl_{id}.[\mathtt{C}', \mathtt{nst}]\mathtt{f} = \mathsf{v} > \overset{AssO(ncl_{id},\mathtt{C}',\mathtt{f},\mathsf{v})}{---------->}_{\mathrm{p}}{}^{1} < \mathtt{skip} >}$$

if $ncl_{id} \neq t_{id}$

TAss6 *The assignment of a thread field modifies its memory.*

$$\frac{}{< t_{id}.[\mathtt{C}',\mathtt{nst}]\mathtt{f} = \mathtt{v}, m > \overset{\tau}{---\!>}_{\mathtt{p}}{}^{1} < \mathtt{skip}, ModifyMem(\mathtt{C}', \mathtt{f}, \mathtt{v}, m) >}$$

if $ncl_{id} \neq t_{id}$

TAss7 *The assignment of a field accessed by super modifies the memory of the current object.*

$$\frac{}{< \mathtt{super}.[\mathtt{C}',\mathtt{nst}]\mathtt{f} = \mathtt{v}, lmS > \overset{AssO(ncl_{id},\mathtt{C}',\mathtt{f},\mathtt{v})}{-----------\!>}_{\mathtt{p}}{}^{1} < \mathtt{skip}, lmS >}$$

if $CurEnt(lmS) = ncl_{id}$

- CONCATENATION

TConc

$$\frac{< \mathtt{stm}_1, tSt, m, lmS > \overset{l}{---\!>}_{\mathtt{p}}{}^{1} < \mathtt{stm}_1', tSt', m', lmS' >}{< \mathtt{stm}_1; \mathtt{stm}_2, tSt, m, lmS > \overset{l}{---\!>}_{\mathtt{p}}{}^{1} < \mathtt{stm}_1'; \mathtt{stm}_2, tSt', m', lmS' >}$$

- IF - ELSE

TIf1 *The boolean expression is evaluated first.*

$$\frac{< \mathtt{e} > \overset{l}{---\!>}_{\mathtt{p}}{}^{1} < \mathtt{e}' >}{< \mathtt{if\ e\ stm\ else\ stm}' > \overset{l}{---\!>}_{\mathtt{p}}{}^{1} < \mathtt{if\ e}'\mathtt{stm\ else\ stm}' >}$$

TIf2 *If the expression value is* true*, the first list of statements is executed.*

$$\frac{}{< \mathtt{if\ true\ stm\ else\ stm}' > \overset{\tau}{---\!>}_{\mathtt{p}}{}^{1} < \mathtt{stm} >}$$

TIf3 *Otherwise it is executed the second one*

$$\frac{}{< \mathtt{if\ false\ stm\ else\ stm}' > \overset{\tau}{---\!>}_{\mathtt{p}}{}^{1} < \mathtt{stm}' >}$$

- RETURN

TReturn1 *The returned expression is evaluated.*

$$\frac{< \mathtt{e} > \overset{l}{--->}_{\mathtt{p}}{}^{1} < \mathtt{e}' >}{< \mathtt{return\ e} > \overset{l}{--->}_{\mathtt{p}}{}^{1} < \mathtt{return\ e}' >}$$

TReturn2 *The value of the expression is returned; the local memory is popped up from the stack; if a method invocation is used as a statement and returns a value, then such value is ignored: we assume that* $\mathtt{v};\mathtt{stm} = \mathtt{stm}$.

$$\frac{}{< \mathtt{return\ v}, lmS > \overset{\tau}{--->}_{\mathtt{p}}{}^{1} < \mathtt{v}, Pop(lmS) >}$$

if v is r_ground

TReturn3 *No value is returned if there is no expression to be evaluated; the local memory is popped up from the stack.*

$$\frac{}{< \mathtt{return}, lmS > \overset{\tau}{--->}_{\mathtt{p}}{}^{1} < \mathtt{skip}, Pop(lmS) >}$$

• Block

TBlock *The thread executes a statement inside a block.*

$$\frac{< \mathtt{stm}, m, lmS > \overset{l}{--->}_{\mathtt{p}}{}^{1} < \mathtt{stm}', m', lmS' >}{< \{\mathtt{stm}\}, m, lmS > \overset{l}{--->}_{\mathtt{p}}{}^{1} < \{\mathtt{stm}'\}, m', lmS' >}$$

6.4.4.0.8 Thread control statements

For simplicity, we omit the annotations of the control statements (besides, they are instance methods, so to resolve them it is not necessary to consider static information). In the following, a *running* thread is a thread that has already been started and can execute its run method; a *pre-stopped* thread is one that received a stop signal before being started.

• Start

callerSTART1 *A thread tries to start itself: it is ignored.*

$$\frac{}{< t_{id}, t_{id}.\mathtt{start()} > \xrightarrow{\tau}_{\mathtt{p}}{}^{1} < t_{id}, \mathtt{skip} >}$$

callerSTART2 *A thread sends a* `start` *message to another one.*

$$\frac{}{< t_{id}, t_{id}'.\mathtt{start()} > \xrightarrow{Send(StartT, t_{id}')}_{\mathtt{p}}{}^{1} < t_{id}, \mathtt{skip} >} \quad \textbf{if} \;\; t_{id}' \neq t_{id}$$

receiverSTART1 *A started or stopped thread does not change its state when it receives a start signal.*

$$\frac{}{< t_{id}, tSt > \xrightarrow{Receive(StartT, t_{id})}_{\mathtt{p}}{}^{1} < t_{id}, tSt >} \quad \textbf{if} \;\; tSt \notin \{New, PreStop\}$$

receiverSTART2 *A non-started but pre-stopped thread stops immediately when it receives a start signal.*

$$\frac{}{< t_{id}, PreStop > \xrightarrow{Receive(StartT, t_{id})}_{\mathtt{p}}{}^{1} < t_{id}, Stop >}$$

receiverSTART3 *A non-started thread becomes runnable.*

$$< t_{id}, \mathtt{no\_stat}, New, \mathsf{emptyst} > \xrightarrow{Receive(StartT, t_{id})}_{\mathtt{p}}{}^{1} < t_{id}, Code(\mathsf{mst}), Run, Loc(\mathsf{mst}, emptyl, t_{id}) >$$

if $\mathsf{mst} = MethodDec(\mathtt{run}, \mathrm{emptyl}, Class(t_{id}), \mathtt{p})$

• Stop

callerSTOP1 *A thread stops itself.*

$$\frac{}{< t_{id}, t_{id}.\mathtt{stop()}, Run > \xrightarrow{\tau}_{\mathtt{p}}{}^{1} < t_{id}, \mathtt{skip}, Stop >}$$

callerSTOP2 *A thread sends a* `stop` *message to another thread.*

$$\frac{}{< t_{id}, {t_{id}}'.\mathtt{stop()} > \xrightarrow{Send(Stop, {t_{id}}')}_{\mathtt{p}}{}^{1} < t_{id}, \mathtt{skip} >} \quad \textbf{if} \;\; {t_{id}}' \neq t_{id}$$

receiverSTOP1 *A non-started thread becomes pre-stopped when it receives a stop signal.*

$$\frac{}{< t_{id}, New > \xrightarrow{Receive(Stop, t_{id})}_{\mathtt{p}}{}^{1} < t_{id}, PreStop >}$$

receiverSTOP2 *A pre-stopped thread does not change its state when it receives a stop signal.*

$$\frac{}{< tid, PreStop > \xrightarrow{Receive(Stop, t_{id})}_{\mathtt{p}}{}^{1} < tid, PreStop >}$$

receiverSTOP3 *A started or stopped thread is stopped when it receives a stop signal.*

$$\frac{}{< tid, \mathtt{stm}, tSt > \xrightarrow{Receive(Stop, t_{id})}_{\mathtt{p}}{}^{1} < tid, \mathtt{no\_stat}, Stop >}$$

if $tSt \notin \{PreStop, New\}$

- SUSPEND

callerSUSPEND1 *A thread suspends itself.*

$$\frac{}{< t_{id}, t_{id}.\mathtt{suspend()}, Run > \xrightarrow{\tau}_{\mathtt{p}}{}^{1} < t_{id}, \mathtt{skip}, Susp >}$$

callerSUSPEND2 *A thread sends a* `suspend` *message to another thread.*

$$\frac{}{< t_{id}, {t_{id}}'.\mathtt{suspend()} > \xrightarrow{Send(Suspend, {t_{id}}')}_{\mathtt{p}}{}^{1} < t_{id}, \mathtt{skip} >} \quad \textbf{if} \;\; {t_{id}}' \neq t_{id}$$

receiverSUSPEND1 *A non-running thread does not change its state when it receives a suspend signal.*

$$\frac{}{< t_{id}, tSt > \xrightarrow{Receive(Suspend, t_{id})}_{\mathtt{p}}{}^{1} < t_{id}, tSt >} \quad \textbf{if} \;\; tSt \neq Run$$

receiverSUSPEND2 *A running thread becomes suspended when it receives a suspend signal.*

$$\frac{}{< t_{id}, Run > \xrightarrow{Receive(Suspend, t_{id})}_{\mathtt{p}}{}^{1} < t_{id}, Susp >}$$

- RESUME

callerRESUME1 *A thread ignores a* `resume` *message for itself.*

$$\frac{}{< t_{id}, t_{id}.\mathtt{resume()} > \xrightarrow{\tau}_{\mathtt{p}}{}^{1} < t_{id}, \mathtt{skip} >}$$

callerRESUME1 *A thread sends a* `resume` *message to another thread.*

$$\frac{}{< t_{id}, t_{id}'.\mathtt{resume()} > \xrightarrow{Send(Resume, t_{id}')}_{\mathtt{p}}{}^{1} < t_{id}, \mathtt{skip} >} \quad \textbf{if} \;\; t_{id}' \neq t_{id}$$

receiverRESUME1 *A non-suspended thread does not change its state when it receives a resume signal.*

$$\frac{}{< t_{id}, tSt > \xrightarrow{Receive(Resume, t_{id})}_{\mathtt{p}}{}^{1} < t_{id}, tSt >} \quad \textbf{if} \;\; tSt \neq Susp$$

receiverRESUME2 *A suspended thread becomes running again when it receives a resume signal.*

$$\frac{}{< t_{id}, Susp > \xrightarrow{Receive(Resume, t_{id})}_{\mathtt{p}}{}^{1} < t_{id}, Run >}$$

6.4.5 Classes and objects states

We describe classes in the program states, since class fields can be accessed by any instance of the class, but there exists exactly one incarnation [4] of it: in order to preserve its consistency the value can be read and changed only by messages sent to the class. Class and object states are simpler then threads' ones, since they lacks information related to code execution and just contain information on the identity (identifier and class name for an object or just the class name for a class state) and on the value of the instance (for an object) or the class (for a class) fields.

6.4.5.0.9 Object States $\mathsf{O\_states}_1 = \mathsf{OB\_ids} \times \mathsf{MEMORY}$ is the set of object states, ranged over by metavariable ocomp.

An object state $<o_{id}, m>$ consists of:

- o_{id}: the object identifier (containing also the class name);
- m: the memory.

6.4.5.0.10 Class States $\mathsf{C\_states}_1 = \mathsf{CL\_ids} \times \mathrm{Set}(\mathsf{OB\_ids}) \times \mathsf{MEMORY} \times \mathsf{MEMORY}$ is the set of class states, ranged over by metavariable $cc \in \mathsf{C\_states}$.

A class state $<c_{id}, S_{id}, lm, m>$ consists of:

- c_{id}: the class identifier;
- S_{id}: the set of the identifiers of the previously created objects of this class;
- lm: the initial memory for an instance of the class, used for creating new instances;
- m: the memory.

6.4.6 Program Semantics

The semantics of programs is given by the *lts*: $\mathsf{P\_LTS} = (\mathsf{P\_states}_1, \mathsf{P\_labels}_1, \longrightarrow_{\mathrm{P}}{}^1)$.

[4] It is part of the Java terminology, as used in [J.G96]

6.4.6.0.11 States

A program state consists of a set of thread and passive component states (that is a set of object and class states).

$\mathsf{P\_states}_1 = \mathrm{Set}(\mathsf{T\_states}_1 \cup \mathsf{Pass\_states}_1)$

In the following:

- $\mathsf{Pass\_states}_1 = \mathsf{O\_states}_1 \cup \mathsf{C\_states}_1$

  is the set of all the passive component states, ranged over by metavariable pc;

- $\mathsf{NonClass\_states}_1 = \mathsf{O\_states}_1 \cup \mathsf{T\_states}_1$

  is the set of non-class components, ranged over by nc;

- $\mathsf{G\_states}_1 = \mathsf{Pass\_states}_1 \cup \mathsf{T\_states}_1$

  is the set of all the component states, ranged over by metavariable gc. We use the short notation $gc_1| \ldots | gc_n$for $\{gc_1, \ldots, gc_n\}$.

  Sometime we also use the short notation $gc_1| \ldots |gc_j|\mathsf{S}$(where $\mathsf{S} \in \mathrm{Set}(\mathsf{G\_states}_1)$ denotes a generic set of both active and passive components), when we want to put more emphasis only some components, mixing together the other active and passive ones.

  (From now on, for simplicity, we forget about the "1" from the previous sets.)

6.4.6.0.12 Auxiliary operations

Auxiliary operations on program states are:

- $IdInfo$: $\mathsf{O\_states} \rightarrow \mathsf{OB\_ids}$

  defined by: $IdInfo(<o_{id}, m>) = o_{id}$;

  returns the identifier of the object component (analogous for thread and class components);

- Mem: $\mathsf{O\_states} \rightarrow \mathsf{MEMORY}$

  defined by: $Mem(<o_{id}, m>) = m$

  returns the memory of the object component (analogous for class and thread components);

- tSt: $\mathsf{T\_states} \rightarrow \mathsf{Status}$

  defined by: $tSt(<g_{id}, \mathtt{stm}, tSt, m, lmS>) = tSt$;

  returns the status of a thread;

- $OidSet$: $\mathsf{C\_states} \to \mathcal{S}\rceil\sqcup(\mathsf{OB\_ids})$

  defined by: $OidSet(<c_{id},S_{id},lm,m>)= S_{id}$

  returns the list of instances of a class;

- $NewMem$: $\mathsf{C\_states} \to \mathsf{Locals}$

  defined by: $NewMem(<\mathtt{C},S_{id},m,\ lm>)= lm$

  returns the initial memory of a class instance.

6.4.6.0.13 $\mathsf{P\_labels}_1 \quad =\{\tau\}$

The labels describe the information made available to an external observer at each step of the computation, i.e., every kind of interaction a Java program can have with the external environment. Since Java_1 does not contain I-O operations, we can have only the τ (usually omitted) label describing transitions without information exchange with the environment.

6.4.6.0.14 Transition relation $\longrightarrow_{\mathtt{P}}{}^1$ is defined by the rules below. Any program transition result by combining one or more transitions of the active components.

- Internal Action

PInternal *An internal transition of a thread is an internal transition of the whole program.*

$$\frac{tc \xrightarrow{\tau}_{\mathtt{P}}{}^1 tc'}{tc|\mathsf{S} \xrightarrow{\tau}_{\mathtt{P}}{}^1 tc'|\mathsf{S}}$$

- New

PNew *If a thread calls the new method of a class, the returned instance is added to the program state.*

$$\frac{tc \xrightarrow{New(\mathtt{C},nc_{id})}_{\mathtt{P}}{}^1 tc'}{tc| < \mathtt{C}, lm, S_{id}, m > |\mathsf{S} \xrightarrow{\tau}_{\mathtt{P}}{}^1 tc'| < \mathtt{C}, lm, \{nc_{id}\} \cup S_{id}, m > | < nc_{id}, lm > |\mathsf{S}}$$

if $nc_{id} \notin S_{id}$ and $Class(nc_{id})= \mathtt{C}$

- Method Call

PCall1 *A thread calls a method of an existing object of the program; the thread component is modified.*

$$\frac{tc \overset{CallO(nc_{id},\mathtt{m},\mathtt{args},MethodDec(\mathtt{m},\mathtt{args},Class(\mathsf{ocomp}),\mathtt{p}))}{\dashrightarrow}{}_{\mathtt{p}}{}^{1}\ tc'}{tc|\mathsf{ocomp}|\mathsf{S} \xrightarrow{\tau}{}_{\mathtt{p}}{}^{1}\ tc'|\mathsf{ocomp}|\mathsf{S}}$$

if $IdInfo(\mathsf{ocomp}) = nc_{id}$

PCall2 *A thread calls a method of a class of the program; the thread component of the program is modified* [5].

$$\frac{tc \overset{CallC(\mathtt{C},\mathtt{m},\mathtt{args})}{\dashrightarrow}{}_{\mathtt{p}}{}^{1}\ tc'}{tc|\mathsf{S} \xrightarrow{\tau}{}_{\mathtt{p}}{}^{1}\ tc'|\mathsf{S}}$$

PCall3 *A thread calls one of its methods: the thread component of the program is modified.*

$$\frac{tc \overset{SelfCall(\mathtt{m},\mathtt{args})}{\dashrightarrow}{}_{\mathtt{p}}{}^{1}\ tc'}{tc|\mathsf{S} \xrightarrow{\tau}{}_{\mathtt{p}}{}^{1}\ tc'|\mathsf{S}}$$

- FIELD ACCESS

PGet1 *A thread reads the value of an object field*

$$\frac{tc_1 \overset{GetO(nc_{id},\mathtt{T},\mathtt{f},\mathsf{v})}{\dashrightarrow}{}_{\mathtt{p}}{}^{1}\ tc_1'}{tc_1|< nc_{id}, m >|\mathsf{S} \xrightarrow{\tau}{}_{\mathtt{p}}{}^{1}\ tc_1'|< nc_{id}, m >|\mathsf{S}}$$

if $m(\mathtt{f},\mathtt{T}) = \mathsf{v}$

PGet2 *A thread reads the value of a class field*

$$\frac{tc_1 \overset{GetC(\mathtt{C},\mathtt{T},\mathtt{f},\mathsf{v})}{\dashrightarrow}{}_{\mathtt{p}}{}^{1}\ tc_1'}{tc_1|< c_{id}, S_{id}, lm, m >|\mathsf{S} \xrightarrow{\tau}{}_{\mathtt{p}}{}^{1}\ tc_1'|< c_{id}, S_{id}, lm, m >|\mathsf{S}}$$

if $m(\mathtt{f},\mathtt{T}) = \mathsf{v}$

PGet3 *A thread reads the value of a field of another thread*

$$\frac{tc \overset{GetC(\mathtt{C},\mathtt{T},\mathtt{f},\mathsf{v})}{\dashrightarrow}{}_{\mathtt{p}}{}^{1}\ tc'}{tc|< t_{id}, \mathtt{stm}, tSt, m, lmS >|\mathsf{S} \xrightarrow{\tau}{}_{\mathtt{p}}{}^{1}\ tc'|< t_{id}, \mathtt{stm}, tSt, m, lmS >|\mathsf{S}}$$

if $m(\mathtt{f},\mathtt{T}) = \mathsf{v}$

[5] These two labels will be more useful in Java_2 when a call of a synchronized method cannot be treated as an internal action of the thread

PAss1 *A thread writes the value of an object field*

$$\frac{tc \overset{AssO(nc_{id},\mathtt{T},\mathtt{f},\mathsf{v})}{----------}>_{\mathtt{p}}{}^{1}\, tc'}{tc| < nc_{id}, m > |\mathsf{S} \overset{\tau}{\longrightarrow}_{\mathtt{p}}{}^{1}\, tc'| < nc_{id}, m' > |\mathsf{S}}$$

if $m' = m[(\mathtt{f},\mathtt{T}) \rightarrow \mathsf{v}]$

PAss2 *A thread writes the value of a class field*

$$\frac{tc \overset{AssC(\mathtt{C},\mathtt{T},\mathtt{f},\mathsf{v})}{---------}>_{\mathtt{p}}{}^{1}\, tc'}{tc| < c_{id}, S_{id}, lm, m > |\mathsf{S} \overset{\tau}{\longrightarrow}_{\mathtt{p}}{}^{1}\, tc'| < c_{id}, S_{id}, lm, m' > |\mathsf{S}}$$

if $m' = m[(\mathtt{f},\mathtt{T}) \rightarrow \mathsf{v}]$

PAss3 *A thread writes the value of another thread field*

$$\frac{tc \overset{GetC(\mathtt{C},\mathtt{T},\mathtt{f},\mathsf{v})}{---------}>_{\mathtt{p}}{}^{1}\, tc'}{tc| < t_{id}, \mathtt{stm}, tSt, m, lmS > |\mathsf{S} \overset{\tau}{\longrightarrow}_{\mathtt{p}}{}^{1}\, tc'| < t_{id}, \mathtt{stm}, tSt, m', lmS > |\mathsf{S}}$$

if $m' = m[(\mathtt{f},\mathtt{T}) \rightarrow \mathsf{v}]$

- Thread Control

PThreadC *A thread sends a control signal to another thread that receives it. In the new program state both the thread components are updated.*

$$\frac{tc_1 \overset{Send(l,\mathrm{t}_{id_2})}{------}>_{\mathtt{p}}{}^{1}\, tc_1{}' \quad tc_2 \overset{Receive(l,\mathrm{t}_{id_2})}{---------}>_{\mathtt{p}}{}^{1}\, tc_2{}'}{tc_1|tc_2|\mathsf{S} \overset{\tau}{\longrightarrow}_{\mathtt{p}}{}^{1}\, tc_1{}'|tc_2{}'|\mathsf{S}}$$

if $IdInfo(tc_2) = \mathrm{t}_{id_2}$

6.5 Java$_2$

In this section we give the static and dynamic semantics of a richer sublanguage (Java$_2$) obtained by adding the *method synchronization* and thus the methods `wait`, `join`, `notify`, `notifyAll` and the `synchronized` statement the to Java$_1$. Both the semantics are given by extending the rules defining the semantics of Java$_1$.

6.5.1 Syntax and Static Semantics

The statement syntax of Java$_2$ is the one of Java$_1$ enriched by:
Statement:

SynchronizedStatement

SynchronizedStatement:

`synchronized(`*Expression*`)` *Block*

The *MethodModifier* rule is replaced by
MethodModifier:
`st , nst , sy`

The type inference system is enriched by the rules for the blocks and for the synchronization statement, reported below:

- **Block** *the statements inside a block are decorated considering also the local environment*

$$\frac{\rho_l;\rho \vdash \texttt{stm} : \texttt{T}}{\rho \vdash \{[\rho_l]\texttt{stm}\} : \texttt{T}}$$
$$A\{\{[\rho_l]\texttt{stm}\},\rho\} = \{[\rho_l]A\{\texttt{stm},\rho_l;\rho\}\}$$

- **SyncStat**

$$\frac{\rho_l;\rho \vdash \texttt{stm} : \texttt{T}}{\rho \vdash \texttt{synchronized(e)}\{[\rho_l]\texttt{stm}\} : \texttt{T}}$$
$$A\{\texttt{synchronized(e)}\{[\rho_l]\texttt{stm}\},\rho\} = \texttt{synchronized}(A\{\texttt{e},\rho\})\{A\{[\rho_l]\texttt{stm},\rho\}\}$$

The rule `MethodCall` is left unchanged, but now the value of `mod` can also be `sy`.

6.5.2 Semantics

Java_2 differs from Java_1, since a (instance or class) method can be *synchronized*. This means that just one thread at a time can execute such method. In order to prevent multiple calls to synchronized methods from different threads, a *lock* is associated with each object or class. Any thread must acquire such lock before starting the execution of a synchronized method. If an object (or class) has more than one synchronized method, then just one of its methods may be executed at a given time. A thread can obtain many times the lock for the same object, so, if it performs n *locking* operations on a lock, then it has to perform n *unlocking* operations before relinquishing it. Every object, class or threads keeps track of the `lock` actions performed by threads on its own lock, so that it is possible to control the correct matching among `unlock` and `lock` actions.

6.5.2.1 Thread Semantics

The semantics of the threads in Java_2 is given by an *lts*

$\mathsf{T\_LTS}_2 = (\mathsf{T\_states}_2, \mathsf{T\_labels}_2, \ --->_{\mathtt{P}}{}^2\ )$.

6.5.2.1.1 States

$\mathsf{T\_states}_2 = \mathsf{TH\_ids} \times \mathsf{STATEMENT} \times \mathsf{Status}_2 \times \mathsf{MEMORY} \times \mathsf{Stacks} \times \mathsf{Locks}$

The set of the status elements is modified:

$\mathsf{Status}_2 = \mathsf{Status} \cup (\{Awaken\} \times \mathsf{COMP\_ids}) \cup (\{Relin\} \times \mathrm{List}(\mathsf{COMP\_ids} \times \mathsf{INTEGER}) \times \mathsf{Status})$

$\cup\ (\{Wait\} \times \mathsf{COMP\_ids} \times \mathsf{INTEGER})$:

- $Awaken(g_{id})$ says the thread has been notified but it is still not running; it has to acquire the lock for g_{id} again.
- $Relin(ll,s)$ says the thread has to relinquish locks listed in ll before becoming s again.
- $Wait(g_{id},n)$ says the thread has been suspended on object g_{id} while executing a synchronized method; it has to perform n `lock` operations on the lock of g_{id} before being running again

$\mathsf{Locks} = (\mathsf{TH\_ids} \times \mathsf{NAT}) \cup \{none\}$

is the set of the lock information; ranged over by *lock* metavariable. *none* is the value denoting that no thread has locked the object.

6.5.2.1.2 Labels

Some new labels are added to the set $\mathsf{T\_labels}_1$

$\mathsf{T\_labels}_2 = \mathsf{T\_labels}_1 \cup$

$(\{\mathit{Wait}, \mathit{Notify}, \mathit{Notified}, \mathit{Lock}, \mathit{Unlock}\} \times \mathsf{TH\_ids} \times \mathsf{COMP\_ids}) \cup$

$(\{\mathit{NotifyA}\} \times \mathsf{COMP\_ids})$:

- $\mathit{Wait}(t_{id}, g_{id})$

  t_{id} executes the `wait` statement in the body of a method of g_{id} and becomes waiting for it;

- $\mathit{Notify}(t_{id}, t_{id}'), \mathit{NotifyA}(t_{id}, g_{id})$

  t_{id} executes the `notify` and `notifyAll` statements and it sends a signal to t_{id}' or to every thread waiting for the lock of g_{id};

- $\mathit{Notified}(t_{id}, t_{id}')$

  t_{id} communicates it received the notify signal from t_{id}' and it is no more waiting for an object;

- $\mathit{Lock}(t_{id}, g_{id})$

  t_{id} performs a lock action on g_{id};

- $\mathit{Unlock}(t_{id}, g_{id})$ t_{id} performs an unlock action on g_{id};

6.5.2.1.3 Transition relation

The transition relation $---\!>_{\mathtt{p}}{}^2$ is defined by the rules below.

The transitions defined for Java_1 are still transitions for Java_2, except for method calls and `load`. A call to a synchronized method can occur only if the thread obtains the lock; the `load` occurring while the thread is executing a synchronized method is to be distinguished from the other `load` occurrences, since they can occur only if the corresponding `read` have occurred before, inside the same synchronized block.

Finally, the method structures returned after a method call are slightly modified to indicate if the method is synchronized or not: $\mathsf{MET\_str} = \mathsf{FORMAL\_PAR\_LIST} \times \mathsf{STATEMENT} \times \mathsf{Locals} \times \mathsf{Bool}$.

TOld1 *The transition defined for Java_1that do not contain method calls, load, stop actions or the receiving of a signal from another thread are transitions of the Java_2 programs too.*

$$\frac{tc \overset{l}{\dashrightarrow}_{\mathtt{p}}{}^{1}\, tc'}{tc \overset{l}{\dashrightarrow}_{\mathtt{p}}{}^{2}\, tc'}$$

if $l \notin \{CallO(\ldots), CallC(\ldots), SelfCall(\ldots), Load(\ldots), Receive(\ldots)\}$

- SYNCHRONIZED

TSynch1 *A synchronized statement is executed by first evaluating the expression.*

$$\frac{< \mathtt{e} > \overset{l}{\dashrightarrow}_{\mathtt{p}}{}^{2} < \mathtt{e}' >}{< \mathtt{synchronized\ (e)}\{[L]\mathtt{stm}\} > \overset{l}{\dashrightarrow}_{\mathtt{p}}{}^{2} < \mathtt{synchronized\ (e')}\{[L]\mathtt{stm}\} >}$$

TSynch2 *The executing thread takes the lock of the object referenced by the value of the expression and then executes the block; the working memory is empty and a new local memory is pushed into the stack.*

$$\frac{}{< \mathtt{synchronized}\ (g_{id})\{[L]\mathtt{stm}\}, lmS > \overset{Lock(t_{id},g_{id})}{\dashrightarrow}_{\mathtt{p}}{}^{2} < \{\mathtt{stm}\}^{g_{id}}, Push(L, lmS) >}$$

- METHOD CALLS

TLock1 *A call to a static synchronized method requires a lock action on the lock of the class. Such action occurs together with the method call (that is an internal action and does not appear on the label).*

$$\frac{< g_{id}.[\mathtt{C}, \mathsf{args}, \mathsf{st}]\mathtt{m}(\mathsf{vals}), lmS > \overset{CallC(\mathtt{C},\mathtt{m},\mathsf{args})}{\dashrightarrow}_{\mathtt{p}}{}^{1} < \{\mathtt{stm}'\}, lmS' >}{< g_{id}.[\mathtt{C}, \mathsf{args}, \mathsf{st}; \mathsf{sy}]\mathtt{m}(\mathsf{vals}), lmS > \overset{Lock(t_{id},\mathtt{C})}{\dashrightarrow}_{\mathtt{p}}{}^{2} < \{\mathtt{stm}'\}^{\mathtt{C}}, lmS' >}$$

TNonSynch1 *A call to a static and non-synchronized method has the semantics defined in* $Java_1$*; the transition becomes an internal action of the thread.*

$$\frac{< g_{id}.[\mathtt{T}, \mathsf{args}, \mathsf{st}]\mathtt{m}(\mathsf{vals}), lmS > \overset{CallC(\mathtt{T},\mathtt{m},\mathsf{args})}{\dashrightarrow}_{\mathtt{p}}{}^{1} < \{\mathtt{stm}'\}, lmS' >}{< g_{id}.[\mathtt{T}, \mathsf{args}, \mathsf{st}]\mathtt{m}(\mathsf{vals}), lmS > \overset{\tau}{\dashrightarrow}_{\mathtt{p}}{}^{2} < \{\mathtt{stm}'\}, lmS' >}$$

TLock2 *A call to a non-static and synchronized method requires a lock action on the lock of the object. Such action occurs together with the method call (that is an internal action and does not appear on the label).*

$$\frac{< nc_{id}.[\mathtt{T},\mathtt{args},\mathtt{nst};\mathtt{sy}]\mathtt{m}(\mathsf{vals}), lmS > \xrightarrow{CallO(nc_{id},\mathtt{m},\mathtt{args},\mathtt{mst})}{}_{\mathtt{p}}{}^{1} < \{\mathtt{stm}\}, lmS' >}{< nc_{id}.[\mathtt{T},\mathtt{args},\mathtt{nst}]\mathtt{m}(\mathsf{vals}), lmS > \xrightarrow{Lock(t_{id},nc_{id})}{}_{\mathtt{p}}{}^{2} < \{\mathtt{stm}\}^{nc_{id}}, lmS' >}$$

TNonSynch2 *A call to a non-static and non-synchronized method becomes an internal action of the thread.*

$$\frac{< nc_{id}.[\mathtt{T},\mathtt{args},\mathtt{nst}]\mathtt{m}(\mathsf{vals}), lmS > \xrightarrow{CallO(nc_{id},\mathtt{m},\mathtt{args},\mathtt{mst})}{}_{\mathtt{p}}{}^{1} < \{\mathtt{stm'}\}, lmS' >}{< nc_{id}.[\mathtt{T},\mathtt{args},\mathtt{nst}]\mathtt{m}(\mathsf{vals}), lmS > \xrightarrow{\tau}{}_{\mathtt{p}}{}^{2} < \{\mathtt{stm'}\}, lmS' >}$$

TLock3 *A call to a not static and synchronized method requires a lock operation on the lock of the thread itself.*

$$\frac{< t_{id}.[\mathtt{T},\mathtt{args},\mathtt{nst}]\mathtt{m}(\mathsf{vals}), lmS > \xrightarrow{SelfCall(\mathtt{T},\mathtt{m},\mathtt{args})}{}_{\mathtt{p}}{}^{1} < \{\mathtt{stm}\}, lmS' >}{< t_{id}.[\mathtt{T},\mathtt{args},\mathtt{nst};\mathtt{sy}]\mathtt{m}(\mathsf{vals}), lmS > \xrightarrow{Lock(t_{id},t_{id})}{}_{\mathtt{p}}{}^{2} < \{\mathtt{stm}\}^{t_{id}}, lmS' >}$$

TNonSynch3 *A call to a not static and not synchronized method is an internal action of the thread*

$$\frac{< t_{id}.[\mathtt{T},\mathtt{args},\mathtt{nst}]\mathtt{m}(\mathsf{vals}), lmS > \xrightarrow{SelfCall(\mathtt{m},\mathtt{args})}{}_{\mathtt{p}}{}^{1} < \{\mathtt{stm'}\}, lmS' >}{< t_{id}.[\mathtt{T},\mathtt{args},\mathtt{nst}]\mathtt{m}(\mathsf{vals}), lmS > \xrightarrow{\tau}{}_{\mathtt{p}}{}^{2} < \{\mathtt{stm'}\}, lmS' >}$$

Analogous rules if the method is static and synchronized and static and not synchronized.

- WAIT

TWait *A thread becomes waiting for an object and must relinquish n-time the g_{id} locks it owns; the number of unlock operations to be performed on g_{id} is recorded in the component itself.*

$$< g_{id}.\mathtt{wait()}, Run > \xrightarrow{Wait(g_{id},n)}{}_{\mathtt{p}}{}^{2} < \mathtt{skip}, Relin((g_{id},n), Wait(g_{id},n)) >$$

- NOTIFY, NOTIFY ALL

TNotify1 *A thread can send a* `notify` *signal to another thread.*

$$< g_{id}.\mathtt{notify()} > \xrightarrow{Notify(t_{id}',g_{id})}{}_{\mathtt{p}}{}^{2} < \mathtt{skip} >$$

TNotify2 *A thread can send a* `notifyAll` *signal to all the threads waiting for the object's lock.*

$$< g_{id}.\mathtt{notifyAll()} > \xrightarrow{NotifyA(g_{id})}_{\mathtt{p}}{}^{2} < \mathtt{skip} >$$

TNotify3 *A thread can receive a notify signal becoming awaken.*

$$< Wait(g_{id}, n) > \xrightarrow{Notified(t_{id}, g_{id})}_{\mathtt{p}}{}^{2} < Awaken(g_{id}, n) >$$

• Lock

TLock5 *after a thread becomes awaken, it tries to acquire the locks it needs*

$$< t_{id}, Awaken(g_{id}, n) > \xrightarrow{Lock(t_{id}, g_{id})}_{\mathtt{p}}{}^{2} < t_{id}, Awaken(g_{id}, n-1) > \quad \textbf{if} \quad n > 1$$

TLock6 *when it acquires all the locks again, it becomes running.*

$$< t_{id}, Awaken(g_{id}, 1) > \xrightarrow{Lock(t_{id}, g_{id})}_{\mathtt{p}}{}^{2} < t_{id}, Run >$$

• Join

TJoin *A thread executes a* `join()` *statement for a thread and becomes waiting.*

$$< t_{id}'.\mathtt{join()}, Run > \xrightarrow{\tau}_{\mathtt{p}}{}^{2} < \mathtt{skip}, Wait(t_{id}', 1) >$$

• Stop

TStop1 *If the thread is not started, it ignores a stop signal*

$$\frac{tc \xrightarrow{Receive(Stop, t_{id})}_{\mathtt{p}}{}^{1} tc'}{tc \xrightarrow{Receive(Stop, t_{id})}_{\mathtt{p}}{}^{2} tc'} \quad \textbf{if} \quad tSt(tc) \in \{PreStop, New, \ Stop\}$$

TStop2 *If the thread is started, it starts relinquishing every locks it owns*

$$\frac{< \mathtt{stm}, Running > \xrightarrow{Receive(Stop, t_{id})}_{\mathtt{p}}{}^{1} < \mathtt{no\_stat}, Stop >}{< \mathtt{stm}, Running > \xrightarrow{Receive(Stop, t_{id}, LocksList)}_{\mathtt{p}}{}^{2} < \mathtt{no\_stat}, Relin(LocksList, Stop) >}$$

if $tSt \notin \{PreStop, New\}$

TUnlock1 *A thread start relinquishing the locks it owns.*

$$\frac{}{< Relin((g_{id}, n); ll, tSt) > \xrightarrow{Unlock(t_{id}, g_{id})}_{\mathtt{p}}{}^{2} < Relin((g_{id}, n-1); ll, tSt) >}$$

if n > 2

TUnlock2

$$\frac{}{< Relin((g_{id}, 1); ll), tSt > \xrightarrow{Unlock(t_{id}, g_{id})}_{\mathtt{p}}{}^{2} < Relin(ll, tSt) >}$$

TUnlock3 *After having relinquished all the locks, if it had been previously stopped, the thread sends a* `notifyAll` *signal to all threads waiting for its own lock and becomes stopped.*

$$\frac{}{< Relin(emptyl, Stop) > \xrightarrow{NotifyA(tid)}_{\mathtt{p}}{}^{2} < Stop >}$$

TUnlock4 *After having relinquished all the locks, the thread becomes waiting*

$$\frac{}{< Relin((g_{id}, 1), Wait(g_{id}, m)) > \xrightarrow{Unlock(t_{id}, g_{id})}_{\mathtt{p}}{}^{2} < Wait(g_{id}, m) >}$$

- UNLOCK

TUnlock5 *When the synchronized block is finished, the thread performs an unlock action*

$$\frac{}{< t_{id}, \{\mathtt{skip}\}^{g_{id}} > \xrightarrow{Unlock(t_{id}, g_{id})}_{\mathtt{p}}{}^{2} < t_{id}, \mathtt{skip} >}$$

6.5.3 Program Semantics

The semantics of Java_2 programs is given by the lts

$\mathsf{P\_LTS}_2 = (\mathsf{P\_states}_2, \mathsf{P\_labels}_1, \longrightarrow_{\mathtt{p}}{}^{2})$.

6.5.3.0.4 States

The passive component states are enriched with the information about the status of the lock.

6.5.3.0.5 Object States

$\mathsf{O\_states}_2 = \mathsf{O\_states}_1 \times \mathsf{Locks}$ is the set of object states, ranged over by the metavariable ocomp.

An object state in Java_2 is $<o_{id}, m, \mathit{lock}>$ where *lock* is the information about the status of the lock of the object.

6.5.3.0.6 Class States

$\mathsf{C\_states}_2 = \mathsf{C\_states}_1 \times \mathsf{Locks}$ is the set of class states, ranged over by metavariable $cc \in \mathsf{C\_states}$.

A class state in Java_2 is $<c_{id}, S_{id}, lm, m, \mathit{lock}>$ where *lock* is the information about the status of the lock of the object.

6.5.3.0.7 Labels

The labels for the *lts* of a program are left unchanged w.r.t. Java_1 semantics

6.5.3.0.8 Transition relation

$\longrightarrow_{\mathrm{P}}{}^2$ extends the $\longrightarrow_{\mathrm{P}}{}^1$ relation as defined for Java_1 on the new states.

- Method Call

Non synchronized method calls are internal actions, thus covered by **PInternal** rule of Java_1.
Synchronized method calls are labelled by lock label, and then treated by the **Plock** rules below.

- Lock and Unlock actions

PLock1 *A thread performs a lock operation on an object lock that it already owns.*

$$\frac{tc \xrightarrow{Lock(t_{id}, o_{id})}_{\mathrm{P}}{}^2\, tc'}{tc| < o_{id}, m, (t_{id}, n) > |\mathsf{S} \xrightarrow{\tau}_{\mathrm{P}}{}^2\, tc'| < o_{id}, m, (t_{id}, n+1) > |\mathsf{S}}$$

PLock2 *A thread performs a lock operation on an object lock that it is not assigned to any thread.*

$$\frac{tc \xrightarrow{Lock(t_{id}, g_{id})}_{\mathtt{P}}{}^{2}\ tc'}{tc| < o_{id}, m, none > |\mathsf{S} \xrightarrow{\tau}_{\mathtt{P}}{}^{2}\ tc'| < o_{id}, m, (t_{id}, 1) > |\mathsf{S}}$$

PLock3 *A thread performs a lock operation on a class lock that it already owns.*

$$\frac{tc \xrightarrow{Lock(t_{id}, \mathtt{C})}_{\mathtt{P}}{}^{2}\ tc'}{tc| < \mathtt{C}, S_{id}, lm, m, (t_{id}, n) > |\mathsf{S} \xrightarrow{\tau}_{\mathtt{P}}{}^{2}\ tc'| < \mathtt{C}, S_{id}, lm, m, (t_{id}, n+1) > |\mathsf{S}}$$

PLock4 *A thread performs a lock operation on a class lock that it is not assigned to any thread.*

$$\frac{tc \xrightarrow{Lock(t_{id}, \mathtt{C})}_{\mathtt{P}}{}^{2}\ tc'}{tc| < \mathtt{C}, S_{id}, lm, m, none > |\mathsf{S} \xrightarrow{\tau}_{\mathtt{P}}{}^{2}\ tc'| < \mathtt{C}, S_{id}, lm, m, (t_{id}, 1) > |\mathsf{S}}$$

PLock5 *A thread performs a lock operation on a thread lock that it already owns.*

$$\frac{tc \xrightarrow{Lock(t_{id}, t_{id}')}_{\mathtt{P}}{}^{2}\ tc'}{tc| < t_{id}', \mathtt{stm}, tSt, m, lmS, (t_{id}, n) > |\mathsf{S} \xrightarrow{\tau}_{\mathtt{P}}{}^{2}\ tc'| < t_{id}', \mathtt{stm}, tSt, m, lmS, (t_{id}, n+1) > |\mathsf{S}}$$

PLock6 *A thread performs a lock operation on a thread lock that it is not assigned to any thread.*

$$\frac{tc \xrightarrow{Lock(t_{id}, t_{id}')}_{\mathtt{P}}{}^{2}\ tc'}{tc| < t_{id}', \mathtt{stm}, tSt, m, lmS, none > |\mathsf{S} \xrightarrow{\tau}_{\mathtt{P}}{}^{2}\ tc'| < t_{id}', \mathtt{stm}, tSt, m, lmS, (t_{id}, 1) > |\mathsf{S}}$$

PUnlock1 *A thread performs an unlock operation over an object lock and the number of lock operations is decremented.*

$$\frac{tc \xrightarrow{Unlock(t_{id}, g_{id})}_{\mathtt{P}}{}^{2}\ tc'}{tc| < o_{id}, m, (t_{id}, n) > |\mathsf{S} \xrightarrow{\tau}_{\mathtt{P}}{}^{2}\ tc'| < o_{id}, m, (t_{id}, n-1) > |\mathsf{S}} \quad \textbf{if} \ \ n > 0$$

PUnlock2 *A thread performs the last unlock operation over an object lock, and the lock is assigned to no thread.*

$$\frac{tc \xrightarrow{Unlock(t_{id}, g_{id})}_{\mathtt{P}}{}^{2}\ tc'}{tc| < o_{id}, m, (t_{id}, 1) > |\mathsf{S} \xrightarrow{\tau}_{\mathtt{P}}{}^{2}\ tc'| < o_{id}, m, none > |\mathsf{S}}$$

Analogously we have the rule for the unlock operation over the lock of a class or thread.

• WAIT

PWait *A thread, starts the unlock of the lock operation it performed on g_{id} before becoming waiting.*

$$\frac{tc \overset{Wait(o_{id},n)}{------>}_{\mathtt{P}}{}^2\, tc'}{tc| < o_{id}, m, (t_{id}, n) > |\mathsf{S} \xrightarrow{\tau}_{\mathtt{P}}{}^2\, tc'| < o_{id}, m, (t_{id}, n-1) > |\mathsf{S}}$$

if *IdInfo*$(tc)=\ t_{id}$

$$\frac{tc \overset{Wait(g_{id},1)}{------>}_{\mathtt{P}}{}^2\, tc'}{tc| < o_{id}, m, (t_{id}, 1) > |\mathsf{S} \xrightarrow{\tau}_{\mathtt{P}}{}^2\, tc'| < o_{id}, m, none > |\mathsf{S}}$$

if *IdInfo*$(tc)=\ t_{id}$

Analogously for locks of classes or thread objects.

• NOTIFY, NOTIFY ALL

PNotify *A thread sends a notify signal to one thread waiting for a component; the thread becomes awaken.*

$$\frac{tc_1 \overset{Notify(\mathrm{t}_{id_2},g_{id})}{--------->}_{\mathtt{P}}{}^2\, tc_1{}'\ \ tc_2 \overset{Notified(\mathrm{t}_{id_2},g_{id})}{---------->}_{\mathtt{P}}{}^2\, tc_2{}'}{tc_1|tc_2|\mathsf{S} \xrightarrow{\tau}_{\mathtt{P}}{}^2\, tc_1{}'|tc_2{}'|\mathsf{S}}$$

PNotifyAll *A thread sends a* `notifyAll` *signal for an object or class and all the threads waiting for it become awaken again.*

$$\frac{tc \overset{NotifyA(g_{id})}{------->}_{\mathtt{P}}{}^2\, tc'\ \ tc_i \overset{Notified(t_{idi},g_{id})}{---------->}_{\mathtt{P}}{}^2\, tc_i{}'\ i = 1 \ldots n}{tc|tc_1, \ldots tc_n|Tset_1|\mathsf{S} \xrightarrow{\tau}_{\mathtt{P}}{}^2\, tc'|tc_1, \ldots tc_n|Tset_1|\mathsf{S}}$$

if *NoWait*$(Tset_1)$

NoWait$(Tset, g_{id})$ is true if there is no thread in $Tset$ waiting for g_{id}; [6]

[6] In our compositional approach to the semantics, every information on the state of the threads used in a transition of the program should be given by the labels of the thread transitions. To be precise, in this case there should be a transition performed by each thread that communicates whether it is waiting for a given object or not. For simplicity, we just use a predicate on a thread set that checks if the elements are threads that are not waiting for the lock of the object

• STOP

PStop1 *When a thread is stopped, it must relinquish every lock it has.*

$$\frac{tc_1 \xrightarrow{Send(Stop,\mathrm{t}_{id_1})}{}_{\mathrm{P}}^{2}\ tc_1{}' \quad tc_2 \xrightarrow{Receive(Stop,\mathrm{t}_{id_1},LocksList)}{}_{\mathrm{P}}^{2}\ tc_2{}'}{tc_1|tc_2|\mathsf{S} \xrightarrow{\tau}{}_{\mathrm{P}}^{2}\ tc_1{}'|tc_2{}'|\mathsf{S}}$$

if LocksList = LocksOwned(t_{id_1},S)

PStop2 *All the other statements within a synchronized method are executed.*

$$\frac{tc \xrightarrow{l}{}_{\mathrm{P}}^{2}\ tc'}{tc|\mathsf{S} \xrightarrow{\tau}{}_{\mathrm{P}}^{2}\ tc'|\mathsf{S}}$$

if $l \notin$ { *CallO*(...),*CallC*(...),*SelfCall*(...), *Notif*(...), *Stop*(...),*Lock*(...), *Unlock*(...)}

6.6 Java$_1$ syntax

We use the special notation for grammar introduced in [J.G96]. The definition of a non-terminal is introduced by the name of a nonterminal followed by a colon; alternative right-hand sides for the non terminal follow on succeeding lines. The subscripted suffix *opt*, which may appear after either a terminal or nonterminal, indicates an *optional symbol*.

For example:
BreakStatement:
`break` *Identifier*$_{opt}$

is a convenient abbreviation for:
BreakStatement:
`break` *Identifier*
`break`

Below is the syntax of programs and environments.

Env:
    StandardEnv
    Env, Decl
StandardEnv:

Exception text Objects,..., ρ_Φ[7]

Decl:

ClassDeclaration

VariableDeclaration

ClassDeclaration:

ClassExtDec{ *ClassDeclarations$_{opt}$*}

ClassExtDec:

*t*class *ClassName* text *ClassName*

Type:

ClassName

tbool

tchar

tint

ClassDeclarations:

ClassDeclarations ClassDecl

ClassDecl

ClassDecl:

ClassMemberDeclaration

ClassMemberDeclaration:

FieldDeclaration

MethodDeclaration

FieldDeclaration:

FieldModifier$_{opt}$ VariableDeclaration

VariableDeclaration:

Type VariableDeclaratorId

VariableDeclaratorId:

Identifier

FieldModifier:

t`st`

MethodDeclaration:

MethodHeader

MethodHeader:

MethodModifier$_{opt}$ResultType MethodDeclarator

ResultType:

Type

t`void`

MethodDeclarator:

[7]Standard Environment is essentially a non-empty list of the declarations of all the predefined classes of Java

Identifiert(*FormalParameterList*$_{opt}$ *t*)
FormalParameterList:
FormalParameterList,FormalParameter
FormalParameter:
Type VariableDeclaratorId
MethodModifier:
t`st`, `nst`

6.6.0.0.9 Programs

Programs:
ClassBodies$_{opt}$
ClassBodies:
ClassBody$_{opt}$
ClassBodies ClassBody
ClassBody:
ClassInhDec MethodBodies
MethodBodies:
MethodBody$_{opt}$
MethodBodies MethodBody
MethodBody:
Identifier(ResultType TypeList) Block
t;
TypeList:
Type
TypeList Type
Block:
{ *BlockStatements*}
BlockStatements:
BlockStatement
[VariableDeclarations] BlockStatements
BlockStatement:
Statement

VariableDeclarations:
VariableDeclaration$_{opt}$
VariableDeclarations VariableDeclaration

6.6.0.0.10 Expressions (a subset of Primary Expressions for Java)

Primary:

Literal
*t*this
t(*Expressiont*)
ClassInstanceCreationExpression
FieldAccess
MethodInvocation
Literal:
IntegerLiteral
FloatingPointLiteral
BooleanLiteral
CharacterLiteral
StringLiteral
NullLiteral
ClassInstanceCreationExpression:
tnew *ClassTypet*($ArgumentList_{opt}$ *t*)
ArgumentList:
AssignmentExpression
ArgumentListt,*AssignmentExpression*
AssignmentExpression:
ConditionalExpression[8]
Assignment
Assignment:
LeftHandSidet = *AssignmentExpression*
LeftHandSide:
ExpressionName
FieldAccess
ArrayAccess
FieldAccess:
Primary. Expression
tsuper.*Identifier*
MethodInvocation:
MethodNamet($ArgumentList_{opt}$*t*)
Primaryt.Identifiert($ArgumentList_{opt}$*t*)
tsuper .*Identifiert*($ArgumentList_{opt}$*t*)
Name:
SimpleName
QualifiedName
SimpleName:
Identifier

[8]see Appendix for a complete description

QualifiedName:
    Name. Identifier

6.6.0.0.11 Statements

Statement:
    EmptyStatement
    ClassInstanceCreationExpression
    Assignment
    MethodInvocation
    IfThenElseStatement
    ReturnStatement
EmptyStatement:
    ;
IfThenElseStatement:
    `if` (*Expression*) *Statement* `else` *Statement*
ReturnStatement:
    `return` $Expression_{opt}$

6.7 Java_3

Java_3 is the extension of Java_2 obtained by introducing some Java standard classes supporting the communication among threads and with the external environment.

Obviously, we picked up only some of such classes that implements all the kinds of communication supplied by the `JTN` notation; we have also considered other mechanisms, like, for example, sockets used by process of distributed applications, because we include them in the future extensions of `JTN`.

The semantics of the I-O mechanisms implemented in Java is of particular interest for what concern *observations* over the model of a program. The information exchange that occurs during I-O operations is made visible outside the program and will be used to determine the equivalence notion among programs: two programs are equivalent iff an external user can observe the same information exchange with the external environment in both the models.

MEAN	RESTRICTIONS	OPERATIONS
Files	**applications** using local files	read - write
other applet's method call	**applets**, not applications	read- write
Displaying Html pages	**applets**, not applications	read
Reading from an URL	none	read
connection to an URL	none	read and (sometimes) write
Sockets	none	read and write
Remote Method Invocation	applications	method call

Table 6.1: Communication in Java

6.7.1 Considerations on communication in Java

Java programs can communicate and interact with external entities in different ways. First, we can separate graphical interfaces and reading and writing operations performed on external information resources. In the first case, a program can interact with an external user that sends information by, for example, clicking on a button or typing text in a field. The program can process the occurrences of these events and reacts by changing the graphical output.

In the other case, a program can retrieve information from an external source or send them out to an external destination. The information can be located everywhere: in a disk, in the memory, or on a host on the network, and it can also be of any type: text, images, sounds, ...etc.

For the moment, we forget about graphical interfaces, and we focus on the other case. In table 6.1 we give a schematic classification of some Java mechanisms establishing connections with external resources; we indicate which are the access restrictions and which operations are allowed. In the first column we list the main Java mechanisms to communicate with the external, that is by using files, using URL simply to read a document or to establish a connection with a resource, opening sockets to connect to remote processes, on calling methods of remote objects, asking the browser to display an HTML page, calling methods of other applets. Some mechanisms (the last two, merely concerning applets) are presented just for completion, but they will not be considered later, other ones (remote method invocation), will be considered in future work.

The Java I/O fundamental concept is the *stream*. A stream is an ordered sequence of data that have a source (for input streams) or a destination (for output streams). To read or write data from or into an external resource, that could be a file, an URL, a socket or something else, a program opens a stream on that resource. Java defines several classes for streams operating on different data types. No matter where the information is coming from or going to, and no matter the type of data is being read or written, operations on streams are pretty much the same.

It is worthwhile to note that external resources can change dynamically (like files that can be read or written by other processes, CGI programs that can be modified or moved, etc) and the Java programs do not have the complete control over them. More specifically:

- a **file** can be read/written/canceled from outside the Java program by other processes; there is no way for the program to have exclusive access to a file.
- an **URL connection** may connect with a file, a HTML page or a CGI program; a priori, we do not know what kind of resource is linked to the connection;
- a **socket** is a TCP (but also UDP communication with socket is supported) connection with a given port on a given machine.

From these observation, we can deduce that the "real" content of a file (or a socket or a HTML page) cannot be modeled as a component of the program state, since it is known only outside the program. Besides, we need to have components of the program states that describe streams and sockets to resolve the expression evaluation and to identify the entities that exchange data with the program (for simplicity, we do not consider the file descriptors, used just to create streams to external files); we suppose that a socket connection with a file can be created only by suppling a string that is the file name.

Streams can also be connected with other streams, inside the same program, to create a pipelined communication mean among threads.

As we said above, giving a complete semantics of I-O in Java requires a long (thus repetitive in some parts) work and it is behind the scope of our work. We are interested in giving the semantics of a sub-language of Java that is powerful enough to implement a wide class of reactive-concurrent systems. It is not interesting, from our point of view, to consider all the classes implementing I-O operations in Java; we want to focus on the most relevant ones (from our point of view).

We have to adopt some pragmatic criteria in the choice of the classes, considering that the systems developed by our method probably need:

- **files** for
  - standard input-output operations; they are the simplest mechanisms to interact with an external user (we do not consider graphical interfaces for the moment);
  - data persistency (files are used to store the results of the computations of a part of a complex system);

- **pipelines** for implementing communication by streams among threads inside the same program;
- **sockets** as the main mechanism to communicate with processes residing on different hosts; they are the basic and simplest mechanisms for distributed programming in Java.

6.7.2 Streams and Files in Java

A file is an external "physical" resource, but Java gives the way to see it locally as an object of the program. Operations for receiving and sending data to external resources are implemented by methods. So, if we are interested in observing the flow of data between a Java program and an external resource, we can simply enrich the transition labels with the information about the data exchanged during the I/O method calls.

However, for the clarity, notice that objects suppling reading and writing methods represent streams rather than files. A *stream* is a communication channel between a source and a destination. Input streams allow to read from a source; output streams allow to write to a destination. In Java it is possible to read from or to write to a stream acting as on a common channel between the program and an external file, but it is also possible to use streams for pipelined connections among threads of the same program.

Java also differentiates between **character**, **byte** and **object** streams.

Obviously, we take into consideration just a very little subset of these classes, if we consider the wide hierarchy of classes representing different kinds of streams in Java.

6.7.2.1 Streams for Standard Input and Output

The standard input, generally associated with the keyboard, is accessed by means of the field `in` in the class `System` that is an instance of the class `InputStream`. Actually, `InputStream` is an abstract class. To be more precise, we would have to consider streams that are instances of some class implementing `InputStream` but we prefer to consider `InputStream` for the sake of simplicity.

The method that reads bytes from the source is the most important one. This method is overloaded in the class to allow the reading of a bunch of bytes rather then just one. For simplicity, we only consider the reading of a single byte at a time.

The instances of `InputStream` we consider are only the standard input (that we assume to be in the initial configuration of the program, and so it is not explicitly created as instance of `InputStream`) and, as we will see in the following, the input streams associated with sockets (that are implicitly created by a method of `Socket` class).

This is the simplified version of `InputStream` class used in Java$_3$:

```
public abstract class java.io.InputStream
  extends java.lang.Object
{
//Methods
public void close();
public int read(byte b);
}
```

The standard output, generally associated with the screen, is accessed by means of the field `out` in the class `System` that is an instance of the class `PrintStream`.

The method that writes bytes to the destination is the most important one. This methods is overloaded in the class to allow the writing of a bunch of bytes or of a complete line. For simplicity, we only consider the writing of a single byte at a time and, for the reasons described above.

These simplifications make classes `PrintStream` and `OutputStream` (the superclass of `PrintStream`) essentially the same. So, in Java$_3$ we use only `OutputStream` and we suppose that `System.out` has type `OutputStream`. This is the simplified version of `OutputStream` class used in Java$_3$:

```
public class java.io.OutputStream
{
//Methods
public void close();
public void write (byte b);
}
```

6.7.2.2 Streams to Read and to Write Files

`FileInputStream` and `FileOutputStream` represent input and output streams on a file that lives on the native file system. For simplicity, we assume that is possible to create a file stream only by using a file name (and not with `File` and `FileDescriptor` objects). Again, we only consider read and write methods that manage one byte at a time.

These are the simplified versions of the classes `FileInputStream` and `FileOutputStream` used in Java$_3$:

```
public class FileInputStream extends InputStream {
//Constructors
public FileInputStream(String path);

//Methods
public void close();
public int read(byte b);
}
```

```
public class FileOutputStream extends OutputStream{
// Constructors
public FileOutputStream(String path);

//Methods
public void write(byte b);
public void close();
}
```

It is clear that performing read or write operations on standard I-O files or on streams on external files is the same for a Java program.

6.7.2.3 Streams for pipelined communications

A piped input (output) stream is the receiving (sending) end of a communication pipe. Two threads can communicate by having one thread send data through a piped output stream and having the other thread read the data through a piped input stream.

The Java classes supplying these two kinds of objects are `PipedInputStream` and `PipedOutputStream`. These two classes has two constructors each: one that with no parameter, the other with a parameter that is the identity of the other end of the pipe (if it has been already created).

6.7.2.4 To read or write objects on streams

The classes `ObjectInputStream` and `ObjectOutputStream` can be used to read/write complete objects over streams, thus also over pipelined channels (creating an object stream over an instance of `PipedInputStream`or `PipedOutputStream`) as well as over files, (creating an object stream over an instance of `FileInputStream` and `FileOutputStream`). Notice that the two classes supply methods to read/write not only objects by also: bytes, chars, float, lines... that we forget, for the moment.

	INPUT	OUTPUT
STANDARD FILES EXTERNAL FILES	`InputStream` `FileInputStream`	`OutputStream` `FileOutputStream`
INTERNAL STREAMS	`PipedInputStream`	`PipedOutputStream`
OBJECT STREAMS	`ObjectInputStream`	`ObjectOutputStream`

Table 6.2: Stream Classes in $Java_3$

CLASS	Constructor	Methods	
`InputStream`		`close()`	`read(Byte b)`
`FileInputStream`	`FileInputStream(String s)`	`close()`	`read(Byte b)`
`PipedInputStream`	`PipedInputStream(String s)`	`close()`	`read(Byte b)`
`ObjectInputStream`	`ObjectInputStream(InputStream i)`	`close()`	`readObject(Object o)`
`OutputStream`		`close()`	`write(Byte b)`
`FileOutputStream`	`FileOutputStream`(String s)	`close()`	`write(Byte b)`
`PipedOutputStream`	`PipedOutputStream`(String s)	`close()`	`write(Byte b)`
`ObjectOutputStream`	`ObjectOutputStream(InputStream i)`	`close()`	`writeObject(Object o)`

Table 6.3: Classes for IO operations on files

6.7.3 Sockets in Java

At the core of Java's networking support are the classes `Socket` and `DatagramSocket` defining communication channels for processes over an IP network. A new socket is created by specifying an host, either by name or with an IP address, and a port number on that host.

The class `Socket` is used to create TCP connections over an IP network (while `DatagramSocket` manages UDP connections, and will be discarded, for the moment). A process on the remote host must be listening on the specified port number for incoming connection request. In Java, this can be done using the class `ServerSocket` whose main activity is to listen on a port for connection requests from clients and eventually return a new socket on a new port to allow such connection. In fig6.6 we report the two parts of a socket connection. On one side a client performs a connection request by requiring the creation of a new socket connected to a remote port on a specified host, for example using this code:

Client Code:

```
Socket myS;
InputStream in;
OutputStream out;
....
```

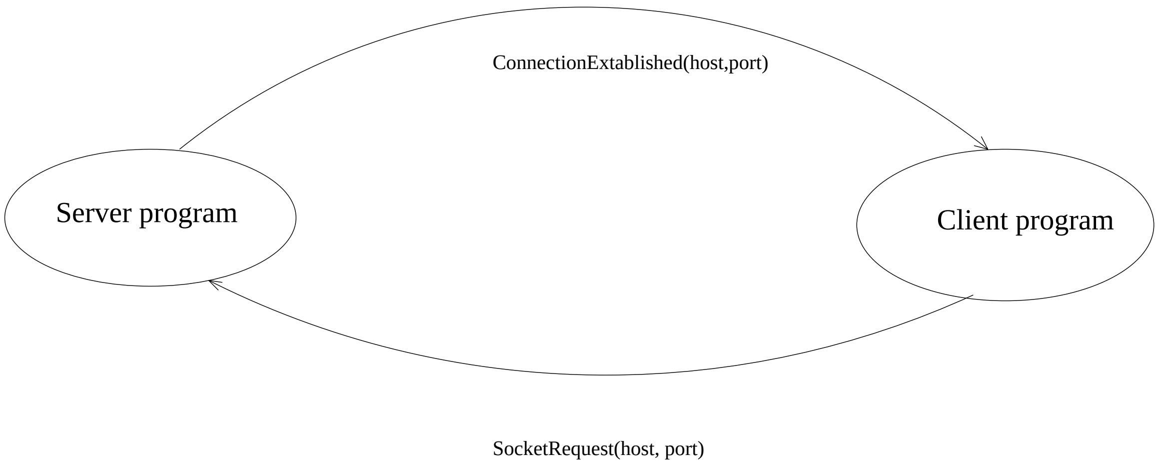


Figure 6.6: A socket connection request

```
mySS = new Socket("myServer", 3000);
in = myS.getInputStream();
out = myS.getOutputStream();
```

The `new` operation requires that the client program performs an interaction with the external environment (see fig.6.6), as a connection request must be sent to the server at the given port on the given host.

In the case that:

- the host does non exist;
- the port does not exist or no Java server is listening on it;
- new connections are no more accepted by the server;

then the corresponding exception is thrown; otherwise, a socket bound on a client local port is returned. Two streams can be used to read and write over the socket.

On the other side, the server program uses the class `ServerSocket` for creating a server side socket.
Server Code:

```
ServerSocket serverS;
```

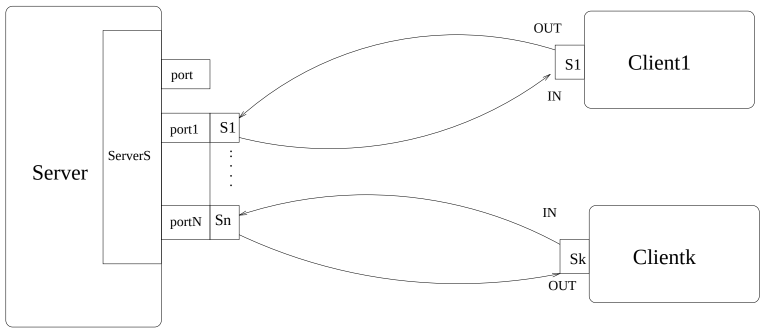


Figure 6.7: A client-server socket schema

```
Socket clientS;
...
serverS = new ServerSocket(3000);
clientS = serverS.accept();
```

The `accept` method waits until a client requests a socket connection on port 3000 on the server. When the connection is accepted and established, the method returns a `Socket` object that is bound to a new port (see fig.6.7). Here are the simplified versions of `ServerSocket` and `Socket` classes used in Java$_3$:

```
public class ServerSocket extends Object{
  ServerSocket(int port);
  Socket accept();
  void close();
  int getLocalPort(); // returns the port on which the server socket is listening;
}
```

```
public class Socket extends Object{
  Socket(String host, int port);
  void close();
  int getLocalPort();// returns the local port on which the socket is bound;
  int getPort;// returns the remote port to which the socket is connected;
  InputStream getInputStream();
  OutputStream getOutputStream();
}
```

6.8 Syntax and Static Semantics

The syntax of Java_3 is the syntax of Java_2 enriched by the class declarations and class bodies in
`InputStream`, `PrintStream`, `FileInputStream`, `FileOutputStream`, `ObjectInputStream`, `ObjectOutputStream`, `PipedInputStream`, `PipedOutputStream`, `Socket`, `ServerSocket`.

Thus Java_3 has the same statements as Java_2. The difference is that every program is implicitly enriched by the method code of the standard classes above.

The inference rules for term typing and annotation are the same given in Java_2, but in the following we can discard the annotations, as they are not meaningful.

6.9 Thread Semantics

The introduction of communication mechanism of Java does not modify thread states, but it requires new labels and rules.

In Java_3 a thread is modelled by $\mathsf{T\_LTS}_3 = (\mathsf{T\_states}_2, \mathsf{T\_labels}_3, \ {--->}_{\mathtt{p}}{}^3)$.

6.9.0.0.1 States The states are unchanged w.r.t. Java_2.

6.9.0.0.2 Labels We define the syntactic sets:

- $\mathsf{InputStream\_ids}$: identifiers of `InputStream` instances, ranged over by stnd_{in}.
- $\mathsf{OutputStream\_ids}$: identifiers of `OutputStream` instances, ranged over by stnd_{out}.
- $\mathsf{FileInputStream\_ids}$: identifiers of `FileInputStream` instances, ranged over by fs_{in};
- $\mathsf{FileOutputStream\_ids}$: identifiers of `FileOutputStream` instances, ranged over by fs_{out};
- $\mathsf{PipedInputStream\_ids}$: identifiers of `PipedInputStream` instances, ranged over by ps_{in};
- $\mathsf{PipedOutputStream\_ids}$: identifiers of `PipedOutputStream` instances, ranged over by ps_{out};
- $\mathsf{PipedStream\_ids} = \mathsf{PipedInputStream\_ids} \cup \mathsf{PipedOutputStream\_ids}$;
- $\mathsf{ObjectInputStream\_ids}$: identifiers of `ObjectInputStream` instances, ranged over by os_{in};

- ObjectOutputStream_ids: identifiers of `ObjectOutputStream` instances, ranged over by os_{out};
- ObjectStream_ids= ObjectInputStream_ids∪ ObjectOutputStream_ids;
- Input_ids= InputStream_ids∪ FileInputStream_ids∪ PipedInputStream_ids∪ ObjectInputStream_ids: identifiers of input streams, ranged over by st_{in};
- Output_ids= OutputStream_ids∪ FileOutputStream_ids∪ PipedOutputStream_ids∪ ObjectOutputStream_ids: identifiers of output streams, ranged over by st_{out};
- Stream_ids= Input_ids∪ Output_ids stream identifiers ranged over by str.
- Socket_ids: socket names, ranged over by sk_{id}.
- ServerSocket_ids: server socket names, ranged over by Ssk_{id}.

The identifiers in Java_3 (as in $\mathrm{Java}_1$and in Java_2) contain also the name of their class. Implicitly, this information is returned by function *Class*, that extends *Class* defined in Java_1.

Now we can precisely define the new set of labels that is:

$\mathsf{T\_labels}_3 = \mathsf{T\_labels}_2 \cup$
$(\{\mathit{NewFileIn}, \mathit{NewFileOut}\} \times \mathsf{Strings} \times (\mathsf{FileInputStream\_ids} \cup \mathsf{FileOutputStream\_ids}) \cup$
$(\{\mathit{NewObjectIn}, \mathit{NewObjectOut}\} \times \mathsf{ObjectInputStream\_ids} \cup \mathsf{OutputStream\_ids}) \cup$
$(\{\mathit{NewPipeIn}, \mathit{NewPipeOut}\} \times (\mathsf{PipedInputStream\_ids} \cup \mathsf{OutputStream\_ids} \cup \{\mathsf{none}\})) \cup$
$(\{\mathit{NewServerSocket}\} \times \mathsf{INTEGER} \times \mathsf{ServerSocket\_ids}) \cup$
$(\{\mathit{NewSocket}\} \times \mathsf{Strings} \times \mathsf{INTEGER} \times \mathsf{Socket\_ids}) \cup$
$(\{\mathit{Accept}\} \times \mathsf{ServerSocket\_ids} \times \mathsf{Socket\_ids}) \cup$
$(\{\mathit{ReadB}, \mathit{WriteB}\} \times \mathsf{Stream\_ids} \times \mathsf{Bytes}) \cup$
$(\{\mathit{ReadO}, \mathit{WriteO}\} \times \mathsf{Stream\_ids} \times \mathsf{NonCL\_ids}) \cup$
$(\{\mathit{GetInputStream}\} \times \mathsf{InputStream\_ids} \times \mathsf{Socket\_ids}) \cup$
$(\{\mathit{GetOutputStream}\} \times \mathsf{OutputStream\_ids} \times \mathsf{Socket\_ids})$
We briefly analyse every label:

- *NewFileIn*(*fname*,fs_{in}): a new stream fs_{in} of class FileInputStream_idsover file *fname* is created;
- *NewFileOut*(*fname*,fs_{out}): a new stream fs_{out} of class FileOutputStream_idsover file *fname* is created;
- *NewObjectIn*(st_{in},os_{in}): a new object stream os_{in} of class ObjectInputStream_idsover input stream st_{in} is created;

- $NewObjectOut(\mathsf{st}_{out},\mathsf{os}_{out})$: a new object stream os_{out} of class ObjectOutputStream_idsover output stream st_{out} is created;

- $NewPipeIn(\mathsf{ps}_{in},\mathsf{ps}_{out})$: a new stream ps_{in} of class PipedInputStream_idsconnected to ps_{out} (that can also be none) is created;

- $NewPipeOut(\mathsf{ps}_{in},\mathsf{ps}_{out})$: a new stream ps_{out} of class PipedOutputStream_idsconnected to ps_{in} (that can also be none) is created;

- $NewServerSocket(\mathsf{i},\mathsf{Ssk}_{id})$: a new server socket Ssk_{id} is created on port number i;

- $NewSocket$(host,p,sk_{id}): a new socket sk_{id} is created on port p of host host.

- $Accept(\mathsf{Ssk}_{id},\mathsf{sk}_{id})$: a server socket Ssk_{id} accepts a connection and returns a new socket sk_{id}.

- $ReadB(\mathsf{st}_{id},\mathtt{b}), WriteB(\mathsf{st}_{id},\mathtt{b})$: a byte b is read from (written to) socket st_{id}.

- $ReadO(\mathsf{st}_{id},nc_{id}), WriteO(\mathsf{st}_{id},nc_{id})$: an object nc_{id} is read from (written to) socket st_{id}.

- $GetInputStream(\mathsf{stnd}_{in},\mathsf{sk}_{id}), GetOutputStream(\mathsf{stnd}_{out},\mathsf{sk}_{id})$: the input (output) stream associated with socket sk_{id} is returned.

6.9.0.0.3 Transition relation $--->_{\mathtt{P}}{}^{3}$

is defined by the rules below (remember that we omit the non meaningful state components, every time it does not cause confusion; for example, in the following <`stm`> stands for $<t_{id}$,`stm`,$tSt,m,lmS>$)

New Input File *A thread calls the new method of class* `FileInputStream`*.*

$$< \texttt{new FileInputStream}(fname) > \xrightarrow{NewFileIn(fname,\mathsf{fs}_{in})}_{\mathtt{P}}{}^{3} < \mathsf{fs}_{in} >$$

New Output File *A thread calls the new method of class* `FileOutputStream`*.*

$$< \texttt{new FileInputStream}(fname) > \xrightarrow{NewFileOut(fname,\mathsf{fs}_{out})}_{\mathtt{P}}{}^{3} < \mathsf{fs}_{out} >$$

New Piped Input File *A thread calls the new method of class* `PipedInputStream`*.*

$$< \texttt{new PipedInputStream}(\mathsf{ps}_{out}) > \xrightarrow{NewPipeIn(\mathsf{ps}_{in},\mathsf{ps}_{out})}_{\mathtt{P}}{}^{3} < \mathsf{ps}_{in} >$$

New Piped Output File *A thread calls the new method of class* `PipedOutputStream`*.*

$$< \texttt{new PipedOutputStream}(\mathsf{ps}_{in}) > \xrightarrow{NewPipeOut(\mathsf{ps}_{out},\mathsf{ps}_{in})}{}_{\mathtt{p}}{}^{3} < \mathsf{ps}_{out} >$$

New Object Input File *A thread calls the new method of class* `ObjectInputStream` *to create an object input stream over the* st_{in} *stream.*

$$< \texttt{new ObjectInputStream}(\mathsf{st}_{in}) > \xrightarrow{NewObjectIn(\mathsf{st}_{in},\mathsf{os}_{in})}{}_{\mathtt{p}}{}^{3} < \mathsf{os}_{in} >$$

New Object Output File *A thread calls the new method of class* `ObjectOutputStream` *to create an object output stream over the* st_{out} *stream.*

$$< \texttt{new ObjectOutputStream}(\mathsf{st}_{out}) > \xrightarrow{NewObjectOut(\mathsf{st}_{out},\mathsf{os}_{out})}{}_{\mathtt{p}}{}^{3} < \mathsf{os}_{out} >$$

Read Byte *A thread reads a byte from a* `FileInputStream` *or* `PipedInputStream` *instance.*

$$< \mathsf{st}_{in}.\texttt{read}() > \xrightarrow{ReadB(\mathsf{st}_{in},\mathtt{b})}{}_{\mathtt{p}}{}^{3} < \mathtt{b} >$$

Write Byte *A thread writes a byte to a* `FileOutputStream` *or* `PipedOutputStream` *instance.*

$$< \mathsf{st}_{out}.\texttt{writeS}(\mathtt{b}) > \xrightarrow{WriteB(\mathsf{st}_{out},\mathtt{b})}{}_{\mathtt{p}}{}^{3} < \texttt{skip} >$$

Read Object *A thread reads an object from an* `ObjectInputStream` *instance.*

$$< \mathsf{os}_{in}.\texttt{readObject}() > \xrightarrow{ReadO(\mathsf{os}_{in},nc_{id})}{}_{\mathtt{p}}{}^{3} < nc_{id} >$$

Write Object *A thread writes an object to an* `ObjectOutputStream` *instance.*

$$< \mathsf{os}_{out}.\texttt{writeObject}(nc_{id}) > \xrightarrow{WriteO(\mathsf{os}_{out},nc_{id})}{}_{\mathtt{p}}{}^{3} < \texttt{skip} >$$

NewServerS *A thread calls the new method to create a server socket listening on a given port.*

$$< \texttt{new ServerSocket}(\mathsf{p}) > \xrightarrow{NewServerSocket(\mathsf{p},\mathsf{Ssk}_{id})}{}_{\mathtt{p}}{}^{3} < \mathsf{Ssk}_{id} >$$

Accept *A server socket* Ssk_{id} *accepts a connection request and returns a new socket.*

$$< \mathsf{Ssk}_{id}.\texttt{accept}() > \xrightarrow{Accept(\mathsf{Ssk}_{id},\mathsf{sk}_{id})}{}_{\mathtt{p}}{}^{3} < \mathsf{sk}_{id} >$$

NewSocket *A thread tries to connect to the server. It is returned a socket connected to a socket on the server's host.*

$$< \texttt{new Socket}(host,\mathsf{p}) > \xrightarrow{NewSocket(host,\mathsf{p},\mathsf{sk}_{id})}{}_{\mathtt{p}}{}^{3} < \mathsf{sk}_{id} >$$

GetInputStream *A thread gets an input stream connected to a socket.*

$$< \mathsf{sk}_{id}.\texttt{getInputStream}() > \xrightarrow{GetInputStream(\mathsf{st}_{in},\mathsf{sk}_{id})}{}_{\mathtt{p}}{}^{3} < \mathsf{stnd}_{in} >$$

GetOutputStream *A thread gets an output stream connected to a socket.*

$$< \mathsf{sk}_{id}.\mathtt{getOutputStream}() > \xrightarrow{GetOutputStream(\mathsf{st}_{out},\mathsf{sk}_{id})}_{\mathtt{p}}{}^{3} < \mathsf{stnd}_{out} >$$

6.10 Program Semantics

The description of an intermediate state in the computation of a Java$_3$ program is enriched by elements describing objects of class streams and sockets.

The labels describe the information exchange with the external caused by I-O operations.

6.10.0.1 I/O Stream State

The streams connected with the external environment are passive components as they do not perform independent activities on their own. We model them simply by describing their states in each moment of the computation.

A I/O stream state must contain:

- an identifier for the stream;
- a reference to the resource it is attached to (a file, a socket or a file stream in case of an object stream);
- a status (open-closed) bit.

$\mathsf{IOStream} = \mathsf{Stream\_ids} \times \mathsf{Ext\_Resource} \times \mathsf{Bool}$

where $\mathsf{Ext\_Resource} = \mathsf{Strings} \cup \mathsf{Socket\_ids} \cup \mathsf{ObjectStream\_ids}$ is the set of file names or socket identifiers; it is ranged over by metavariable *res*.

$<\mathsf{str},\mathrm{res},\mathsf{b}>$ represents a state for the stream str associated with the external resource identified by name res, that can be opened or not, depending on the value of b.

For simplicity, input and output standard streams are modeled like streams to files, where the name of the resource is *IN* for standard input and *OUT* for standard output.

We define the extractors:

- *IdInfo*:$\mathsf{IOStream} \rightarrow \mathsf{Stream\_ids}$, defined by *IdInfo*$(<\mathsf{str},\mathrm{res},\mathsf{b}>) = \mathsf{str}$, that returns the identity of the stream.
- *Resource*: $\mathsf{IOStream} \rightarrow \mathsf{Ext\_Resource}$, defined by *Resource*$(<\mathsf{str},\mathrm{res},\mathsf{b}>) = \mathrm{res}$, that returns the reference to the resource associated with the stream:

- *IsOpen*:IOStream→ Stream_ids, defined by *IsOpen*(<str,res,b>)= b, that returns the status bit.

6.10.0.2 Piped Stream State

A pipelined communication does not exchange data with the external environment. A thread uses a piped output stream to write data that will be read by a thread using the other ending of the pipe.

The two streams are tightly connected and shares a memory in which read or write data. Thus, the two streams are described by the same component, whose state must contain:

- the identifier of the input piped stream;
- the identifier of the output piped stream;
- the status (open-closed) bit.
- the list of the bytes written on the stream.

PipedStream= Stream_ids× Stream_ids× Bool× $STREAM\_VALUES$

and

$STREAM\_VALUES$= List(Bytes) are the lists of bytes; ranged over by the str_{val} metavariable.

6.10.0.3 Socket State

A `ServerSocket` object waits for connection requests on a port of the server host. A `Socket` object is bound to a local port on a network host and is connected to another socket bound on a port on the server host. A socket can be identified by a port number because just one socket can be connected to a single port. The state of a socket must contain two port numbers, one for the local port and one for the remote port to which the socket is connected.

In the following, p is a metavariable ranging over INTEGER used to denote a port number.

6.10.0.4 Server Socket states

The state of a `ServerSocket` instance contains:

- a status (open-closed) bit;
- a local port number;

$\texttt{ServerSocket} = \mathsf{ServerSocket\_ids} \times \mathsf{BOOL} \times \mathsf{INTEGER}$.

$< \mathsf{Ssk}_{id}, \mathsf{b}, \mathsf{p} >$ represents a state for server socket Ssk_{id} listening on server port p; b is true if the server socket is open.

We use the extractors *IdInfo*, *IsOpen* and *Port*: the definition of the first two ones is analogous to the ones used above that operates on streams; *Port*:$\texttt{ServerSocket} \rightarrow \mathsf{INTEGER}$ is defined by

$Port(<\mathsf{Ssk}_{id},\mathsf{b},\mathsf{p}>) = \mathsf{p}$ returns the port to which the server socket is bound.

6.10.0.4.1 Socket states The state of a `Socket` instance contains:

- input and output stream references;
- a status (open-closed) bit;
- a local port number;
- a port number to identify the socket on the other end of the connection;

$\texttt{Socket} = \mathsf{Socket\_ids} \times \mathsf{Stream\_ids} \times \mathsf{Stream\_ids} \times \mathsf{BOOL} \times \mathsf{INTEGER} \times \mathsf{INTEGER}$.

$<\mathsf{sk}_{id},\mathsf{str\_IN},\mathsf{str\_OUT},\mathsf{b},\mathsf{p}_L,\mathsf{p}_R>$ represents a state for socket sk_{id} whose input stream is $\mathsf{str\_IN}$ and output stream is $\mathsf{str\_OUT}$; p_L and p_R identify, respectively, the local and remote ports; $\mathsf{b}$is true if the socket is open.
We define the following extractors:

- *IdInfo*, *IsOpen*, *LocalPort*,*RemotePort*: $\texttt{ServerSocket} \rightarrow \mathsf{INTEGER}$ defined by
  $LocalPort(<\mathsf{sk}_{id},\mathsf{str\_IN},\mathsf{str\_OUT},\mathsf{b},\mathsf{p}_L,\mathsf{p}_R>) = \mathsf{p}_L$
  $RemotePort(<\mathsf{sk}_{id},\mathsf{str\_IN},\mathsf{str\_OUT},\mathsf{b},\mathsf{p}_L,\mathsf{p}_R>) = \mathsf{p}_R$
- *InputStr*,*OutputStr*: $\texttt{ServerSocket} \rightarrow \mathsf{Stream_ids}$defined by
  $InputStr(<\mathsf{sk}_{id},\mathsf{str\_IN},\mathsf{str\_OUT},\mathsf{b},\mathsf{p}_L,\mathsf{p}_R>) = \mathsf{str\_IN}$
  $OutputStr(<\mathsf{sk}_{id},\mathsf{str\_IN},\mathsf{str\_OUT},\mathsf{b},\mathsf{p}_L,\mathsf{p}_R>) = \mathsf{str\_OUT}$
  (the definitions of *IdInfo* and *IsOpen* are obvious).

6.10.0.4.2 Communication standard classes

It is worthwhile to note that classes `InputStream`,`OutputStream`,`FileInputStream`,`FileOutputStream`, `PipedInputStream`,`PipedOutputStream`,`ObjectInputStream`,`ObjectOutputStream`, `Socket`, `ServerSocket`
can be modeled by a simpler state w.r.t. Java_2, since these classes have neither class nor instance fields and, consequently, the memory and initializer components can be discarded.

Thus, the I/O classes are modeled by couples in $\texttt{IOClass} = \mathsf{CL\_ids} \times Set(\mathsf{OB\_ids})$.

A class state $< c_{id}, S_{id} >$ consists of:

- c_{id} : the class identifier;
- S_{id}: a set of identifiers of previously created instances.

6.10.0.5 Program Semantics

In Java_3 a program is modelled by $\mathsf{P\_LTS}_3 = (\mathsf{P\_states}_3, \mathsf{P\_labels}_3, \longrightarrow_{\mathtt{P}}{}^3)$.

6.10.0.5.1 Program states

$\mathsf{P\_states}_3$= $\text{Set}(\mathsf{P\_states}_2 \cup \mathsf{IOStream} \cup \mathsf{PipedStream} \cup \texttt{ServerSocket} \cup \texttt{Socket} \cup \texttt{IOClass})$

and in `IOClass` there are the components for classes `InputStream`,`OutputStream`...

6.10.0.5.2 Program labels

$\mathsf{P\_labels}_3 = \{\tau\} \cup$
$(\{NEW\_IN\_F, NEW\_OUT\_F\} \times \mathsf{Strings}) \cup$
$(\{READ, WRITE\} \times \mathsf{Strings} \times \mathsf{Bytes}) \cup$
$(\{READ\_O, WRITE\_O\} \times \mathsf{Stream\_ids} \times \mathsf{Strings} \times (\mathsf{O\_states}_1 \cup \mathsf{T\_states}_1) \cup$
$(\{ACC\_CONN, REQ\_CONN\} \times \mathsf{INTEGER} \times \mathsf{Strings} \times \mathsf{Socket\_ids})$

We briefly analyse every label:

- $NEW\_IN\_F(fname)$,$NEW\_OUT\_F(fname)$: a new input (output) stream associated with the file *fname* is created;
- $READ(\mathsf{st}_{id},\text{res},\mathsf{b})$, $WRITE(\mathsf{st}_{id},\text{res},\mathsf{b})$: a byte b is read from (written into) the external resource res;
- $READ\_O(str,nc)$, $WRITE\_O(str,nc)$: an object state *nc* is read from (written into) the stream named *str*

- $ACC\_CONN(\mathsf{p},\mathsf{host},\mathsf{p}_{rem},\mathsf{p}_{loc},\mathsf{sk}_{id})$: a socket connection with the port p on host host from port p_{rem} is accepted, external to the program; a new socket sk_{id} on port $\mathsf{p}_{loc}$is returned.
- $REQ\_CONN(\mathsf{p},\text{host},\mathsf{p}_{rem},\mathsf{p}_{loc},\mathsf{sk}_{id})$:a socket connection is requested from port p_{rem} to the port p on host host; a new socket sk_{id} on the port p_{loc} is returned.

6.10.0.5.3 Transition Relation

$\longrightarrow_{\mathtt{p}}{}^3 : \mathsf{P\_states}_3 \times \mathsf{P\_labels}_3 \rightarrow \mathsf{P\_states}_3$ is defined by the inductive rules below.

- Stream creation

PNewInFile *A thread creates a new file input stream, associated with a file object. There is an interaction with the external environment, since the system has to create the file if it does not exist.*

A new component, modeling the new stream, is added to the program state.

$$\frac{tc \xrightarrow{NewFileIn(fname,\mathsf{st}_{in})}{}_{\mathtt{p}}{}^3\; tc'}{\begin{array}{c} tc|<\mathtt{FileInputStream}, S_{id}>|\mathsf{S} \xrightarrow{NEW\_IN\_F(fname)}{}_{\mathtt{p}}{}^3 \\ tc'|<\mathtt{FileInputStream}, S_{id}\cup\{\mathsf{stnd}_{in}\}>|<\mathsf{st}_{in}, fname, \mathsf{true}>|\mathsf{S}\end{array}}$$

if $\mathsf{st}_{in}\notin S_{id}$, $Class(\mathsf{st}_{in}) =$`FileInputStream`
Analogous for the creation of an output stream.

PNewInObject *A thread creates a new input object stream over an input stream. A new component, modeling the new stream, is added to the program state. The transition has no label visible to the external.*

$$\frac{tc \xrightarrow{NewObjectIn(\mathsf{st}_{in},\mathsf{os}_{in})}{}_{\mathtt{p}}{}^3\; tc'}{\begin{array}{c} tc|<\mathtt{ObjectInputStream}, S_{id}>|\mathsf{S} \xrightarrow{\tau}{}_{\mathtt{p}}{}^3 \\ tc'|<\mathtt{ObjectInputStream}, S_{id}\cup\{\mathsf{st}_{in}\}>|<\mathsf{os}_{in}, \mathsf{st}_{in}, \mathsf{true}>|\mathsf{S}\end{array}}$$

if $\mathsf{os}_{in}\notin S_{id}$, $Class(\mathsf{os}_{in}) =$`ObjectInputStream`
Analogous for the creation of an output stream.

PNewInPipe 1 *A thread creates a new input pipe stream eventually connected to an exist-*

ing output stream. If the other end of the pipe has not yet been created, a new component, modeling the new stream, is added to the program state.

$$\frac{tc \xrightarrow{\mathit{NewPipeIn}(\mathsf{ps}_{in},\mathsf{ps}_{out})}{}_{\mathrm{p}}{}^{3}\, tc'}{tc| < \mathtt{PipedInputStream}, S_{id} > | < \mathtt{undef}, \mathsf{ps}_{out}, \mathsf{false}, \mathit{emptyl} > |\mathsf{S} \xrightarrow{\tau}{}_{\mathrm{p}}{}^{3} \; tc'| < \mathtt{PipedInputStream}, S_{id} \cup \{\mathsf{ps}_{in}\} > || < \mathsf{ps}_{in}, \mathsf{ps}_{out}, \mathsf{true}, \mathit{emptyl} > |\mathsf{S}}$$

if $\mathsf{ps}_{out} \neq$ none, $\mathsf{ps}_{in} \notin S_{id}$, $\mathit{Class}(\mathsf{ps}_{in}) =$`PipedInputStream`

PNewInPipe 2 *Otherwise the state of the pipe connection is updated with the identity of the input stream*

$$\frac{tc \xrightarrow{\mathit{NewPipeIn}(\mathsf{ps}_{in},\mathsf{none})}{}_{\mathrm{p}}{}^{3}\, tc'}{tc| < \mathtt{PipedInputStream}, S_{id} > |\mathsf{S} \xrightarrow{\tau}{}_{\mathrm{p}}{}^{3} \; tc'| < \mathtt{PipedInputStream}, S_{id} \cup \{\mathsf{ps}_{in}\} > | < \mathsf{ps}_{in}, \mathtt{undef}, \mathsf{false}, \mathit{emptyl} > |\mathsf{S}}$$

if $\mathsf{ps}_{in} \notin S_{id}$, $\mathit{Class}(\mathsf{ps}_{in}) =$`PipedInputStream`

Analogous for the creation of on output stream.

- READ OVER STREAM

PReadByte1 *The program reads a byte from the external file fname by means of input stream* st_{id}. *The stream must be open.*

$$\frac{tc \xrightarrow{\mathit{ReadB}(\mathsf{fs}_{in},\mathtt{b})}{}_{\mathrm{p}}{}^{3}\, tc'}{tc| < \mathsf{fs}_{in}, \mathit{fname}, \mathsf{true} > |\mathsf{S} \xrightarrow{\mathit{READ}(\mathit{fname},\mathtt{b})}{}_{\mathrm{p}}{}^{3}\, tc'| < \mathsf{fs}_{in}, \mathit{fname}, \mathsf{true} > |\mathsf{S}}$$

PReadByte2 *The program reads a byte from the input stream* ps_{in} *pipelined with another thread. The stream must be open, the read value is consumed.*

$$\frac{tc \xrightarrow{\mathit{ReadB}(\mathsf{ps}_{in},\mathtt{b})}{}_{\mathrm{p}}{}^{3}\, tc'}{tc| < \mathsf{ps}_{in}, \mathsf{ps}_{out}, \mathsf{true}, \mathsf{str}_{val} > |\mathsf{S} \xrightarrow{\tau}{}_{\mathrm{p}}{}^{3}\, tc'| < \mathsf{ps}_{in}, \mathsf{ps}_{out}, \mathsf{true}, \mathit{Pop}(\mathsf{str}_{val}) > |\mathsf{S}}$$

if $\mathtt{b} = \mathit{Top}(\mathsf{ps}_{in})$

PReadObject1 *The program reads an object from the object input stream* os_{in} *over the stream* ps_{in} *pipelined with another thread. The stream* os_{in} *must be open and the object* *value is consumed from* ps_{in} *(notice that the stream contains object identifiers.*

$$\frac{tc \xrightarrow{ReadO(\mathsf{os}_{in}, nc_{id})}_{\mathrm{P}}{}^{3}\ tc'}{tc| < \mathsf{os}_{in}, \mathsf{ps}_{in}, \mathsf{true} > | < \mathsf{ps}_{in}, \mathsf{ps}_{out}, \mathsf{true}, \mathsf{str}_{val} > |\mathsf{S} \xrightarrow{\tau}_{\mathrm{P}}{}^{3} \\ tc'| < \mathsf{os}_{in}, \mathsf{ps}_{in}, \mathsf{true} > | < \mathsf{ps}_{in}, \mathsf{ps}_{out}, \mathsf{true}, Pop(\mathsf{str}_{val}) > |\mathsf{S}}$$

if $nc_{id} = Top(\mathsf{str}_{val})$

PReadObject2 *The program reads an object from the external file fname by means of an object input stream* os_{in} *over* st_{id}*. The stream must be open; the read object is assigned an new identity and becomes a component of the system.*

$$\frac{tc \xrightarrow{ReadO(\mathsf{os}_{in}, nc_{id})}_{\mathrm{P}}{}^{3}\ tc'}{tc| < \mathtt{Object}, S_{id}, m, lm > | < \mathsf{st}_{id}, fname, \mathsf{true} > |\mathsf{S} \xrightarrow{READ\_O(fname, m)}_{\mathrm{P}}{}^{3} \\ tc'| < \mathsf{st}_{id}, fname, \mathsf{true} > | < nc_{id}, m > |\mathsf{S}}$$

if $nc_{id} \notin S_{id}$

PReadSocket1 *The program reads a byte from the external socket str by means of input stream* st_{id}*. The stream and the socket must be open.*

$$\frac{tc \xrightarrow{ReadB(\mathsf{st}_{id}, \mathsf{b})}_{\mathrm{P}}{}^{3}\ tc'}{tc| < \mathsf{st}_{id}, SOCKET, \mathsf{sk}_{id}, \mathsf{true} > |\mathsf{skcomp}|\mathsf{S} \xrightarrow{READ(\mathsf{st}_{id}, \mathsf{sk}_{id}, \mathsf{b})}_{\mathrm{P}}{}^{3} \\ tc'| < \mathsf{st}_{id}, \mathsf{sk}_{id}, \mathsf{true} > |\mathsf{skcomp}|\mathsf{S}}$$

if $IdInfo(\mathsf{skcomp}) = \mathsf{sk}_{id}, Open(\mathsf{skcomp})$

- Write over stream

PWriteByte1 *The program writes a byte to the external file fname by means of an input stream* st_{id}*. The stream must be open.*

$$\frac{tc \xrightarrow{WriteB(\mathsf{st}_{id}, \mathsf{b})}_{\mathrm{P}}{}^{3}\ tc'}{tc| < \mathsf{st}_{id}, fname, \mathsf{true} > |\mathsf{S} \xrightarrow{WRITE(str, lm)}_{\mathrm{P}}{}^{3}\ tc'| < \mathsf{st}_{id}, fname, \mathsf{true} > |\mathsf{S}}$$

PWriteByte2 *The program writes a byte to the output stream* ps_{out} *pipelined with another thread. The stream must be open, the written value is added to the state of* ps_{out}*.*

$$\frac{tc \xrightarrow{ReadB(\mathsf{ps}_{in},\mathsf{b})}{}_{\mathrm{p}}{}^{3}\, tc'}{tc| < \mathsf{ps}_{in}, \mathsf{ps}_{out}, \mathsf{true}, \mathsf{v} > |\mathsf{S} \xrightarrow{\tau}{}_{\mathrm{p}}{}^{3}\, tc'| < \mathsf{ps}_{in}, \mathsf{ps}_{out}, \mathsf{true}, Push(\mathsf{b}, \mathsf{str}_{val}) > |\mathsf{S}}$$

PWriteObject1 *The program writes an object on the object input stream* os_{out} *over the stream* ps_{in} *pipelined with another thread. The stream* os_{out} *must be open and the object identity is added to* ps_{in} *(notice that the stream contains object identifiers.*

$$\frac{tc \xrightarrow{WriteO(\mathsf{os}_{out},nc_{id})}{}_{\mathrm{p}}{}^{3}\, tc'}{\begin{array}{l} tc| < \mathsf{os}_{out}, \mathsf{ps}_{out}, \mathsf{true} > | < \mathsf{ps}_{in}, \mathsf{ps}_{out}, \mathsf{true}, \mathsf{str}_{val} > | < nc_{id}, m > |\mathsf{S} \xrightarrow{\tau}{}_{\mathrm{p}}{}^{3} \\ tc'| < \mathsf{os}_{out}, \mathsf{ps}_{out}, \mathsf{true} > | < \mathsf{ps}_{in}, \mathsf{ps}_{out}, \mathsf{true}, Push(m, \mathsf{str}_{val}) > | < \\ nc_{id}, m > |\mathsf{S} \end{array}}$$

PWriteObject2 *The program writes an object to the external file fname by means of an object output stream* os_{out} *over* st_{id}*. The stream must be open; the status of the written object is sent outside.*

$$\frac{tc \xrightarrow{WriteO(\mathsf{os}_{out},nc_{id})}{}_{\mathrm{p}}{}^{3}\, tc'}{\begin{array}{l} tc| < \mathsf{os}_{out}, \mathsf{st}_{id}, \mathsf{true} > | < \mathsf{st}_{id}, \mathit{fname}, \mathsf{true} > | < nc_{id}, m > |\mathsf{S} \xrightarrow{READ\_O(\mathit{fname},m)}{}_{\mathrm{p}}{}^{3} \\ tc'| < \mathsf{os}_{out}, \mathsf{st}_{id}, \mathsf{true} > | < \mathsf{st}_{id}, \mathit{fname}, \mathsf{true} > | < nc_{id}, m > |\mathsf{S} \end{array}}$$

PWriteSocket *The program writes a byte to an open socket.*

$$\frac{tc \xrightarrow{WriteB(\mathsf{st}_{id},\mathsf{b})}{}_{\mathrm{p}}{}^{3}\, tc'}{tc| < \mathsf{st}_{id}, \mathsf{sk}_{id}, \mathsf{true} > |\mathsf{skcomp}|\mathsf{S} \xrightarrow{WRITE(\mathsf{st}_{id},\mathsf{sk}_{id},\mathsf{b})}{}_{\mathrm{p}}{}^{3}\, tc'| < \mathsf{st}_{id}, \mathsf{sk}_{id}, \mathsf{true} > |\mathsf{skcomp}|\mathsf{S}}$$

if $\mathit{IdInfo}(\mathsf{skcomp}) = \mathsf{sk}_{id}, \mathit{Open}(\mathsf{skcomp})$

PNewServerSocket *The program creates a new server socket listening on a given port.*

$$\frac{tc \xrightarrow{NewServerSocket(\mathsf{p},\mathsf{Ssk}_{id})}{}_{\mathrm{p}}{}^{3}\, tc'}{\begin{array}{l} tc| < \mathtt{ServerSocket}, S_{id} > |\mathsf{S} \xrightarrow{\tau}{}_{\mathrm{p}}{}^{3} \\ tc'| < \mathtt{ServerSocket}, S_{id} \cup \{\mathsf{Ssk}_{id}\} > | < \mathsf{Ssk}_{id}, \mathsf{true}, \mathsf{p} > |\mathsf{S} \end{array}}$$

if $\mathit{Class}(\mathsf{Ssk}_{id}) = \mathtt{ServerSocket}$, $\mathsf{Ssk}_{id} \notin S_{id}$

PNewSocket *It is created a new socket object that is connected to a socket at the other end of the link. Note that the input and output streams for the socket are created by the* getInputStream *and* getOutputStream *calls.*

$$\frac{tc \xrightarrow{NewSocket(host, \mathsf{p}, \mathsf{sk}_{id})}{}_{\mathrm{p}}{}^{3}\; tc'}{tc| < \texttt{Socket}, S_{id} > |\mathsf{S} \xrightarrow{REQ\_CONN(\mathsf{p}, host, \mathsf{p}_{rem}, \mathsf{p}_{loc}, \mathsf{sk}_{id})}{}_{\mathrm{p}}{}^{3}\; tc'| < \texttt{Socket}, S_{id} \cup \{\mathsf{sk}_{id}\} > |\mathsf{skcomp}|\mathsf{S}}$$

if $\mathsf{sk}_{id} \notin S_{id}$, $Class(\mathsf{sk}_{id})=$ Socket, $\mathsf{skcomp}=<\mathsf{sk}_{id},\mathsf{undef},\mathsf{undef},\mathsf{true},\mathsf{p}_{loc},\mathsf{p}_{rem}>$

PAccept *A server socket accepts a socket connection request coming from outside. The returned socket is connected to a different port.*

$$\frac{tc \xrightarrow{Accept(\mathsf{Ssk}_{id}, \mathsf{sk}_{id})}{}_{\mathrm{p}}{}^{3}\; tc'}{\begin{array}{c} tc| < \texttt{Socket}, S_{id} > | < \mathsf{Ssk}_{id}, \mathsf{open}, \mathsf{p} > |\mathsf{S} \xrightarrow{ACC\_CONN(\mathsf{p}, host, \mathsf{p}_{rem}, \mathsf{p}_{loc}, \mathsf{sk}_{id})}{}_{\mathrm{p}}{}^{3} \\ tc'| < \texttt{Socket}, S_{id} \cup \{\mathsf{sk}_{id}\} > | < \mathsf{Ssk}_{id}, \mathsf{open}, \mathsf{p} > |\mathsf{skcomp}|\mathsf{S} \end{array}}$$

if $\mathsf{sk}_{id} \notin S_{id}$, $Class(\mathsf{sk}_{id})=$Socket, $Port(\mathsf{Sskcomp})=\mathsf{p}$,

$\mathsf{skcomp}=<\mathsf{sk}_{id},\mathsf{undef},\mathsf{undef},\mathsf{true},\mathsf{p}_{loc},\mathsf{p}_{rem}>$

PGetInput *It is created an new input stream. The corresponding reference is put in the socket state.*

$$\frac{tc \xrightarrow{GetInputStream(\mathsf{sk}_{id}, \mathsf{st}_{id})}{}_{\mathrm{p}}{}^{3}\; tc'}{\begin{array}{c} tc| < \mathsf{sk}_{id}, \mathsf{undef}, \mathsf{true}, \mathsf{p}_1, \mathsf{p}_2 > | < \texttt{InputStream}, S_{id} > |\mathsf{S} \xrightarrow{\tau}{}_{\mathrm{p}}{}^{3} \\ tc'| < \mathsf{sk}_{id}, \mathsf{str}, \mathsf{undef}, \mathsf{true}, \mathsf{p}_1, \mathsf{p}_2 > | < \texttt{InputStream}, S_{id} \cup \{\mathsf{st}_{id}\} > | < \mathsf{st}_{id}, \mathsf{sk}_{id}, \mathsf{true} > |\mathsf{S} \end{array}}$$

if $\mathsf{st}_{id} \notin S_{id}$, $Class(\mathsf{st}_{id})=$InputStream,

$InputStr(\mathsf{skcomp'})=\mathsf{st}_{id}$

PGetOutput *It is created an new output stream. The corresponding reference is put in the socket state.*

$$\frac{tc \xrightarrow{GetOutputStream(\mathsf{sk}_{id}, \mathsf{st}_{id})}{}_{\mathrm{p}}{}^{3}\; tc'}{\begin{array}{c} tc| < \mathsf{sk}_{id}, \mathsf{undef}, \mathsf{undef}, \mathsf{true}, \mathsf{p}_1, \mathsf{p}_2 > | < \texttt{OutputStream}, S_{id} > |\mathsf{S} \xrightarrow{\tau}{}_{\mathrm{p}}{}^{3} \\ tc'| < \mathsf{sk}_{id}, \mathsf{str}, \mathsf{undef}, \mathsf{true}, \mathsf{p}_1, \mathsf{p}_2 > | < \texttt{OutputStream}, S_{id} \cup \{\mathsf{st}_{id}\} > | < \mathsf{st}_{id}, \mathsf{sk}_{id}, \mathsf{true} > |\mathsf{S} \end{array}}$$

if $\mathsf{st}_{id} \notin S_{id}$, $Class(\mathsf{st}_{id})=$OutputStream,

$OutputStr(\mathsf{skcomp'})=\mathsf{st}_{id}$

Chapter 7

Correctness of the translation

7.1 Rationale

In the previous chapters we give the precise semantics of a `JTN` specification and its translation into Java. But, how can we be reasonably sure that the obtained Java implementation is correct w.r.t. the `JTN` specification? We would be able to compare the model of the `JTN` specification with the model of the Java program and decide if there are some substantial, observable differences among them. We are facilitated by the choice of giving both the `JTN` semantics and the Java semantics by means of generalized label transition systems (shortly *gltss*, see App. D).

Given a `JTN` specification *jtn* , we want to define the semantical equivalence among the *glts* *SEM*, defined by the `JTN` semantics (that is the initial model of the CASL specification *casl* associated with *jtn* by the semantics defined in Chap.4), and the *glts* *JAVA* associated with the Java program *java* by the Java semantics in Chap.6: *java* is obtained by means of the translation function in Chap.5.1 applied to *jtn* . Figure 7.1 shows this situation.

Thus, we have to compare the two *gltss*:

$SEM{=}(State_S\ ,\ Label_S\ ,\ \rightarrow_S\ ,\ Info_S)$ that models *casl* and

$JAVA{=}(State_J\ ,\ Label_J\ ,\ \rightarrow_J\ ,\ Info_J)$ that models *java* .

To be more precise:

- the CASL specification defines generalized transitions of the form $i : se \xrightarrow{l} se'$, where the associated information i contains the complete description of the transitions performed by the active components participating in the system transition;

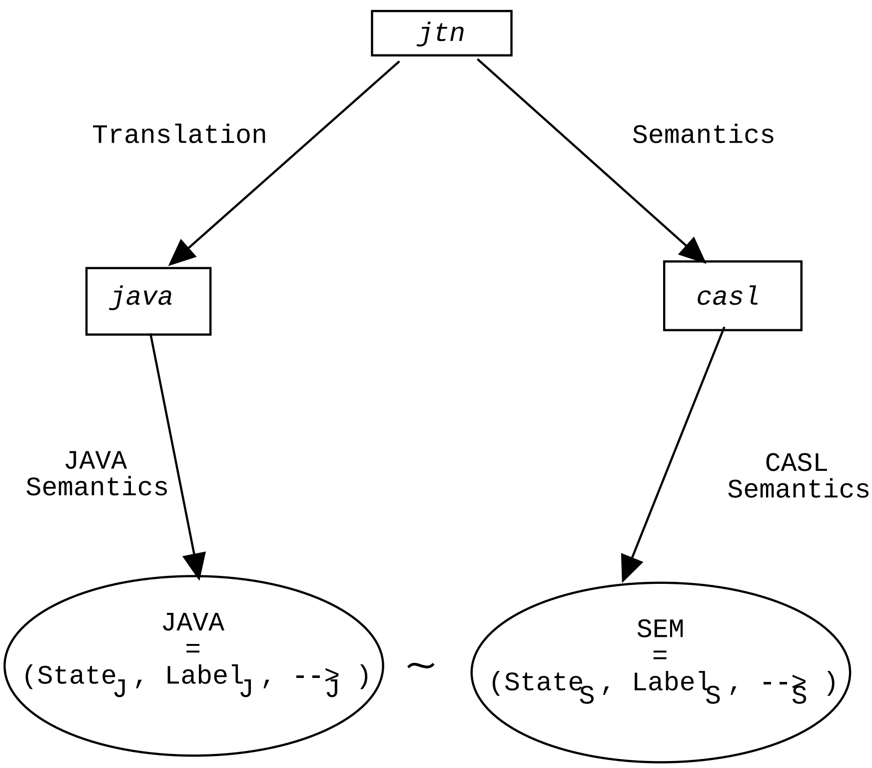


Figure 7.1: The equivalence relation for checking the translation correctness

- the semantics of Java in Chap.6 defines an *lts*. It can be easily modified to define the *glts JAVA*, by transforming a transition $t \equiv ja \xrightarrow{l} ja'$ into $i : ja \xrightarrow{l} ja'$, where i contains the complete description of the transitions performed by the threads that are executed while the system performs t.

We have defined a bisimulation relation among the states of the two *gltss*. Such relation is defined over particular "stable" states of the system, that is over states where each subcomponent (an active component in the *SEM* case and a thread component in the *JAVA* case) has the capability to start the execution of one of the activities described in the corresponding `JTN` behaviour graph.

This limitation is justified because *JAVA* contains lots of intermediate states, used to model every low-level action of the program execution, such as the locking of an object and so on. These states do not correspond to *SEM* states, where complex *JAVA* activities, like a method call or a communication, are modeled by a single, atomic transition.

On the other side, *SEM* active components have additional states and transitions that are used to model the failure of a choice within a `JTN` alternative.

Thus, the only states that appears in both *SEM* and *JAVA* and can be naturally compared are the ones in which the states of the subcomponents correspond to the states of the `JTN` graphs, that is the initial states and the ones that are reached, in both models, by "simulating" a `JTN` transition. Their characterization is given in more details in the following.

Thus, we have defined a *weak* bisimulation relation, limited to a subset of the states in the two models.

We plan to perform further investigations about the possibility to define more strong relations, avoiding the limitations to *stable states*.

We have defined the correspondence among a single *SEM* transition and a sequence of *JAVA* ones. This correspondence is called the *expansion* of a transition: given a *SEM* transition $st \xrightarrow{l}_S st'$ it returns a sequence of *JAVA* transitions $ja_1 \xrightarrow{l_1}_J \ldots\ldots \xrightarrow{l_n}_J ja_n$ such that the observation over l and $l_1, \ldots, l_n$ are the same and ja_n is equivalent to st'. It has also been defined an equivalence over sequences of transitions of the two models, using some form of re-ordering of the *JAVA* transitions, to take into account the non-determinism due to the internal concurrence of the Java programs.

We are reasonably convinced that such results could be used to define some more strong bisimulation relation, probably based on the fact that two non-stable states can be compared if are intermediate states during the execution of the same `JTN` activity. At least, we think it is possible to define some *trace equivalence* relations between the two models.

7.2 The equivalence relation

Now we present the equivalence relation based on a bisimulation relation. Our first idea is that two transition systems are equivalent if the external environment cannot discriminate between them. Recall that the labels of a *glts* describe the interactions that the system has with the external environment. Thus we we want to find a relation $\sim \subseteq State_S \times State_J$ such that $ja \sim se$ iff "we can observe the same things in their behaviours".

Such relation must have the property that:

whenever $ja \sim se$

i. if $i : se \xrightarrow{obs} se'$ then ja can perform $i' : ja \xrightarrow{\tau} \ldots \xrightarrow{obs} \ldots \xrightarrow{\tau} ja'$ and $se' \sim ja'$ (if se can perform an activity with visible information obs, ending in se', then ja can perform some activities with the same observable interactions obs and the two final states are in relation too)

ii. if $i' : ja \xrightarrow{\tau} \ldots \xrightarrow{obs} \ldots \xrightarrow{\tau} ja'$, then se can perform $i : se \xrightarrow{obs} se'$ and $se' \sim ja'$

The only interactions with the external environment, both of the Java program and of the system modeled by the `JTN` specification, are I/O operations, that are the read and write operations over files: the semantical equivalence relation is a based on the observation of these interactions.

We formalize this idea using a bisimulation technique: the two *gltss* are equivalent if the initial states are in $\sim$ relation, where $\sim$ is the union of all the *(weak) bisimulation relations* (see D for some technical recalls about bisimulations).

Def. 7.2.1 A relation $\mathcal{R} \subseteq S_1 \times S_2$ is a bisimulation relation (see [Mil90]) iff,

whenever $s_1 \mathcal{R}\, s_2$ then:

i. if $s_1 \overset{obs}{===>} s_1'$, then there exists $s_2' \in S_2$ s.t. $s_2 \overset{obs}{===>} s_2'$ and $s_1' \mathcal{R}\, s_2'$;

ii. if $s_2 \overset{obs}{===>} s_2'$, then there exists $s_1' \in S_1$ s.t. $s_1 \overset{obs}{===>} s_1'$ and $s_1' \mathcal{R}\, s_2'$;

To prove that the two *gltss JAVA* and *SEM* are equivalent, we prove that the two initial states ja_0 and se_0 are in the $\sim$ relation. This is equivalent to show a (weak) bisimulation relation $\mathcal{WB}$ and to prove that $ja_0\ \mathcal{WB}\ se_0$ are in $\mathcal{WB}$: consequently, they are in the union of all the bisimulation relations and, thus, in $\sim$.

In the following we first define what is the $\mathcal{WB}$ relation, show that it is a bisimulation and then we prove that the initial states are in such relation.

First notice that we are not comparing processes defined on a common syntax: we are considering two *gltss* that are algebras over different signatures and the observation along two observations are not the same, but have to be in some relation.

Thus we have to define:

- the observations that an external entity can make on the behaviours of the two states, thus the relations over the two *gltss* corresponding to ===> ;
- a *similarity* relation over these observations, describing when two observations are similar.

Notice that the Java programs we consider are very peculiar ones: they are programs obtained by means of the translation function applied to `JTN` specifications. Thus the states and the transitions have particular characteristics (that we will tacitly take into account in the following): for instance, the transition cannot be labeled with I/O operations over sockets; the code of the methods of an object can only assign the fields of such object, and so on. This makes easier to prove properties and so the correctness of the translation.

7.2.1 Observations and transitions relations

Now we precisely define which are the observations we are interested in, that is we define the realtions $\_ \xrightarrow{obs} \_$ and $\_ \xrightarrow{obs'} \_$ used in point $i)$ and $ii)$ of the definition of $\sim$ above, and define how to relate them.

Observations are sequences of transitions of the two *gltss*. Obviously, not every sequence can be considered a meaningful observation.

To justify our definition of observation, we introduce some considerations:

- given a `JTN` transition, its implementation in Java (its modelization by *JAVA*) is much more articulated than its Casl semantics (its modelization by *SEM*): for example, the execution of a method call is modelled by a sequence of transitions in *JAVA*, whereas it is modelled just by one transition in *SEM*;

JTN	CASL	JAVA
$s \xrightarrow{l} s'$	$se \xrightarrow{l}_S se'$	$ja_1 \xrightarrow{jl_1}_J \ldots\ldots \xrightarrow{jl_m}_J ja_m$
$s \xrightarrow{l_1} s_1$ $s \xrightarrow{l_n} s_n$	$se \xrightarrow{l_1}_S se_1$ $se \xrightarrow{fail(l_1)}_S fa_1$ $\ldots$ $fa_n \xrightarrow{l_n}_S se_n$	$ja \xrightarrow{jl^1_1}_J ja^1_1 \ldots \xrightarrow{jl^1_{h1}}_J ja^1_{h1}$ $\ldots$ $ja \xrightarrow{jl^n_1}_J ja^n_1 \ldots \xrightarrow{jl^n_{hn}}_J ja^n_{hn}$

Table 7.1: A `JTN` process capability modeled in *SEM* and *JAVA*

- *JAVA* contains lots of transitions modeling the execution of the statements composing the translation of a single `JTN` activity. These transitions have no direct correspondence in *SEM*. This is described in fig.7.1.

  Obviously, there are lots of intermediate states in *JAVA* from the starting of the method call to its conclusion, while in *SEM* the method call is sort of *atomic* activity that completes by one state transition.

  Thus, if we consider any intermediate state of *JAVA* (like ja_2 ,...,ja_{h-1} in fig.7.1), i.e., while a thread is executing a method call, such state cannot be always compared with anyone of *SEM*, because we cannot observe the state of an active component (and, consequently of the whole system) "during" a method execution.

  On the contrary, a thread that is in a state in which it can perform the corresponding of a `JTN` transition (for example ja_1 and ja^1_{h1} in fig.7.1) is always comparable with the state of the corresponding component in the CASL specification (respectively, se and se_1). In other words, supposing that the state s in fig. 7.1 is part of the `JTN` behaviour diagram for a process P, the thread object implementing P in the state s (be ja in the fig.) can be compared with the active component of *SEM* (se) that models P in the state s, because they have the same capabilities and their internal state is obviously comparable.

  Thus we require that the initial and the final states of an observation are two *stable* states, that is states in which all the internal active components are ready to perform the corresponding of a transition of the `JTN` behaviour graph. This allows to compare the *JAVA* and the *SEM* states at the end of an observation.

An observation over *JAVA* is any sequence of τ labels with one visible action, that connect two stable states. We first define the $\Rightarrow$ predicate modelling the execution of any number of internal transitions.

Now we can give some more precise definitions (in the following we assume that $ja \in$

$State_J$ and $se \in States_S$).

Def. 7.2.2 The predicate $\Rightarrow \subseteq State_J \times Label_J \times State_J$ is defined by the following inductive rules:

$$\frac{i : ja \xrightarrow{\tau} ja'}{ja \Rightarrow ja'}$$

$$\frac{ja \Rightarrow ja' \quad i : ja' \xrightarrow{\tau} ja''}{ja \Rightarrow ja''}$$

$$\frac{i : ja \xrightarrow{\tau} ja' \quad ja' \Rightarrow ja''}{ja \Rightarrow ja''}$$

Def. 7.2.3 The predicate $\underset{J}{\overset{}{===>}} \subseteq State_J \times Label_J \times State_J$ is defined by the following inductive rules:

$$\frac{i : ja \xrightarrow{l} ja'}{ja \underset{J}{\overset{l}{===>}} ja'}$$

$$\frac{ja \Rightarrow ja' \quad ja' \underset{J}{\overset{l}{===>}} ja''}{ja \underset{J}{\overset{l}{===>}} ja''} \quad \text{if } StableState(ja) \text{ and } StableState(ja'')$$

$$\frac{ja \underset{J}{\overset{l}{===>}} ja' \quad ja' \Rightarrow ja''}{ja \underset{J}{\overset{l}{===>}} ja''} \quad \text{if } StableState(ja) \text{ and } StableState(ja'')$$

where *StableState* is the predicate that checks if a state is a stable one.

To give a more precise definition of *JAVA* stable state we observe that a thread component in *JAVA* executes a `run` method that is a infinite `while` loop checking the current thread state. Depending on the kind of state, it executes a branch of a `switch` statement that implements the transition capability described for such state in the behaviour graph of the corresponding process in the `JTN` specification. Thus a *JAVA* state is stable if no thread is executing a branch of the `switch` statement, that is iff:

$ThreadComponents(ja) = tc_1, \dots, tc_n$ and $StableSubstate(tc_i)$ for $i = 1, \dots, n$

where $StableSubstate(< t_{id}, c, ... >)$ iff $c=$ `while`(true) { `switch`(`kind`)...

Now an *observation* on the *JAVA glts* is represented by $ja \underset{J}{\overset{l}{===>}} ja'$, that is a sequence of several internal transitions and one transition with a visible interaction, connecting two stable states.

Analogously, the observations over the elements of $State_S$ are defined by a new predicate $\underset{S}{===>}$, defined below.

Def. 7.2.4 The predicate $\underset{S}{===>} \subseteq State_S \times Label_S \times State_S$ is defined by the following inductive rules:

$$\frac{i : se \ \xrightarrow{l} se'}{se \ \underset{S}{\overset{l}{===>}} se'} \quad \text{if } StableState(se\ ) \text{ and } StableState(se'\ )$$

$$\frac{se \ \Rightarrow se' \quad se' \ \underset{S}{\overset{l}{===>}} se''}{se \ \underset{S}{\overset{l}{===>}} se''} \quad \text{if } StableState(se\ ) \text{ and } StableState(se''\ )$$

$$\frac{se \ \underset{S}{\overset{l}{===>}} se' \quad se' \ \Rightarrow se''}{se \ \underset{S}{\overset{l}{===>}} se''} \quad \text{if } StableState(se\ ) \text{ and } StableState(se''\ )$$

A *SEM* state is stable if all its active components are able to perform the corresponding of a `JTN` transition. On the contrary, an active component ac is non-stable if it models a state reached after the failure of a communication within an alternative of the `JTN` specification.

Let us recall that a `JTN` alternative (see fig.7.1) is modeled, in the corresponding CASL specification, by several axioms stating that from state se there is the transition $se \ \xrightarrow{l_1}_S se'$ modeling the successful execution of l but also the transition $se \ \xrightarrow{fail(l_1)}_S fa_1$ modelling the failure of l. Again, the state fa_1 has the capability of performing l_2 or failing again, and so on. Obviously fa_1 does not correspond to any `JTN` state, thus it is not a stable state, whereas se is.

But in the CASL specification we use an integer encoding which are the choices of an alternative that a state has already performed. If it is 0, the state is a stable state.

Hence:

$StableState(se\ )$ if ActiveComponents(ja) = ac_1 , ... ,ac_n , and

$StableSubstate(ac_i\ )$ for i=1, ... ,n

where

$StableSubstate(ac\ )$ iff there exist con, e_1, ... , e_n s.t. $ac = con(e_1, \ldots ,e_n,0)$

The sequence of info-labels [1] $l_1, \dots, l_n$ is a *transition sequence* for the observation $st \overset{l}{\underset{S}{===>}} st'$ iff $st \xrightarrow{l_1} \dots \xrightarrow{l_n} st'$ and l is the only visible transition in $l_1, \dots, l_n$.

7.2.2 Weak bisimulation

We are interested in distinguishing models that differ in the way they interact with the external environment, i.e. if they receive different inputs or do not produce the same output. On the contrary, we do not care if their internal states are different but this cannot be checked from the external. Thus we need a semantical equivalence that is less fine then strong equivalence: a CASL specification and a Java program are equivalent if the corresponding *gltss* behave in the same way, when we observe the non internal actions that label their transitions.

Two elements are *weakly bisimilar* (see App.D) if there exists a weak bisimulation relation that contains them.

To define which states are in bisimulation relation, we observe that, given a `JTN` specification *jtn* :

- a generic state *ja* of the Java program translating *jtn* has much more components then the corresponding state *se* of the CASL specification. Indeed, *ja* has components describing:
  - process instances;
  - object instances;
  - values (instances of classes implementing datatypes);
  - classes (that receive method calls and contain the values of the class fields [2]);
  - threads and objects implementing `JTN` connectors;

  a generic state *se* of the CASL specification has components describing:
  - process instances;
  - object instances;
  - `JTN` connectors;

[1] The term info-label denotes a couple *il*= (l,i) labelling a transition of a *glts* where l is the label describing the system interaction with the external environment and i is the information component of the transition.

[2] let us recall that, even if `JTN` does not support the specification of class fields, class fields are used to implement the values of an enumeration type; analogously, class methods are used to implement some datatype operations.

- in `JTN` (and, consequently, both in the specification giving its semantics and in the Java program translating it) there are neither object nor process creations; thus, given a `JTN` specification, the number of components representing instances of kind object and process in the elements of $State_S$ and $State_J$ is fixed.

A tentative definition of the bisimulation relation (called $\mathcal{WB}$ from here on) is as follows.

Let ($\mathsf{ocomp}_1, \ldots, \mathsf{ocomp}_n$), ($\mathsf{pcomp}_1, \ldots, \mathsf{pcomp}_m$) and ($\mathsf{coop}_1, \ldots \mathsf{coop}_k$) be, respectively, the components of kind *object*, of kind *process* and the *cooperations* in the architecture diagram of *jtn* , then $se\ \mathcal{WB}\ ja$ (with $se \in State_S$ and $ja \in State_J$) iff:

- in *se* there is a component for each object, process and connector above, whereas in *ja* there is a component for each object and process, a set of components for each connector and some components describing values of datatypes;
- there is a correspondence among any object component of *se* and an object component of *ja* and viceversa; the same for the process components, whereas the datatype values of *ja* are ignored;
- every connector component (modelling a synchronous or asynchronous channel) in *se* corresponds to a set of components in *ja* and viceversa.

More precisely (see fig.7.2), $se\ \mathcal{WB}\ ja$ iff:

$se = oc_1 \mid \ldots \mid oc_n \mid ac_1 \mid \ldots ac_m \mid ch_1 \mid \ldots \mid ch_k$
where

- oc_i is an element of a sort corresponding to an *object* class of the `JTN` diagram;
- ac_i is an element of a dynamic sort corresponding to a *process* class of the `JTN` diagram;
- ch_i is an element of a sort modelling synchronous and asynchronous channels or connectors;

and $ja = joc_1 \mid \ldots \mid joc_n \mid tc_1 \mid \ldots tc_m \mid chS_1 \mid \ldots \mid chS_k \mid dv_1 \mid \ldots \mid dv_h$
where

- joc_i is an instance of a class corresponding to an *object* class of the `JTN` diagram;
- tc_i is an instance of a thread class implementing a *process* class of the `JTN` diagram;
- chS_i is a set of instances of particular Java classes needed to implement I/O operations.

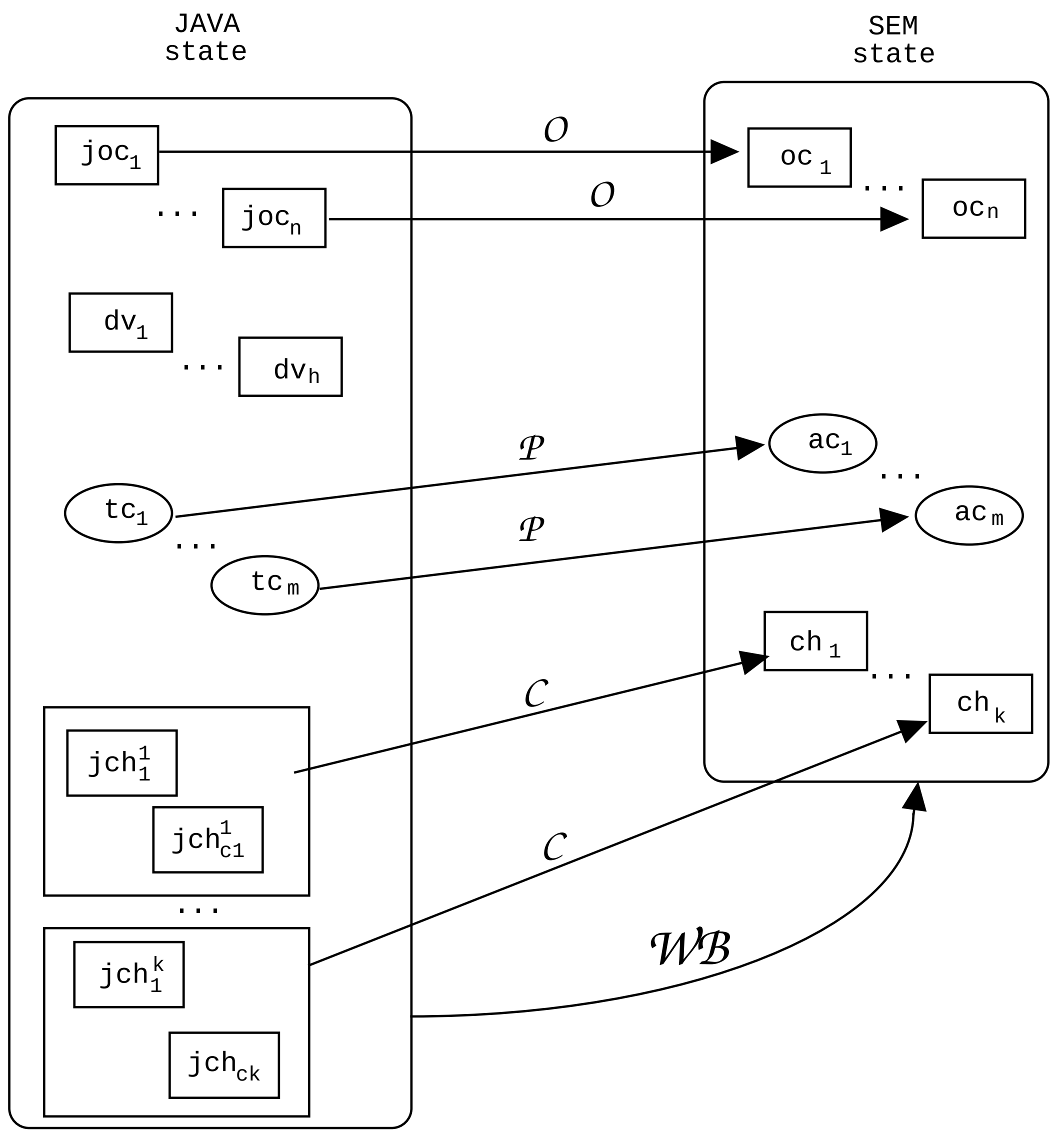


Figure 7.2: Bisimilar states

- dv_i is an instance of an object class implementing a *datatype* class of the `JTN` diagram;

Def. 7.2.5 The relation $\mathcal{WB} \subseteq State_J \times States_S$ is defined as follows:

$\boxed{\boldsymbol{ja\ \mathcal{WB}\ se}}$ iff

- there exists a correspondence among the object components of ja and the ones in se s.t. for $i = 1, \ldots, m$, given oc_i we can find $j \in \{1, \ldots, m\}$ s.t. $oc_i\ \mathcal{O}\ joc_j$ and viceversa;
- there exists a correspondence among the process components of ja and the ones in se s.t. for $i = 1, \ldots, m$, given ac_i we can find $j \in \{1, \ldots, m\}$ s.t. $ac_i\ \mathcal{P}\ tc_j$ and viceversa;
- analogously, there exists a correspondence among a set of instances of Java standard I-O classes in ja and a component in se s.t. for $i = 1, \ldots, k$, given $ch_i \in se$, we can find $chS_j \in ja$ s.t. $ch_i\ \mathcal{C}\ chS_j$

where the definitions of the relations $\mathcal{O}$,$\mathcal{P}$ and $\mathcal{C}$ are given in more details in the following.

Notice that the relation depends on the two states (se and ja) that we are examining: for example, a value dv in ja and a 0-ary constructor $con()$ in se , representing a constant value of class `C` are in relation if in ja there exists also the class `C`, with a field referencing dv.

Thus, we define the relations $\mathcal{O}_{(ja,se)}$,$\mathcal{V}_{(ja,se)}$, $\mathcal{P}_{(ja,se)}$ and $\mathcal{C}_{(ja,se)}$ depending on two states of *SEM* and *JAVA*, even if their names are omitted when do not cause confusion

Value relation Informally, a *JAVA* component dv corresponds to a CASL value `v` if the values of the fields of dv correspond to the encoding of the `v` constructor and to the values of its arguments.

More precisely

- $< did, m >\ \mathcal{V}\ con(e_1, \ldots, e_n)$ iff:
  - the value of the field `kind` encodes the constructor con; if $m(\texttt{kind}) = k$ (recall that did is the identity, m is the memory function) then k is the encoding in *JAVA* of the constructor con of the `JTN` specification, as defined by the translation relation (it is the k-th constructor listed in the corresponding interface diagram of the `JTN` specification).

– the values of the fields $f_1, \ldots, f_n$ (that are meaningful w.r.t. the given `kind` value, thus f_i is the field implementing the i-th argument of the constructor *con*, see 5.1) are equivalent to the arguments of the datatype constructor: for each $f_j \in \{f_1, \ldots, f_n\}$ there exists only one $e_i \in \{e_1 \ldots e_n\}$ s.t. $m(f_j)\ \mathcal{V}\ \ e_i$ and viceversa.

otherwise (the constructor has been implemented by a class field in Java)

- $< o_{id}, m >\ \mathcal{V}\ con()$ with m the empty function (o_{id} has no field) and Class(o_{id})=`C` and $< \mathtt{C}, m, ... > \in ja$ and $m(con) = o_{id}$(the object is the value of the class field describing the 0-ary value);

Object relation Informally, *oc* and *joc* , corresponding to object instances in the `JTN` specification, are similar if the values of the fields are the same.

More precisely, $oc\ \ \mathcal{O} < o_{id}, m >$ iff:

- for each f_i, $m(f_i)=did$ and $dv=<did,m>\in ja$ [3] and $dv\ \mathcal{V}\ f_i(oc)$.

where $f_1, \ldots ,f_n$ are the fields of the `JTN` object modelled by *oc* and *joc* .

Connector relation The definition of the relation $\mathcal{C}$ among the CASL semantics and the Java implementation of a connector is a bit more articulated that the previous ones, as the typology of the connectors (and its Java implementation) is varied. In table7.3 we examine the various connector. The schema is: given the `JTN` connector depicted in the first column, we describe in the second one which is the CASL semantics (the term *ch* that is part of the system state) and in the last one which the Java implementation that are the set of instances *chS* , that is in relation with the CASL term ($ch\ \mathcal{C}\ chS$).

In the table, $ch(ch')$ means that the instance *ch* has been constructed over ch' (if *ch* is an object stream instance) or that *ch* is pipelined to channel ch' (if *ch* is a pipeline stream); SI denotes the standard input channel.

Process relation Informally, two components *ac* and *tc* , corresponding to process instances of the `JTN` specification, are similar if the state of *tc* is the translation of the pattern expression *ac* .

More precisely, $con(e_1, \ldots ,e_n,i)\ \mathcal{P} <t_{id},c,m>$ (t_{id} is the identity, c is the code to be executed and m is the memory function) iff:

[3] recall that the relation $\mathcal{O}$ depends on a a global *JAVA* state *ja* and a global *SEM* state *se*

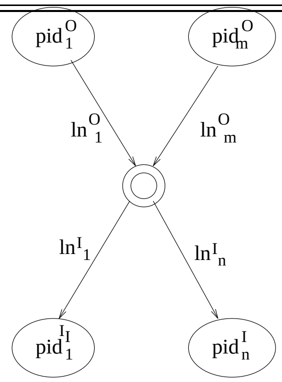
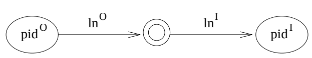
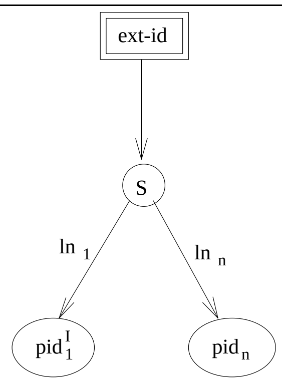
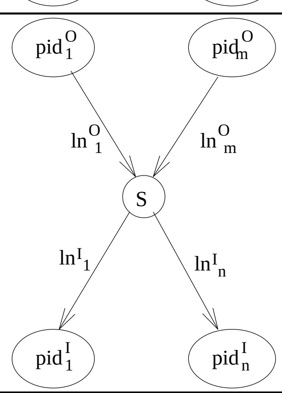
JTN	CASL	JAVA
	ch: SigSyChan	$chS = \{ch\colon \texttt{MultSigChan}\}$ or $chS = \{ch\colon \texttt{MultMsgChan}\}$
	ch: SigSyChan	$chS = \{ch\colon \texttt{OneSigChan}\}$ or $chS = \{ch\colon \texttt{OneMsgChan}\}$
	NONE	$chS = \{ch\colon \texttt{ObjectInputStream}\}$
	ch: $\texttt{Achan}[T]$	$chS = \{$ $ch^I(p^I)$: $\texttt{ObjectInputStream}$, p^I: $\texttt{PipedInputStream}$, $ch^O(p^O)$: $\texttt{ObjectOutputStream}$, $p^O(p^I)$: $\texttt{PipedOutputStream}$, $\}$

Table 7.2: Definition of the $\mathcal{C}$ relation (part1)

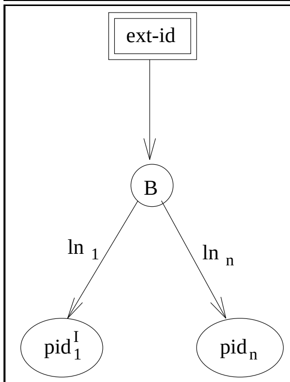
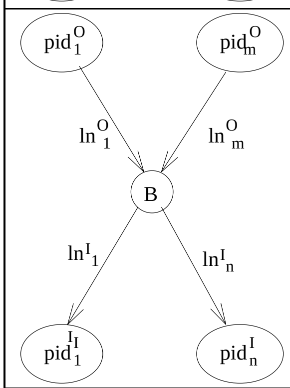
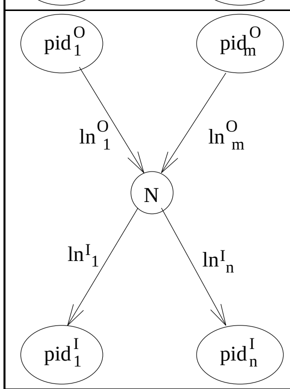
JTN	CASL	JAVA
	ch: `Bchan`$[T]$	$chS = \{$ ch: `ObjectInputStream`(SI) $ch^I{}_1(p^I{}_1), \ldots, ch^I{}_n(p^I{}_n)$: `ObjectInputStream`, $p^I{}_i$: `PipedInputStream`, $ch^O{}_1(p^O{}_1), \ldots, ch^O{}_m(p^O{}_m)$: `ObjectOutputStream`, $p^O{}_i(p^I{}_i)$: `PipedOutputStream`, c: `BConnector`$(ch^O{}_1(p^O{}_1), \ldots, ch^O{}_m(p^O{}_m))\ \}$
	ch:`Bchan`$[T]$	$chS = \{$ $ch^I{}_1(p^I{}_1), \ldots, ch^I{}_n(p^I{}_n)$: `ObjectInputStream`, $p^I{}_i$: `PipedInputStream`, $ch^O{}_1(p^O{}_1), \ldots, ch^O{}_m(p^O{}_m)$: `ObjectOutputStream`, $p^O{}_i(p^I{}_i)$: `PipedOutputStream`, c: `BConnector`$(ch^O{}_1(p^O{}_1), \ldots, ch^O{}_m(p^O{}_m))\ \}$
	ch:`NAchan`$[T]$	analogous

Table 7.3: Definition of the $\mathcal{C}$ relation (part2)

- the value of the `kind` field encodes the constructor *con*; if $m(\texttt{kind}) = k$ then k is the encoding in CASL of the constructor *con* of the `JTN` specification (it is the k-th constructor listed in the corresponding behaviour diagram of the `JTN` specification).

- the values of the fields $f_1, \ldots, f_n$ (that are meaningful w.r.t. the given `kind` value) are equivalent to the arguments of the object constructor: $m(f_1)\ \mathcal{V}\ \ e_1 \ldots m(f_n)\ \mathcal{V}\ \ e_n$

- the value of i is related to the code c that the thread has to execute; we distinguish two cases:

  - if $i > 0$ the constructor describes a state of the `JTN` process that has an alternative of n transitions that have to be tried in a given order and that the first i alternatives (synchronous communications or asynchronous read operations) have failed;

    Let us consider, for example, the case of h read alternatives over ach_id_1, ... ,ach_id_n: the translation function, applied to the `JTN` term returns:

    `bool exit = false;`
    `while (!exit)` $\{c_1 \ldots c_h\}$

    $$\text{where } c_i\!= \begin{cases} T_i\ \texttt{x}_i; \\ \texttt{synchronized}(ach\_id_i)\{ \\ \texttt{if}(ach\_id_i.\texttt{available()} > 0\ )\{ \\ \texttt{x}_i\!=\ (T_i)ach\_id_i\texttt{.readObject();} \\ \mathrm{TJ}_{State}(st,st_i)[\rho]; \\ \texttt{exit = true;}\}\} \end{cases}$$

    Thus i is in relation with c if it is the part of code from the i-th `if` statement: $c\!=\ c_{i+1} \ldots c_h$

  - If $i = 0$ the constructor describes either a state that has n $(n > 1)$ alternatives and has to try the first one (it is a particular case of the situation described above), or a state that has exactly one of the following transition capabilities:

    * a method call: $con() \xrightarrow{o_{id}.\texttt{m}(par\_L)} con'(\ldots)$: its translation is the encoding of the method call plus the code that assigns to the thread state the equivalent of $con'(...)$;
    * a single synchronous communication; its translation is the encoding of the read/write method of the synchronous connector, plus the code setting the new state.
    * a single asynchronous communication; analogous to the previous case.

Thus, c is in relation with i if c is the code implementing the alternative described by the value of i, as shown above.

7.2.3 Similarity of observations

Now, we have to prove that the previously defined relation is a *weak bisimulation*. Thus we prove point *i)* and *ii)* in Def.7.2.1, where $\mathcal{R} \equiv \mathcal{WB}$.

First we introduce some considerations and lemmas.

Intuitively, an observation on *JAVA* can be simulated by one of *SEM* (and viceversa) that contains the same visible activities performed by the corresponding components and s.t. the final states of the components are equivalent. Thus it is necessary, given an internal transition of the whole system, to know which are the active components or threads that are moving and which are the performed activities. But recall that there is no one-to-one correspondence between the transitions of the two *gltss*.

The idea is that the activities of the two *gltss* are obviously the same and corresponds to the ones described in the `JTN` specification. If a state of *SEM* can call a method, perform a communication or failing its immediate completion, we expect to find something that corresponds to it in the *JAVA glts*.

A transition (called a *complete transition*, be α) of an active component *ac* of *SEM* corresponds to several transitions (called *small steps* $\alpha_1, \ldots, \alpha_n$) performed by the corresponding thread component *tc* of *JAVA*. Thus an observation obs_{se} from $se \in SEM$ is simulated by obs_{ja} from $ja \in JAVA$, if the transitions in obs_{ja} are exactly the small steps of the complete transitions of obs_{se} .

But we have to remember that the threads in a Java program are executed concurrently: the single steps α_i of a thread activity may be distributed along an observation and interleaved with the small steps of other complete transitions, performed by different concurrent threads.

If an observation *obs* of *JAVA* contains the sequence of transition labels:

$s \equiv \alpha_1 \ldots \beta_1 \ldots \gamma_1 \ldots \alpha_i \ldots \beta_i \ldots \gamma_i \ldots \ \ldots \alpha_n \ldots \beta_m \ldots \gamma_k$[4],

it is obvious that it is not easy to understand which could be the similar observations over *SEM*. So we want to prove that the execution of s is equivalent to the execution of another sequence s_{ord} corresponding to the same observation *obs* (i.e. with the same arriving state and visible label). We construct s_{ord} because it is easy to deduce from it which is the observation on *se* that is similar to *obs*: s_{ord} is obtained simply re-ordering the labels, in such a way that all the labels representing the small steps of a *SEM* transition are in sequence.

Now, if we are able to prove that the reordering of the steps does not change the final state

[4]From now on, when the internal states of the subcomponents are not meaningful, an observation is identified by the sequence of the transition labels of the subcomponents.

of the observation, the correspondence of $s_{ord} \equiv \alpha_1 \ldots \alpha_n \beta_1 \ldots \beta_m \ldots \gamma_1 \ldots \gamma_k$ (obtained reordering s) with the sequence of *SEM* transition labels $s_{SEM} \equiv \alpha\beta\gamma$ is much more intuitive and easy to prove.

In the following we want to show, in more details:

- which is the correspondence of a *SEM* label α with a sequence of *JAVA* labels; this is called the *expansion* of α.
- the equivalence of a *JAVA* transition sequence s with the "reordered" one s_{ord}; this requires to show some properties about the *true concurrency* of some transitions.

7.2.4 Expansion

First observe that the semantics of a `JTN` transition $t \xrightarrow{l} t'$ is a single transition of *SEM* (let us forget `JTN` alternatives, for the moment, and just consider method calls, blocking synchronous and asynchronous communication and non blocking asynchronous writing operations), whereas its translation into Java returns:

- some statements implementing the activity described by l;
- some statements implementing the state modifications described by t'.

Thus, if $se\ \mathcal{WB}\ ja$, the expansion of a transition $i : se \xrightarrow{l} se'$ of *SEM* is a sequence of transitions of *JAVA* that, starting from ja, implement the same activity.

If the transition is not visible outside ($l \equiv \tau$), all the information about the activity is given by the information component i. Thus the definition of such expansion must take into account the sub-components activities.

If $\mathrm{T} \equiv i : se \xrightarrow{l} se'$ and ac is the only active component into i and it performs α, the expansion of T are the system transitions resulting from the execution of $\mathsf{Expr}(\alpha)$ by tc (corresponding to ac).

For each kind of process transition in a `JTN` behaviour graph, we analyse which are the corresponding *SEM* and *JAVA* transitions. Thus we consider a transition T in the behaviour diagram of a `JTN` process p and we analyse which are the corresponding transitions in the *ltss* modelling p in *SEM* and in *JAVA*.

Let us suppose that ac is the active component of *SEM* and tc is the thread component of *JAVA* modelling p in a state st.

The only cases in which the system activity is the result of the activities of more then one process subcomponent is the case of a synchronous read or write operation. We define the expansion apart, not as the result of combining the expansion of a subcomponent activities.

In the other cases $i \equiv (ac,\ l,\ ac')$, where ac is the identity and l is the label of a transition performed by the active component ac . If T $\equiv$ $i : se \xrightarrow{l} se'$ is a synchronous communication, we define the expansion directly, not as a result of a single subcomponents activity.

In table 7.5, the first column reports a transition (or a transition alternative) for st , the second one the labels of the corresponding ac transition, the third one the labels of the transitions performed by tc during the execution of the implementation of T, as returned by the translation function. Such labels describe the expansion of the *SEM* transition. The precise description of the expansion function, and of the meaning of table 7.5 is in Sect.7.3.

From the observation above we have shown that, given for example $se \xrightarrow{(l,i)} se'$ [5] in *SEM*, the information i is enough to calculate the correspondence with the transitions $ja_1 \xrightarrow{(l_1,\mathrm{i}_1)} ja_2 \ \ldots ja_n \xrightarrow{(l_n,\mathrm{i}_n)} ja_{n+1}$ in *JAVA*.

Thus, we can use a function (whose formal definition is omitted here but easily comes from the previous observations) *Exp* that, given a transition with label α of *SEM* returns the corresponding list of (info)labels describing the behaviour of a *JAVA* state when executing the code that implements the same activity.

If $Exp(\alpha) = \alpha_1, \ldots, \alpha_n$, we say that a list $\alpha_1, \ldots, \alpha_n$ of small steps is the *expansion* of the *SEM* label α.

7.2.5 Reordering

If $il_1 \ldots il_n$ is the sequence of info-labels in an observation $ja \overset{l}{\underset{J}{===>}} ja'$ for *JAVA*, we can construct an analogous sequence with the same external interactions and the same arriving state, obtained by grouping together the transitions in $il_1 \ldots il_n$ that expand the same *SEM* transition.

We use the two lemmas below. The first one ensures that every re-ordered sequence is the expansion of (thus it is equivalent to) the transitions in an observation over *SEM*.

The second one gives the conditions to exchange two transitions along an observation: we prove that if they cannot be exchanged, they are in the right order for the reordered

[5] $se \xrightarrow{(l,i)} se'$ is just an alternative notation for $i : se \xrightarrow{l} se'$

sequence.

Lemma 7.2.6 The sequence of transitions of an observation $ja \overset{l}{\underset{J}{===>}} ja'$ over *JAVA* exactly corresponds to the expansion of a finite number of *SEM* transitions.

Proof We want to prove that if the labels of the transitions in $ja \overset{l}{\underset{J}{===>}} ja'$ are $il_1, \ldots .il_n$, then exists a finite number of *SEM* labels α,β, $\ldots\gamma$ s.t.

$Exp(\alpha) \cup Exp(\beta) \cup \ldots \cup Exp(\gamma) = \{il_1, \ldots .il_n\}$.

Let us remember that if $ja \overset{l}{\underset{J}{===>}} ja'$ is an observation, every thread component in ja' is in a stable state, that is it is starting the execution of the `switch` statement of the `run` method (that is it has just completed the implementation of a `JTN` transition). It is easy to prove that the execution of the `switch` block corresponds to the execution of a method call or of an external or internal communication: a *JAVA* state can reach a stable state only by executing a finite number of these activities, each one corresponding to a *SEM* transition
□**end lemma**

In the following we use two functions over the labels of *JAVA*:

- Thread(il) returns the identity(ies) of the threads (called *active thread(s)*) participating in the program transition.
- Object(il) returns the identity of the object involved in the program transition described by il.

Lemma 7.2.7 (Pure Concurrency) Given two (info)labels il_1 and il_2, if

- the thread performing il_1 is not the thread performing il_2($Thread(il_1) \neq Thread(il_2)$),
- the object modified by il_1 is not the object modified by il_2($Object(il_1) \neq Object(il_2)$),
- il and il' are not both external interactions,

then the two labels are in *pure concurrency* and can be exchanged: if $ja_1 \xrightarrow{il_1} ja_2 \xrightarrow{il_2} ja_3$ then $ja_1 \xrightarrow{il_2} ja_2' \xrightarrow{il_1} ja_3$

Proof The proof is quite obvious. After the execution of $ja_1 \xrightarrow{il_1} ja_2$, from the conditions above we have that the only modified components of ja are the thread component (say tc_1) and the object component (joc_1). The execution of il_2 from ja_2 only modifies a thread component $tc_2 \neq tc_1$ and an object component $joc_2 \neq joc_1$, thus the execution of the two transitions has no relation and can be done in pure concurrency, in any order.

□**end lemma**

Now we prove that the conditions of the lemma 7.2.7 above are sensible: we prove that α_i and β_i are the labels of two subsequent transitions within an observation, then they belong to the expansion of two independent activities and, thus, can be exchanged only in the case described by the condition above.

First we observe that:

Lemma 7.2.8 Every transition in *SEM*, and, thus, every expansion by means of several transitions in *JAVA*, involves just one object, that is just one object has its field read or modified.

Proof Every *SEM* transition is:

- an object method call by an active component: in *SEM* the called object is the only one involved; in fact the method application is a sequence of field assignments, that can contain an operation application operating on datatype values or a call of a method of the same object. In *JAVA* a method execution can be:
  - a class method that corresponds to an operation of a `JTN` datatype class, but an operation returns a value and does not contain assignments: its translation does not modify any object but simply calculates a value. Thus, the execution of a class method or of a method corresponding to a `JTN` datatype operation does not change the value of any object of the program;
  - a method of an object class that can only modify the called object, because it contains only:
    - calls to method of the same object;
    - assignments to fields of the same object;
    - assignments to local variables;
  - a method implementing a datatype operation can only return a value, it does not contain assignments.
- the synchronization of two active components; both the *SEM* transition and the corresponding expansion by *JAVA* transitions only involves the object implementing the channel;

- the asynchronous read/write operation over a channel: both the *SEM* transition and the corresponding expansion by *JAVA* transitions involves only the component and the object representing the channel.

$\square$**end lemma**

Now we prove that two labels can be exchanged if they belong to the expansions of two independent activities, that is performed by different components and involving different objects.

Lemma 7.2.9 If the labels of the transitions in the observation $ja \overset{l}{\underset{J}{===>}} ja'$ are $il_1 \ldots \alpha_i\beta_i \ldots il_n$, and if

i. $Thread(\alpha_i) = Thread(\beta_i)$ (α_i and β_i are executed by the same thread) or

ii. $Thread(\alpha_i) \neq Thread(\beta_i)$ and $Object(\alpha_i) = Object(\beta_i)$ or

iii. $Thread(\alpha_i) \subseteq Thread(\beta_i)$ or $Thread(\beta_i) \subseteq Thread(\alpha_i)$

then α_i and β_i cannot be exchanged.

Proof

Case 1 Every thread has only one execution flow and it cannot execute two independent methods at the same time, thus:

- α_i and β_i operate and possibly modify the same object and cannot be exchanged;
- α_i and β_i operate on different objects and, from Lemma7.2.8 they are, respectively, the last step in the expansion of a transition of α and the first step of the expansion of a transition β and it is reasonable that they are not exchanged during the re-ordering process.

Case 2 Every method, from the definition of the translation function, is synchronized on the called object, thus two threads cannot execute two methods of the same object at the same time, except if α_i is the end of the method execution by a thread and β_i is the start of the execution of another method of the same object by the other thread, thus α_i and β_i belongs to the expansion of different *SEM* transitions and cannot be exchanged;

Case 3 A system transition in *JAVA* involves more then one thread only in the case of a synchronized communication in which a thread component tc_1 sends a notify signal to another suspended thread tc_2 . If the two threads synchronizes in α_i, then both the threads have completed the communication, thus α_i is the last step of the synchronized communication and β_i is a step of a new activity of one of the two threads and the transitions cannot be exchanged. Otherwise the synchronization happens in β_i, and α_i is a previous activity of one of the two threads, thus, again, they cannot be exchanged.

□**end lemma**

7.2.6 Bisimulation Proof

Using the considerations and lemmas above, we can prove that the relation $\mathcal{WB}$ is a bisimulation.

Lemma 7.2.10 The relation $\mathcal{WB} \subseteq State_S \times State_J$ is a *weak bisimulation* relation, that is:

if $ja\ \mathcal{WB}\ se$ then

i) for all $l \in Label_S$, $se' \in State_S$, if $se \overset{l}{\underset{S}{===>}} se'$ there exists $l' \in Label_J$, $ja' \in State_J$

s.t. $ja \overset{l'}{\underset{J}{===>}} ja'$ and $se'\ \mathcal{WB}\ ja'$;

ii) for all $l \in Label_S$, $ja' \in State_J$, if $ja \overset{l}{\underset{J}{===>}} ja'$ there exists $l' \in Label_J$, $se' \in State_S$

s.t. $se \overset{l'}{\underset{S}{===>}} se'$ and $se'\ \mathcal{WB}\ ja'$

Proof

The proof, for both the implications, is by induction on the length of the observations.

i) **proof**

We prove that if $se\ \mathcal{WB}\ ja$ and the observation $se \overset{l}{\underset{S}{===>}} se'$ corresponds to the execution of the transitions labelled by $il_1, \ldots ,l, \ldots ,il_n$, then:

- ja can perform the transitions labelled by: $Exp(il_1), \ldots ,Exp(l), \ldots ,Exp(il_n)$ ending in ja' .

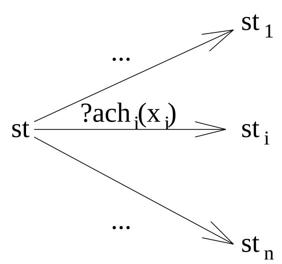
JTN	CASL	JAVA
	Method Call	
$st \xrightarrow{o.\mathtt{m}(e_1;\ldots;e_k)} st'$ and the method code is $\mathtt{m}(pt_1;\ \ldots\ ;\ pt_k)$, $(bexp_1, stms_1)$ $;\ \ldots\ ;$ $(bexp_n, stms_n))$;	$\mathtt{call}(o.\mathtt{m}(e_1,\ \ldots\ , e_k))$	$\tau_{Eval(e_1 \ldots e_n)}$ $CallO(o, \mathtt{m}, e_1,\ \ldots\ , e_k, \text{code}(\text{m}))$ $ll_{stms_1} \ldots ll_{stms_n}$ ll_{return} $ll_{st'}$
	Asynchronous simple write	
$st \xrightarrow{!ach\_id(e)} st'$	$!\,ach\_id(e)$	$\tau_{Eval(e)}$ $WriteO(ach\_id, Eval(e))$ $ll_{st'}$
	Alternative of asynchronous simple read	
	$\mathtt{No?}ach\_id_i()$	$Lock(ach\_id_i)$ $ll_{Available}$ τ_{if} $Unlock(ach\_id_i)$
	$?\,ach\_id_i(x_i)$	$Lock(ach\_id_i)$ $ll_{Available}$ τ_{if} $ReadO(ach\_id_i, o_{id})$ $ll_{st'}$ τ_{exit} $Unlock(ach\_id_i)$
	Asynchronous named read	
analogous	analogous	analogous

Table 7.4: Expansion of a JTN transition (part1)

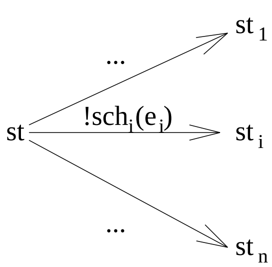
JTN	CASL	JAVA
Single synchronous communication		
$st \xrightarrow{!sch\_id(e)} st'$	$!sch\_id(e)$	$\tau_{Eval(e)}$ $CallO(sch\_id,$`write`$,\dots)$ $Lock(sch\_id)$ $ll_{set(msg)}$ ll_{Check_ready} ll_{Set_ready} ll_{Notify} $Unlock(sch\_id)$ $ll_{st'}$
	$!!sch\_id(e)$	$\tau_{Eval(e)}$ $CallO(sch\_id,$`write`$,\dots)$ $Lock(sch\_id)$ $ll_{set(msg)}$ ll_{Check_ready} ll_{Set_ready} ll_{Wait}
	`resumed`(sch_id)	$Unlock(sch\_id)$ $ll_{st'}$
Alternative of synchronous communications		
	No$(!sch_id())$	$CallO(sch\_id,$`writeAvailable`$,$ $\dots)$ $Lock(sch\_id)$ ll_{Check_ready} $Unlock(sch\_id)$ $ll_{returnfalse}$
	$!sch\_id(e_i)$	$CallO(sch\_id,$`writeAvailable`$,$ $\dots)$ $Lock(sch\_id)$ ll_{Check_ready} ll_{Set_ready} ll_{Notify} $\tau_{Eval(e_i)}$ $ll_{set(msg)}$ $Unlock(sch\_id)$ $ll_{returntrue}$ ll_{st_i}

Table 7.5: Expansion of a `JTN` transition (part2)

- ja' is bisimilar to se' .

It easy to see that the execution of such transitions corresponds to an observation

$ja \overset{\hat{l}}{\underset{J}{===>}} ja'$ with $l\ \mathcal{L}\ \hat{l}$ and $ja'\ \mathcal{WB}\ se'$.

In fact:

- there is a visible label, in $Exp(l)$; this comes from the definition of the *Exp* function, that transforms an internal label of *SEM* in a sequence of internal labels of *JAVA* and a visible label of *SEM* in a sequence of labels of *JAVA*, with just a visible one (see Sect.7.3);

- ja' is a stable state: the proof is not so immediate, but it is based on the fact that se and se' are stable states, thus the intermediate activities corresponds to the execution of one or more `JTN` activities, that is method calls, I/O communications and also multiple alternatives. In the last case, we are sure (from the definition of SEM stable state) that one of the alternatives has been successfully completed.

  The *Exp* function applied to these *SEM* transition labels, returns the sequence of activities corresponding to the execution of the statements associated with a `switch` case of the `run` method of a thread in ja. Thus, after their execution the thread has to restart the `while` cycle again.

To proof this last point, we proof a more restrictive property that consider the execution of just one transition by se . From such property it easily follows that any observation *obs* of *SEM* can be simulated by the similar one *obs*$'$ obtained by concatenating the transitions that expand each single transition in *obs*, such that the corresponding final states are similar: if the sequence of transitions in *obs* is: $\alpha\beta\gamma$ ...then the sequence in *obs*$'$ is $Exp(\alpha)Exp(\beta)Exp(\gamma)$ ... .

Lemma 7.2.11 If $ja\ \mathcal{WB}\ se$ and $se \xrightarrow{\alpha} se'$ and se' is a stable state, then $ja \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_n} ja'$ ($Exp(\alpha) = \alpha_1 \ldots \alpha_n$) and ja' is a stable state *st* $ja'\ \mathcal{WB}\ se'$.

Proof We prove that every single transition from se can be simulated by one or more from ja and that the final states are weakly bisimilar. Every *SEM* transition ending in a stable state is the result of the execution of one of the following activities:

i. external asynchronous communication;

ii. method call;

iii. internal asynchronous communication;

iv. synchronous communication;

by its subcomponents.

Now we analyse which the final *SEM* state after their execution and which is the final state of *JAVA* after the execution of the corresponding expansion.

Let us fix some things just once for the rest of the proof: if $se\ \mathcal{WB}\ ja$, then we assume, without explicitly repeating it, that if $ac\ \mathcal{P}\ tc$ (and $ac'\ \mathcal{P}\ tc'$ ...), and $oc\ \mathcal{O}\ joc$.

Now we introduce another lemma, used in the following

Lemma 7.2.12 If $ac\ \mathcal{P}\ tc$ and $ac\ \xrightarrow{\alpha}\ ac'$ and $Exp(\alpha)= \alpha_1 \ldots \alpha_n$, then $tc\ \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_n} tc'$ and $ac'\ \mathcal{P}\ tc'$.

Proof

The main point of the demonstration is that if a thread component tc is equivalent to an active component ac , then the code of tc is the translation of:

- the only transition capability described in `JTN` for the state described by of ac , plus the code that sets the new state of tc to the arrival state of the simulated transition;
- otherwise it is a sequence of exclusive if-statements, with non-overlapping conditions, that cover all the alternatives that are described in the `JTN` behaviour graph for the process corresponding to ac in *SEM*.

A more articulated proof can be done by cases on the structure of α.

i. $\alpha \equiv$ `call(`o_{id}`.m(`par_L`))`
ac has the form $con(e_1, \ldots ,e_n,0)$: the last parameter is 0, because a method call is not a choice of a multiple alternatives.

From **pCall1**, $con(e_1, \ldots, e_n, 0) \xrightarrow{\texttt{call(}O_{id}\texttt{.m(}parlist)} con'({e_1}', \ldots {e_m}')$;

The expansion of the method call consists of the method call, plus the assignments of the field values: $Exp(\alpha)$ =

- $\alpha_1 = o_{id}$`.m(`par_L'`)` with par_L' the transition of the parameter list into Java
- $\alpha_2 =$ `kind=` i;
- $\alpha_3 = f_1 =$ "translation of ${e_1}'$";
- $\alpha_i =$ ...
- $\alpha_m = f_m =$ "translation of ${e_m}'$";

but, if $ac\ \mathcal{P}\ tc$ then $tc = <t_{id}, c, m>$ with $c = o_{id}$.met($\mathsf{v}_1, \ldots \mathsf{v}_k$);$c_{ass}$ and c_{ass} is the code that translates the new state ($\mathsf{v}_i\ \mathcal{V}\ e_i{}'$for each i=1, ... ,n), thus the code that tc can execute is exactly the expansion of α.

After the execution of this code, the new thread component tc' is surely equivalent to ac' .

ii. $\alpha \equiv\ !ach\_id(e)$

From the rule **pWrite** we have that $ac \equiv con(e_1, \ldots, e_n, 0)$ and

$con(e_1, \ldots, e_n, 0) \xrightarrow{!ach\_id(e)} con'(e_1{}', \ldots, e_m{}', 0)$.

The expansion of α is: $\mathsf{Expr}(\alpha)$=

- α_1= ach_id.write(e)
- α_2= `kind`= "encoding of con'";
- α_3= f_1= "translation of $e_1{}'$";
- α_i= ...
- α_m= f_m= "translation of $e_m{}'$";

Again, we have that $tc\ \mathcal{P}\ ac$, the code that can be executed by tc is exactly the expansion of α.

Analogously we can prove the cases: $\alpha \equiv\ ?\ ach\_id(x)$, $!sch\_id(e)$, $!!sch\_id(e)$, $?sch\_id(x)$, $??sch\_id(x)$. $\square$**end lemma**

Now we can prove points 1-4 of lemma 7.2.11

1) External asynchronous communication: In se (axioms **CReadOut**[6] and **CWriteOut**) there exists an active component ac connected to the external channel $extid$ that performs the communication ending in the state ac' .

Such *SEM* transition capability is formalized by:

$$\textbf{CWriteOut}= \begin{cases} \mathtt{ID(pc)} = \mathsf{IdentComp}(pid_i,\ \rho)\ \wedge\ \mathtt{pc} \xrightarrow{!ln_i(\mathtt{term})} \mathtt{pc\'}\ \Rightarrow \\ (\mathtt{pc}, !ln(\mathtt{term}), \mathtt{pc\'}):\mathtt{pc}\ |\ \mathtt{PasC} \xrightarrow{\mathtt{Write}(extid, \mathtt{term})} \mathtt{pc\'}\ |\ \mathtt{PasC} \end{cases}$$

In the `JTN` behaviour graph of the process (corresponding to) ac , there is a sending operation on the channel ln_i ($!ln_i(\mathsf{v})$); in the architecture diagram, ac is connected with the external

[6] To increase the readability, in the following we sometimes use just the name to refer the rules in Chap. 4 and 5.1.

channel *extid*. The code that tc has to execute is, from the definition of $\mathcal{P}$, exactly the expansion of the external communication on an object joc that is a stream connecting with the external world and that is not modified.

Thus, ja has the form $ja = tc\ |joc\ |$S and the execution of $Exp(\alpha)$ can take place from ja.

Now we prove that $ja'\ \mathcal{WB}\ se'$.

From lemma 7.2.12 below we have that the execution of the code corresponding to the translation of the write operation leads tc to a state tc' s.t. $tc'\ \mathcal{P}\ ac'$. But tc and ac are the only components in ja and se that are modified, thus again $ja'\ \mathcal{WB}\ se'$.

Analogously we can prove that the property holds if the external label is a read operation.

2) Method Call In se (from axioms **CCall1**, **CCall2**) there exists an active component ac that modifies an object oc into oc', differing from oc'.

If $tc\ \mathcal{P}\ ac$, from the definition of $\mathcal{P}$ again, the code of tc corresponds to the Java implementation of the method call and (from lemma 7.2.12) it ends in a state equivalent to ac'.

Furthermore, from the considerations above about the fact that the Java code implementing a `JTN` method just involves one object (the called one) and from the rules of the Java semantics, if tc calls a method of the object component joc, the whole system from state ja moves to a state ja' that differs from ja only in the new state of the thread (tc') and of the object (joc'): $tc\ |joc\ |PC \xrightarrow{\tau} tc'\ |joc'\ |PC$

We can also prove that joc' is the object state after the assignments contained in the method body and, consequently, $joc'\ \mathcal{O}\ oc'$.

Thus, $ja'\ \mathcal{WB}\ se'$.

3) Internal asynchronous communication

In se (axioms **CWriteIn**) there exists an active component ac that performs the write operation over a connector oc ending in the state ac', while the new connector state is oc' (we forget, for simplicity, the distinction about local names and object identities).

Such *SEM* transition capability is formalized by:

$$\textbf{CWriteIn)} \begin{cases} \mathtt{ID}(oc) = ach\_id \wedge oc' = \mathtt{Write}(oc, \mathtt{v}) \wedge \\ \mathtt{ID}(ac) = \mathtt{Iid}_i \wedge ac \xrightarrow{!ln_i{}^I(\mathtt{v})} ac' \Rightarrow \\ (ac, !\,ln_i{}^I(\mathtt{v}), ac'):\ ac\ |oc\ |\mathtt{PasC} \xrightarrow{\tau} ac'\ |oc'\ |\mathtt{PasC} \end{cases}$$

if the process component ac performs a write operation of value v on a stream component oc (with identity ach_id), the system state passes to a state where oc is replaced by oc' obtained by writing v in oc .

CWriteIn) is the semantics of an asynchronous simple cooperation (called ach_id) where pid, corresponding to ac , acts as a sender. The Java translation function creates a pipelined stream joc and assigns it to a field of the thread tc .

But if ac can write on oc , from the definition of $\mathcal{P}$ and lemma 7.2.12, tc can perform the expansion of the write operation on joc , modify its final state and pass to state tc' that is equivalent to ac' ($tc'\ \mathcal{P}\ ac'$).

From rules **PWriteObject1** and **PWriteObject2**, when tc performs the write method call, the whole Java program performs $tc\ |joc\ |S \xrightarrow{\tau} tc'\ |joc'\ |S$ where joc' is the stream (or file) obtained by adding to joc the value of the object o_{id}. Again, it is easy to prove that $joc'\ \mathcal{O}\ oc'$.

Analogously we prove the property for broadcast or named asynchronous communications.

4) Synchronous Communication If the component ac_1 of se performs a synchronous communication, it can be suspended (**CSynchBlockedRead**) or it can resume a suspended thread (**CSynchRead**) and complete the communication.

In the first case, it writes its identity on the synchronous channel oc and passes to the suspended state. In the second case, it writes the message in oc , wakes up the communication partner ac_2 and both the processes complete the communication.

The thread tc_1 corresponding to ac_1 , from the definition of $\mathcal{P}$, can simulate the same communication over the object component joc corresponding to oc ($oc\ \mathcal{O}\ joc$).

In the first case, there is also a thread tc_2 corresponding to ac_2 that is suspended on joc . Thus ja performs some internal interactions corresponding to the execution of the write call of tc_1 over joc . tc_1 performs two lock operations, assigns the joc field containing the message and sends a notify signal to tc_2 that, concurrently, performs a transition and completes the communication. Thus ja performs some internal transitions ending in $ja' \equiv tc_1'\ |tc_2'\ |joc'\ |$SS.

Again, from lemma 7.2.12, we have that the new thread components of the $JAVA$ state corresponds to the process components in se' ($ac_1'\ \mathcal{P}\ ac_1'$ and $tc_2'\ \mathcal{P}\ ac_2'$), as well as it easy to prove that the object component joc' corresponds to oc' ($joc'\ \mathcal{O}$ ocP).

Analogously we prove the second case above, when the synchronous communication cannot be immediately completed.

□**end lemma**

Now we can consider the case in which *SEM* models the execution of several transitions of an alternative: the first transitions describe the failure of some communications, the last one the successful completion of a communication and the reaching of a stable state.

Lemma 7.2.13 If $ja\ \mathcal{WB}\ se$ and $se \xrightarrow{a_1} se_1 \ldots\ldots \xrightarrow{a_n} se_n$, $a_1 \ldots a_n$ models the execution of an alternative, and se_n is a stable state, then $ja \xrightarrow{\mathsf{Expr}(a_1)} ja_1 \ldots \xrightarrow{\mathsf{Expr}(a_n)} ja_m$ ($ja \xrightarrow{\mathsf{Expr}(a)} ja_n$ is a short notation for $ja \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_n} ja_n$ if $\mathsf{Expr}(a) = \alpha_1, \ldots, \alpha_n$) and ja_m is a stable state *st* $ja_m\ \mathcal{WB}\ se_n$. **Proof** If *SEM* executes n transitions of an alternative, the first n-1 labels describe failures in a communication attempt (let us suppose, for the moment, to have read operations over asynchronous connectors). Thus $a_i \equiv (ac$, No(? $ach\_id_i()), ac'$), but from $ja\ \mathcal{WB}\ se$ and from $ac\ \mathcal{P}\ tc$ and from the definition of $\mathcal{P}$ the code of *tc* is a sequence of nested if-statements checking if on the generic channel ach_id_i there is an available message. If the component *ac* cannot complete the first n-1 communications, the same is for *tc* (their environment is equivalent). Thus *tc* is able to perform the expansions of the labels $a_1, \ldots, a_{n-1}$, each one corresponding to the lock of a channel, to check the message availability, to find that the condition is false and nothing else. a_n corresponds to the lock of a channel, to check the message availability, to find that the condition is true and setting the new state, then exit from the while loop and, consequently, from the `case` of the `switch`: the final state reached by *JAVA* is a stable state, equivalent to the one reached by *ac* . The other components of *SEM* and *JAVA* have not been modified, thus $se_n\ \mathcal{WB}\ ja_m$.

□**end lemma**

ii) **proof**

Given an observation $obs \equiv\ ja \overset{l}{\underset{J}{===>}} ja'$, if:

- the sequence of transitions in *obs* corresponds to the re-ordered sequence $\alpha_1 \ldots \alpha_n \beta_1 \ldots \beta_m \ldots \gamma_1 \ldots \gamma_k$;
- $\alpha_1 \ldots \alpha_n = Exp(\alpha)$, $\beta_1 \ldots \beta_m = Exp(\beta)$, $\ldots \gamma_1 \ldots \gamma_k = Exp(\gamma)$;

then we prove that the execution of $\alpha\beta \ldots \gamma$ is an observation $se \overset{\hat{l}}{\underset{S}{===>}} se'$ of *SEM* and $l\mathcal{L}\ \hat{l}$ and $se'\ \mathcal{WB}\ ja'$.

The proof comes observing that:

i. from Lemma 7.2.6, a *JAVA* observation exactly corresponds to the expansion of a finite number of *SEM* transitions; such sequence can be re-ordered, conditions above are satisfied;

ii. from the definition of the expansion function, we can easily prove that, if $se\ \mathcal{WB}\ \ ja$, then se can execute $\alpha\beta\ldots\gamma$ending in state se' s.t. $se'\ \mathcal{WB}\ ja'$

$\Box$**end lemma**

7.3 Expansion definition

Here we explain the expansion function in more details, taking as reference the table 7.5. For each kind of process transition in `JTN`, we report the related clause of CASL semantics and which is the label modeling its execution in the CASL specification. Then we report its translation into Java, as defined in Chap.5.1, and the labels of the model described by the Java semantics in Chap.6. These labels are the expansion of the *SEM* label.

Method Call

CASL semantics

- *Clause*: **pCall1** describes a *SEM* transition
- *Label*: `call`(o_{id}.`m`(...)) whose final state is the system states after the method execution.

Translation

$\mathrm{TJ}_{LabStateList}(st_1,(o_{id}.\mathtt{m}(\mathtt{e}_1;\ \ldots\ ;\ \mathtt{e}_n),st_2))[\rho] =$
$o_{id}.\mathtt{m}(\mathrm{TJ}_{Exp}(\mathtt{e}_1,\ T_1)[\rho],\ \ldots\ ,\ \mathrm{TJ}_{Exp}(\mathtt{e}_n,\ T_n)[\rho])$`;`
$\mathit{fieldass}_1$
...
$\mathit{fieldass}_n$

Java Semantics

- *Clause*: **TCall1**; the returned code is the method call statement plus the assignments of the new values of the thread fields.
- *Labels*: the execution of such code corresponds to several transitions: some labels for the evaluations of the arguments ($ll_{Eval(e)}$), one for the start of the method execution ($CallO(o_{id},\mathtt{m},\ \ldots)$), one or more for the execution of each statement of the method body (ll_{stm_i}), one describing the end of the method (τ_{return}, some transitions setting the new state value ($ll_{st'}$)

$\boxed{\textbf{Asynchronous write}}$ We describe the expansion of a write operation performed over an asynchronous channel. In the case of a read operation (if it is not part of a multiple choice), the expansion is defined in a similar way. A `JTN` asynchronous read operation can be the unique transition or just one of the alternatives from a state.
In the first case:

CASL semantics

- *Clause*: **pWrite** describes a simple asynchronous write operation on an asynchronous channel *ach_id*.
- *Label*: $!\,ach\_id(\mathtt{v})$;

Translation $\mathrm{TJ}_{LabStateList}(st_1,(!sch\_id(\mathtt{e}),st_2)) =$
sch_id`.write(`$\mathrm{TJ}_{Exp}(\mathtt{e},\ T)$
$fieldass_1$
...
$fieldass_n$

Java Semantics

- *Clause*: **TWriteOut** returns a call of the write method (we suppose that *ach_id* is also the identity of the Java object implementing the channel) plus a sequence of assignments ($ll_{pt_1} \ldots ll_{pt_1}$) setting the value of the new thread state.
- *Labels*: $ll_{Eval(e)}$ for the argument evaluation, $WriteO(ach\_id,\mathtt{v})$ for the method call and $ll_{pt_1} \ldots ll_{pt_n}$ for the assignments and the setting of the new state.

$\boxed{\textbf{Alternative of asynchronous simple read}}$ In this case, the read operation over an asynchronous channel can succeed or fail (in such case another alternative will be tried).

In the table we just report the case of the execution of an intermediate transition, thus not the last one, neither the `else` transition (but they are completely analogous to the generic case).

We first describe the expansion of a failed read operation, then the case of a successful one.

CASL semantics

- *Clauses*: the second axiom of **pMultAsynch1**and the first one of **pMultAsynch2**describe the failure of a read communication on a channel *ach_id* .
- *Labels*: the unsuccessful reading is labelled by `No?`$ach\_id()$

Translation A process in JTN fails in reading from an external component (that is it returns from a reading operation with no available message) only if it has some other reading alternatives (otherwise, if the reading operation is the only transition from a given state, it waits for a message). The code returned by the translation is: `bool exit = false;`

```
while (!exit) {
...
T_i x_i;
synchronized(ach_id_i){
if(ach_id_i.available() > 0 )
x_i= (T_i)ach_id_i.readObject();
TJ_State(st,st_i)[ρ];
exit = true;}
```

Java Semantics

- *Clause*: **TMultRead** returns the code implementing a multiple alternative from a process state (it is not a call of one of the I-O methods of the standard classes, that are modeled by just one transition, but it is the execution of the code implementing a test-and-read procedure).
- *Labels*: the code execution corresponds to the execution of the locking operation on the channel ($Lock(ach\_id)$), the checking of the data availability ($ll_{Available}$), the evaluation of the condition of an `if` statement (τ_{if}) that is false and of the unlocking operation ($Unlock(ach\_id)$)

In the other case, when the reading alternative succeed:

CASL semantics

- *Clauses*: the second axiom of **pMultAsynch1**and the first one of **pMultAsynch2**describe the failure of a read communication on a channel *ach_id* .
- *Labels*: the unsuccessful reading is labelled by `No?`$ach\_id()$

Translation The returned code is the one described above, that will be executed in a different way.

Java Semantics

- *Clause* : **TMultRead** as above;
- *Labels*: the labels of the code execution are the same of the previous case, except that the `if` condition evaluation (τ_{if}) returns true and the system performs a read operation ($ReadO(ach\_id_i, o_{id})$, the transitions setting the new state ($ll_{pt_1} \ldots ll_{pt_m}$, the assignment of the new value to exit (τ_{Exit}) and the release of the lock $Unlock(ach\_id_i)$)

Asynchronous named communication The observations are similar to the previous case, but the code returned by **TReadIn**, **TWriteIn** clauses not only calls the corresponding read or write method but also the code that manages the information about the sender or the receiver of the message (i.e. to create a new message or assigns the variable denoting the sender and the content of the message).

Synchronous communication

CASL semantics The case of a `JTN` transition describing a synchronous communication is a bit different. A synchronous read (write) communication transition of a process in `JTN` is modeled by rule **pSynchRead** (**pSynchWrite**) defining three different transitions for the active component in CASL:

1. the capability of completing the communication ($!sch\_id(msg)$);
2. the the capability of being suspended because the communication cannot be immediately completed ($!!sch\_id(msg)$);
3. the capability of being resumed by another component (`resumed`(sch_id)).

Translation the translation function applied to the `JTN` transition returns: $\mathrm{TJ}_{Label}(!sch\_id(\mathtt{e}))[\rho]$ sch_id`.write(`$\mathrm{TJ}_{Exp}(\mathtt{e},\ T)[\rho]$`);`

Java Semantics The possible executions of the Java code corresponding to the SEM transitions above are:

1. the thread evaluates the message ($ll_{Eval(msg)}$), then calls the reading method of the connector object ($CallO(sch\_id,$`read`$,\dots)$), performs the locking ($Lock(sch\_id)$), checks if there is a thread waiting to complete the communication (ll_{Check}), modifies the value of the *ready* variable ($ll_{SetReady}$), notify the waiting thread (ll_{Notify}), release the lock ($Unlock(sch\_id)$) and then returns from the method call with the read message (τ_{return}) and sets the new state ($ll_{st'}$).
2. the thread evaluates the message ($ll_{Eval(msg)}$), then calls the reading method of the connector object ($CallO(sch\_id,$`read`$,\dots)$), performs the locking ($Lock(sch\_id)$), checks that there is no thread waiting to complete the communication (ll_{Check}), modifies the value of the *ready* variable ($ll_{SetReady}$) and becomes waiting (ll_{Wait}).
3. the thread, that has been notified by another thread, release the lock ($Unlock(sch\_id)$) and sets the new state $ll_{st'}$.

Alternative of synchronous communications

CASL semantics The case of a `JTN` transition describing an alternative synchronous communication is modeled by the rules **pMultSynch1** and **pMultSynch2** defining two different transitions for the active component in CASL:

1. the capability of completing the communication ($!sch\_id(msg)$);
2. the the capability of failing the communication, without being blocked on it (No($!sch\_id()$)).

Translation the translation function applied to the `JTN` transition returns:

$\mathrm{TJ}_{LabStateList}(st,(!\ sch\_id_1(\mathtt{e}_1),st_1);\ \ldots\ ;(!sch\_id_n(\mathtt{e}_n),st_n))[\rho])=$

$\mathtt{if(}sch\_id_1\mathtt{.writeAvailable()(}\mathrm{TJ}_{Exp}(\mathtt{e}_1,\ T_1)[\rho]\mathtt{))}$
$\quad\{\mathrm{TJ}_{State}(st,st_1)[\rho]\mathtt{;}\}$
`else`
$\ldots$
`else`
$\mathtt{if(}sch\_id_{n-1}\mathtt{.writeAvailable}_{n-1}\mathtt{(}\mathrm{TJ}_{Exp}(\mathtt{e}_{n-1},\ T_{n-1})[\rho]\mathtt{))}$
$\quad\{\mathrm{TJ}_{State}(st,st_{n-1})[\rho]\mathtt{;}\}$
$\mathtt{else}\{sch\_id_n\mathtt{.write(}\mathrm{TJ}_{Exp}(\mathtt{e}_n,\ T_n)[\rho]\mathtt{);}$
$\mathrm{TJ}_{State}(st,st_n)[\rho]\mathtt{;}\}$

Java Semantics The possible executions of the Java code corresponding to the two *SEM* transitions above are:

1. the thread calls the method of the connector object that checks if it is possible to complete the communication immediately ($CallO(sch\_id,$`writeAvailable`$,\ldots)$), performs the locking ($Lock(sch\_id)$), checks if there is a thread waiting to complete the communication (ll_{Check}), modifies the value of the *ready* variable ($ll_{SetReady}$), notifies the waiting thread (ll_{Notify}), evaluates the message ($ll_{Eval(e_i)}$), sets the new message value ($ll_{set(msg)}$) release the lock ($Unlock(sch\_id)$), then returns from the method call with a true value ($\tau_{return(true)}$) and sets the new state (ll_{st_i}).
2. the thread calls the method of the connector object that checks if it is possible to complete the communication immediately ($CallO(sch\_id,$`writeAvailable`$,\ldots)$), performs the locking ($Lock(sch\_id)$), checks if there is a thread waiting to complete the communication (ll_{Check}), modifies the value of the and then returns from the method call with a false value ($\tau_{return(false)}$) *ready* variable ($ll_{SetReady}$)

7.4 Translation correctness

To check that the translation is correct w.r.t. a `JTN` specification, we have to show that the initial state of the *glts* modeling the semantics of the `JTN` specification is weakly bisimilar to the initial state of the model of the Java program obtained by the translation function applied to the `JTN` specification.

The initial state of the *SEM glts* has been defined by the semantics of the architecture diagram and by the axioms that, in each object and process specirfications, define the intial values of the fields and of the constructor arguments.

The initial state of the *JAVA glts* is the one reached after the execution of the instance creations inside the `main` method, just before the calls of the `start` methods of the thread components.

It is easy to prove that the two states are in the $\mathcal{WB}$ relation.

Chapter 8

Conclusions and Future Work

In the following we briefly synthesize the work presented in this thesis and the main results we obtained. Then we compare our formalism with analogous ones developed by other authors, especially focusing on the ones that seem to be popular and widely accepted. Finally we delineate which could be the most promising directions of our future work as well as the necessary enrichments and extensions to the `JTN` formalism that we have only sketched in this dissertation.

The work presented in this thesis has provided a visual and formal notation (called `JTN`) for the specification of reactive/concurrent systems. Such notation has been thought to be used in the context of a more articulated system development process, to support the design and the translation phases. But it can also be used directly, as a formal, precise but also clear and easy tool to describe and reason over Java programs, using visual presentations of their architecture or their possible executions.

The main ideas underlying the development of such notation comes from previous work ([EG94, EA96, ML97]) on such topics. Our main effort has been to start from these basis and develop a formal, Java-targeted notation and give it the fashion of a completely visual presentation. Targeting an existing development and specification framework to an implementation language does not simply affect the specification language used at the design level, but also on the whole development process. The class of system for which the method is tailored should be re-considered too.

Our main effort has been to investigate these effects and to introduce a more Java-dependent design phase in the development process. In such phase it is required to perform some design choices that makes straightforward the implementation of the design specifications into Java.

In the first part we have described the visual notation, that takes into account the most well accepted concepts from OO world. A visual JTN specification consists of diagrams of different nature that supplies different views of the system. What is crucial is that the notation is completely formal, thus every fragment of a JTN specification has a clear semantics. There are precise relations among these diagrams and constraints on their forms and contents that ensure the consistency of the overall specification.

Then we have provided a system for statically checking the correctness of the specification: the rules of the system ensure that is possible to associate a type with a JTN specification only if every part has been correctly constructed. The subsequent formal semantics of a JTN specification and its translation into Java have been defined assuming the well-formed with regard to this static semantics.

We have stressed the absolute formality of the JTN notation: this required to precisely define the semantics of the JTN specification. We have used the CASL (see [oLD99]) formal language and its extension CASL-LTL ([E.A99]) to describe the algebraic counterpart of the visual JTN specifications.

One of the principal objectives of our work has been to provide an (automatic) translation of the specifications into Java. Such translation, defined by a function, has been completely described in Chap.5.1: it does not require any interaction with the JTN user and definitely produce a Java running program.

In order to give convincing arguments about the correctness of such implementation, we need a notion of equivalence between the produced Java programs and the JTN specifications. This required to compare the model of the formal counterpart of a JTN specifications with the one of a Java program. Thus we need a comparable and simple model for the Java programs obtained by the translation of the JTN specifications. This moves us to define a semantics for a very large subset of Java, including all the constructs and mechanism used to implement the JTN specifications.

Such activity also increased our familiarity with the features of the language and pointed out some aspects of the semantics that we can abstract from (see Chap.6).

8.0.0.0.4 Related works In the following we compare JTN with some of the specification formalisms that seem to have success in academic or in industrial world. In particular, we take into considerations the formalisms (both formal or semi-formal) that address the same kind of systems, thus are well-suited for concurrent rather then sequential systems and supply some form of visual notation.

Perhaps the most popular formalisms with the above features are *Petri Nets*, *SDL*, *statecharts* and *UML*: they have been developed for the treatment of distributed-concurrent systems, and supply a visual notations, sometimes combined with a textual counterpart. In some cases they also provide a precise, formal model and a definition of the semantic.

- *Statecharts*([D.H87]) generalize state-transition diagrams. They are used to specify the design of reactive systems, i.e. by describing the response of a system to input stimuli. While also addressing aspects like concurrency and real-time behavior, statecharts retain the visual and intuitive appeal inherent to ordinary state diagrams.

- *Live Sequence Charts (LSC)* ([DH98]) are an extension of Message Sequence Chart MSCs, whose name reflects that one the main extensions deals with specifying "liveness", i.e., things that must occur. In fact, *LSCs* allow the distinction between possible and necessary behavior both globally, on the level of an entire chart and locally, when specifying events, conditions and progress over time within a chart.

- *SDL* (Specification and Description Language) (see [SDL93])is a graphical, formal specification language for real-time systems. The visual description of the specification, also including some non-visual part, has an equivalent textual presentation. It supports formalization of synchronous communication only, by means of typed signals, thus it is commonly used within telecommunication systems.

- *Petri Nets* (see [Rei82]) is graphical appealing notation which is appropriate for modeling systems with concurrency. The language is a generalization of automata theory such that the concept of concurrently occurring events can be expressed. It is particularly well-suited to represent causal dependencies and independence relation, concurrency relation among events.

  The notation is completely formal, but it supports only the development of design specifications. It does not include any OO concepts, neither it supports the encoding of the specifications.

- *UML* ([Rat97a, Rat97b]) is probably the most widely used visual notation for an OO informal approach to the software development. It supplies a plethora of diagrams to describe different views of the systems. The notation can be used to describe the whole development process and also the phases of code testing and maintenance.

  Rational Rose [Ros] is a visual modeling tool supporting the use of UML and the encoding of the UML specifications into class templates for Java, C++ and other languages.

  Unfortunately the UML notation lacks a formal semantics and a method or precise guidelines that helps the user in the development process. Moreover, the automatic translation just returns the templates of the classes: the precise definition of the method bodies is left to the user, for example.

There is still something that has to be done in order to make JTN an acceptable, user-friendly and very powerful specification language. We can delineate several directions for future work related to the possible extensions and improvements of the JTN language.

We distinguish among the extensions purely related to the JTN language, the extensions we can consider on the formal framework in order to cover the other phases of the development process and the project of the Gaggia tool support for JTN.

8.1 JTN extensions

In this thesis we have focused the main features of JTN, leaving apart some others, that does not significantly increase the expressiveness of the language. We want to show, by convincing considerations, that JTN is a specification language with formal basis, but also with effective usability. So, in the previous chapters we avoid to consider all the particular extensions of the language that are not of paramount relevance but require more work (and space) to be treated extensively and are beyond the scope of the thesis. Some of such extensions have already been considered and sketched, for example, in the user manual (Chap.2); some other are probably not so trivial and require a little more investigation and work.

Class Diagram extensions

- We have to define, in a more precise way, which are the *datatypes* that are *predefined*, thus that do not appear explicitly in the class diagram. They are the classical basic datatypes: integer, natural, boolean, parametric lists, parametric sets, etc. The syntax, the semantics and the translation parts have to be slightly extended to consider them.

- In JTN, in order to take under control which are the effects of a method call on the overall system, an object instance cannot call methods of other objects and, thus, has no *sub-components* of kind object: an object can be accessed and modified only by methods, thus there is no reason to have sub-components if they cannot be accessed neither modified. This ensures that a call to a method just modifies the called object and allow to localize the modifications over shared objects.

  However, we are conscious that is quite natural and really useful to allow that objects can reference and use other objects. We could introduce a mechanism for the definition of inner-objects (that are objects whose visibility is limited to the object that created them, called the outer-object), thus a method can modify the sub-components but also use their services. This extension requires some modifications of the static semantics to check that the sub-components cannot be accessed from other classes but the outer one; the CASL semantics and the Java translation of a statically correct specification (thus, in which the access restrictions over these sub-objects are respected) should not reserve surprises.

- Another OO concept that we plain to introduce in `JTN` is the use of *class fields* and *methods*. A class field is a field that is common to all the instances of a given class, a class method is a method that is not called on a particular instance of the class. Their formalization is quite simple, the static checker can be easily extended and the translation in Java comes for free. Class methods and fields are already used in the implementation of datatype classes: a constant datatype value is implemented by a Java class field, whereas an operation that has no argument of the datatype in which it is defined, and, thus no particular Java object to which send the method call, is implemented by a Java class method.

- In the Chap.2 we have described how to represent object and process classes whose instances are *parameterized* over datatype values, defining local datatypes. The formal semantics, as well as the translation of these diagrams, have still to be considered; however in CASL the specification corresponding to the parametric class is equivalent to the specification of the other object and process classes, except that the parameter value is treated as an additional argument for each constructor, in the case of a process class, and an operation returning its value (analogously to the fields), in the case of an object class.

  The specification corresponding to the class of the parameter is the usual semantics of a datatype class (assuming the static correctness of the `JTN` specification ensure that the values of its sort are manipulated only by the elements of the parametric class).

  The Java translation of such classes is quite natural: they simply result in a class with an additional field implementing the parameter.

- Parameterization could also be used in a `JTN` extension for classes (perhaps only of datatype and object kind) whose definition is parameterized over another class. The formal semantics, as well as the Java translation, have still to be considered. However in CASL there is the concept of parametric specification, thus the formal semantics should be quite natural. For what concerns the translation, the idea is that if a parametric class (say T[P], where P is the name of the parameter) is instantiated by a class (say T[p], where p is the name of the class that instantiate the parameter) then T[p] is translated by textually replacing each occurrence of P with p in the code corresponding to the translation of the parametric class T[P].

- The inheritance relation over process classes is not equipped with very powerful mechanisms for modifying and enriching the inherited behaviour: it is only possible to define capabilities that are alternatives of the already defined one. It is reasonable to have more freedom in the re-definition of such behaviour, for example to eliminate some states or only some transitions, to change the conditions of an alternative, to replace parts of the behaviour graph.

- In `JTN` field hiding and method and operation overriding, as well as overloading, have been banished for the moment. This choice also makes less powerful the inheritance mechanism, especially for object and datatype classes, because it is not possible to use method overriding to (partially) re-define a class.

  Essentially, this limitation has been introduced to simplify the translation phase, where we have the problem of knowing the type of a `JTN` fragment to correctly translate it. A possible solution is to use an annotated syntax (following the idea of the annotated Java terms used in the Java semantics, see Chap.6), where every terms is enriched with the type information needed to resolve the field or method invocations.

Architecture Diagram

Some other extensions of `JTN` are not trivial and requires more careful investigation.

The architecture diagram of `JTN` for the moment does not support the creation and deletion of components: we assume to start with an initial configuration where all the components have already been created.

One of the topics we are currently working on is the `JTN` support for the specification and implementation of *distributed systems*. This extension requires to introduce in the architecture diagram the notation for distinguishing components residing over different nodes of a distributed system and for describing the communication channels connecting them. For example we could equip `JTN` with a notation for a socket-based communication mechanism among processes over remote nodes and a notation for method calls on remote objects.

After, we plan to introduce in `JTN` a mechanism for specifying *mobility* of processes and objects: precisely a new form of communication among process to exchange also process (a mechanism similar to the serialization); its formalization requires some form of higher-order specification language.

The translation into Java of mobility and distribution should do not present great surprises: Java is equipped with powerful mechanisms for the communication of processes residing on different nodes of a distributed architecture (by *URL connections and sockets*), for calling remote methods (by the *RMI* mechanism) as well as for sending messages containing objects and processes (using *serialization*).

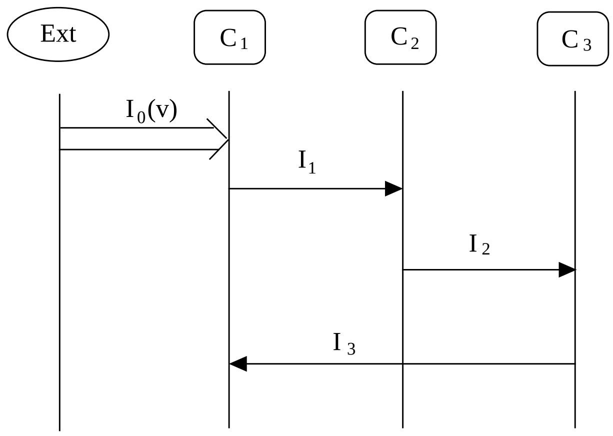


Figure 8.1: A generic sequence diagram

8.1.1 Sequence diagrams

The `JTN` sequence diagrams play the role of the execution diagrams of some methods and notations used in Software Engineering. They can be used with several different meanings: like descriptions of required system behaviours otherwise as presentations of possible use cases of the system.

In any case, their semantics can be rigorously given by some kind of temporal formula describing properties of the system behaviours, starting from the initial state of the sequence diagram. The precise form of the formula depends on essentially three factors:

1. the quantification over the behaviours;
2. the interpretation of the depicted interactions between internal components;
3. the interpretation of the interactions with the external environment;

The first factor determines if the property expressed by the execution diagram must be satisfied by either all or at least one system behaviours. Thus the form of the formula is something like (using the CASL-LTL notation for temporal formulae)

$in\_any\_case("initialstate", ...)$

or

$in\_one\_case("initialstate", ...)$

The second factor determines if the depicted interactions among the components are exactly the ones that can take place along the execution, or if there can also be other (non depicted) ones, involving the represented components.

Obviously, in both the cases we assume that interactions with some non depicted components can always take place, thus the formula does not completely describe the system behaviour.

In the first case, considering for example the simple sequence in fig. 8.1, the formula has the form

"eventually I_1 *and then no other transitions among* c_1, c_2, c_3 *until* I_2 *and then no other transitions among* c_1, c_2, c_3 *until* I_3*"*

In the second case, the formula has the simpler form

"eventually I_1 *and then eventually* I_2 *and then eventually* I_3*"*

For what concerns the third factor, it seems reasonable not to treat the external interactions in the same way of the internal ones. The reason is, essentially, that if we interpret the interactions with the external components as something that must happen, with the value content that we specify, we put perhaps too strong constraints on the capabilities of components that are not parts of the specified system. In the example in fig. 8.1 we can give two interpretations to the interaction $\mathrm{I}_0(\mathrm{v})$:

- something like:*"if the external environment sends a value* v *then eventually* I_1 *..."*
- or : *"the external environment must send a value* v *and eventually* I_1 *..."*

In the first case we just describe a reaction of the systems to a possible interaction with the external: this is the interpretation we adopted for the JTN sequence diagram, as formally described in Chap. 4.

Let us stress that using CASL-LTL we can give the precise semantics of each of the previous (and also other not considered here) interpretations for the sequence diagrams, thus JTN can be easily extended in various ways.

Furthermore, the way in which we depict the interactions describe total ordering, but we can also adopt different approaches: for example it is possible to interpret it as a partial ordering, and to equip JTN with a tool that calculate all the possible ordering suggested by the diagram (a similar approach has been adopted in the Live Sequence Charts formalism, in [DH98]).

8.2 JTN towards more abstract specifications

The work presented in this thesis has been focused on the problem of making more Java-targeted the visual formal design specification language described in previous works [EG94, EA96, ML97], in order to automatize the translation into Java. Because of the time limitations for the thesis work, we left aside the problem of integrating this Java-targeted design phase with the previous phases of the system development process, that is of investigating the effects of the Java-targeting over the "more abstract" design specifications.

The `JTN` language should be extended in order to be used at a more abstract level, where the `JTN` user is not forced to give the precise definitions of the method bodies or of the object internal states but it is possible just to give properties on them.

In the more abstract specifications we plan to maintain the use of O.O. concepts of class, instance and inheritance, but the body of the methods as well as the process behaviours are described by means of logic formulae expressing their abstract properties. In the case of the process classes, the formulae can express also liveness and other behavioural properties.

A possible alternative development directions for the Java-targeting of the requirement specification language is towards a language for specifying *components*, rather then classes and objects. However, this is just an hypothesis requiring much more investigations and work.

The detailed description of the process behaviour and the precise definitions of the methods are demanded to the subsequent design phases of the development process.

The concept of specification refinement as well as the language for expressing behavioural properties by means of first-order (also temporal) properties can be derived by the one in [EG94, EA96, ML97], but properly modified in order to take into account, at this level too, the features and limitation of the more concrete specifications language due to the Java-targeting of the overall method.

8.3 Towards the Gaggia environment

The final aim of our work is to provide a complete *software environment* supporting the use of the `JTN` notation, with tools and facilities supporting the specification as well as the implementation phases of the system development. This framework is `Gaggia` (from the name of the most famous coffee machine, that we suppose will help the user in preparing a cup of Java coffee).

`Gaggia` should consists of software tools and methodological guidelines to use them. The tools should cover every phase, from the diagram construction to the Java encoding, probably interacting among them. Thus it is sensible that the used programming language is the same for all the tools; in particular we plain to use Java as much as possible.

We are currently looking for a common format for the `JTN` specifications that can be used and exchanged among the tools.

In this thesis we have described the textual notation for the `JTN` specifications using a BNF grammar. Another possibility is to use XML, as it is a rich and flexible language, for giving the description of the `JTN` language by a DTD document. Furthermore, it is possible to give the Java internal representation of the DTD (see [W3C98]) that could be used by the graphical environment as well as by the static checker or by any other tool of `Gaggia` that needs to manipulate a common internal format to represent `JTN` specifications.

The phase of the construction of the visual `JTN` specification should be supported by a graphical environment in which the user can manage graphical predefined elements and fill and assemble them in order to construct the class, architecture and sequence diagrams.

In the hypothesis of an extension of the Java-targeting also to the previous phases of the development process, the graphical environment should be able to support also the visualization of the abstract design specifications.

The software tool should avoid to construct bad-formed diagrams, detecting most of the static errors and inconsistencies, such as, for example, loops in the relations among classes, incorrect cooperations and so on.

We could adopt some *visual grammar* (see []) that allow to obtain a grammar for the rigorous description of the visual presentation of the language: this avoid to transform the visual presentation into a textual one. This grammars are also equipped with tools for the visualization of the generated terms, that we could include and use in `Gaggia`. Thus the graphical interface should be integrated by the static checker. We plan to use Java in combination with technologies like Tcl-Tk, Microsoft COM or GEF [1] for the realization of the graphical part.

The *graphic interface* should also be provided with a *browser* that supports the navigation of the `JTN` diagrams. A browser is useful to examine the specification, switching from one diagram to another, or to entering into the specification of a sub-part.

For what concerns the *translation* phase, we have precisely defined it by means of a function, that does not require any user-interaction and can be automatized. It is easy to see that such function is not only computable but also of very low complexity. It has been defined following the structure of the `JTN` terms, it does not use recursion and the necessary context information are quite poor and can be easily implemented. We think that the overall

[1]a free Java library using the Swing Java extension, that has been developed at UCI (University of California, Irvine) for the construction of user graphical interfaces, see the WEB site: http://www.ics.uci.edu/pub/arch/gef

complexity of the translator realization is comparable to the one of a simple compiler.

The translation tool should take the `JTN` specification internal presentation produced by the graphical interface and apply the translation function, returning Java code.

Another necessary tool for `Gaggia` is a *rapid prototyper* for the `JTN` specifications. We aim to project it to be used also as a sort of *debugger* for the specifications. More precisely, we would like to realize a user interface for choosing some system components (also defining their initial states) and some cooperations from the `JTN` diagrams. Then, the prototyper should simulate a system executions starting from the fixed initial state, creating a sort of log-file where it records the interactions that have been occured between the chosen components and the external world. After the execution simulation, the prototyper could supply a representation of the information contained in the log-file, using the `JTN` representation of the cooperations and of the exchanged values. This information could be re-arranged in order to visually appear to the user with the structure of a sequence diagram.

This would of paramount importance if we consider the sequence diagrams as test-descriptions supplied with the system specification. The prototyper could be used to check the conformity of the implemented system with the properties suggested by the sequence diagrams.

Much has to be done with regard to `Gaggia`, but we are convinced that the theoretical problems has been already investigated and resolved (apart from the extensions described above, that we plan to consider in the future) it is meanly a question of finding human resources and apply existing technologies for the software realizations.

Appendix A

JTN Textual Presentation

In this appendix we give a grammar for a textual presentation of `JTN`, which will be used for defining its static and dynamic semantics and its translation into Java.

We use a BNF presentation style of the grammar, thus the symbol `::=` separates the right and the left parts of a production; the right parts of productions having the same left part are put together on the right part of the same production, separated by the symbol |. When needed, round parenteses are used to obtain an anambiguous grammar.

We use the notations NT^* (and NT^+) for the lists (non-empty lists) of elements generated by the nonterminal NT. They are a convenient abbreviations for

$\mathsf{NT}^* ::= \Lambda \mid \mathsf{NT}\ ;\ \mathsf{NT}^*$

$\mathsf{NT}^+ ::= \mathsf{NT} \mid \mathsf{NT}\ ;\ \mathsf{NT}^+$

The notation $\mathsf{NT}_{opt}(\mathsf{t}_{opt})$ says that the non-terminal (terminal) NT (t) is optional.

`JTN` Specification

JTN ::= (Class_dia, Arch_dia, Sequence_dia*)

Class Diagram

Class_dia ::= (DataType_dia*, Obj_dia*, Proc_dia+)

A.0.0.0.5 Datatype

$\text{DataType\_dia} ::= (\text{Type}_{\text{id}},\ \text{Uses},\ \text{Inherits},\ \text{VarDec}^*,\ \text{ConDec}^*,\ \text{ConDec}^*,\ \text{OpnDef}^*,\ \text{OpnDef}^*)$

$\text{Uses} ::= \texttt{use}\ \text{Type}_{\text{id}}{}^*$

$\text{Inherits} ::= \texttt{inherit}\ (\text{Type}_{\text{id}})_{opt}$

$\text{VarDec} ::= (\text{Var}_{\text{id}},\text{Type}_{\text{id}})$

$\text{ConDec} ::= \text{Con}_{\text{id}}(\text{Arg}^*)$

$\text{Arg} ::= (\text{Opn}_{\text{id}},\ \text{Type}_{\text{id}})\ |\ \text{Type}_{\text{id}}$

$\text{OpnDef} ::= (\text{Opn}_{\text{id}},\ \text{Type}_{\text{id}}{}^*,\ \text{Type}_{\text{id}},\ \text{OpnBody})$

$\text{OpnBody} ::= \text{CondRule}^+$

$\text{CondRule} ::= (\text{Opn}_{\text{id}}(\text{Pattern}^*),\ \text{CondExpr}^+)$

$\text{Pattern} ::= \text{Con}_{\text{id}}(\text{Pattern}^*)\ |\ \text{Var}_{\text{id}}$

$\text{CondExpr} ::= (\text{Expr},\ \text{Expr})$

$\text{Expr} ::= \text{Opn}_{\text{id}}(\text{Expr}^*)\ |\ \text{Con}_{\text{id}}(\text{Expr}^*)\ |\ \text{Var}_{\text{id}}\ |\ \text{Meth}_{\text{id}}(\text{Expr}^*)$

A.0.0.0.6 Object Class

$\text{Obj\_dia} ::= (\text{Type}_{\text{id}},\ \text{Relationships},\ \text{VarDec}^*,\ \text{FieldDec}^*,\ \text{MethodDef}^*,\ \text{MethodDef}^*)$

$\text{Relationships} ::= (\text{Clientships},\ \text{Uses},\ \text{Inherits})$

$\text{Clientships} ::= \texttt{client}\ \text{Type}_{\text{id}}{}^*$

$\text{FieldDec} ::= (\text{Field}_{\text{id}},\ \text{Type}_{\text{id}},\ \text{Expr}_{opt})$

$\text{MethodDef} ::= (\text{Meth}_{\text{id}},\ \text{Type}_{\text{id}}{}^*,\ \text{ReturnType},\ \text{MethodBody})$

$\text{ReturnType} ::= \text{Type}_{\text{id}}\ |\ \texttt{void}$

$\text{MethodBody} ::= \text{MethodCase}^+$

MethodCase ::= (Meth_{id}(Pattern*),CondStatement+)

CondStatement ::= (Expr, Statement+)

Statement ::= Assignment | MethodCall

Assignment ::= LeftExpr = Expr

LeftExpr ::= Field_{id} | `res` | Var_{id}

MethodCall ::= Meth_{id}(Expr*)

A.0.0.0.7 Process class

Proc_dia ::= (Type_{id}, Relationships, Interface_dia, Behaviour_dia)

Interface_dia ::= (ObjRef*, AsynCh*, SynCh*)

ObjRef ::= (Comp_{id}, Type_{id})

AsynCh ::= (ReturnType, IOSym, Chan_{id}, Type_{id})

SynCh ::= (IOSym, Chan_{id}, Type_{id}^{*})

IOSym ::= ! | ?

Behaviour_dia::= (ConDec+, VarDec*, Con_{id}(Expr*), BehaviourGraph)

BehaviourGraph::= StateCase+

StateCase ::= (TransitionCase*)

TransitionCase ::= (Pattern, TransitionAlt+)

TransitionAlt ::= (Expr, Label&State+)

Label&State::= (Label,State)

State ::= $\mathsf{Con_{id}}$(Expr*)

Label ::= $\mathsf{Comp_{id}}$.$\mathsf{Meth_{id}}$(Expr*) | $\mathsf{Var_{id}}$ <- $\mathsf{Comp_{id}}$.$\mathsf{Meth_{id}}$(Expr*) | Communication | `else`

Communication ::= (IOSym,$\mathsf{Chan_{id}}$(Expr)) | (Expr,IOSym,$\mathsf{Chan_{id}}$(Expr))

Architecture Diagram

Arch_dia ::= ($\mathsf{Component}^{+}$, Cooperations)

Cooperations ::= (ObjectCoop*, AsynhcCoop*, SynchCoop*)

Component ::= (Kind, $\mathsf{Comp_{id}}$, $\mathsf{Type_{id}}$)

Kind ::= `pr` | `ob`

ObjectCoop ::= ($\mathsf{Comp_{id}}$, $\mathsf{Comp_{id}}$, $\mathsf{Comp_{id}}$, Methods)

Methods ::= $\mathsf{Meth_{id}}^{+}$ | `ALL`

AsynhcCoop ::= ($\mathsf{Conn_{id}}$, $\mathsf{Type_{id}}$, ConnKind, $\mathsf{CoopComp}^{+}$, $\mathsf{CoopComp}^{+}$)

ConnKind ::= `b` | `s` | `n`

SynchCoop ::= ($\mathsf{Conn_{id}}$, $\mathsf{Type_{id}}$, $\mathsf{IntComponent}^{+}$, $\mathsf{IntComponent}^{+}$)

CoopComp ::= IntComponent | ExtComponent

IntComponent ::= ($\mathsf{Comp_{id}}$,$\mathsf{Chan_{id}}$)

ExtComponent ::= `EXT` | (`EXT`, $\mathsf{Ext_{id}}$)

Sequence Diagram

Sequence_dia ::= (VarDec*, $\mathtt{INIT}^{+}$, Coop^{+}, $\mathsf{Annotation}^{+}$)

`INIT` ::= (Actor, InitExpr)

InitExpr ::= Assignment* | Expr

Coop ::= (Actor, LabelSeq, Actor)

Actor ::= $\mathsf{Comp}_{\mathsf{id}}$ | $\mathsf{Conn}_{\mathsf{id}}$

LabelSeq ::= Expr = $\mathsf{Meth}_{\mathsf{id}}$(Expr*) | $\mathsf{Meth}_{\mathsf{id}}$(Expr*) | Communication

Communication ::= !$\mathsf{Chan}_{\mathsf{id}}$(Expr) | Expr !$\mathsf{Chan}_{\mathsf{id}}$(Expr) | ?$\mathsf{Chan}_{\mathsf{id}}$(Expr)

Annotation ::= Expr

Lexical Syntax

Here we define the lexical syntax of the various kinds of identifiers: $\mathsf{Type}_{\mathsf{id}}$, $\mathsf{Chan}_{\mathsf{id}}$, $\mathsf{Con}_{\mathsf{id}}$, $\mathsf{Ext}_{\mathsf{id}}$, $\mathsf{Opn}_{\mathsf{id}}$, $\mathsf{Var}_{\mathsf{id}}$, $\mathsf{Field}_{\mathsf{id}}$, $\mathsf{Meth}_{\mathsf{id}}$ and $\mathsf{Comp}_{\mathsf{id}}$. All such identifiers are words:

$\mathsf{Type}_{\mathsf{id}}$ ::= Identifier

$\mathsf{Chan}_{\mathsf{id}}$ ::= Identifier

...

$\mathsf{Comp}_{\mathsf{id}}$ ::= Identifier

Identifier ::= Word | Word_Word

Word ::= Letter SubWord | Letter

SubWord ::= LetterDigit | LetterDigit SubWord

LetterDigit ::= Letter | Digit

Digit ::= `0` | ... | `9`

Letter ::= `A` | `B` ...

An element of Identifier cannot be a Java or a Casl reserved keywords.

The Java keywords are: `abstract`, `boolean`, `break`, ..., `while` (see [J.G96]).

The Casl keywords are: `and`, `arch`, ..., `within` (see [oLD99]).

Appendix B

JTN Visual Presentation

In this appendix we present the visual presentation of the various `JTN` constructs following the grammar defined in appendix A.

`JTN` Specification

The visual presentation of a `JTN` specification

(*cldia*, *archdia*, *seqdias*)

consists of the visual presentations of *cldia*, *archdia*, and *seqdias*.

Class Diagram

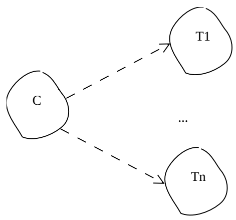
- The visual presentation of a class diagram

  (*datatypes*, *objdias*, *pdias*)

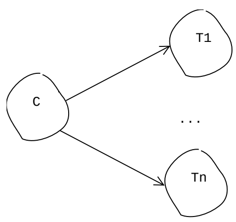
  is a graph whose nodes are the visual representations of the various classes in *datatypes*, in *objdias*, and in *pdias*, and whose arcs are the representations of the the class relationships, as expressed within such classes.

B.0.0.0.8 Relationships

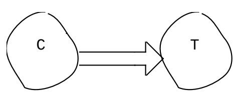
- `client` T_1; ... ; T_n appearing in an object or process class C is represented as follows:



- `use` T_1; ... ; T_n appearing in a (datatype, object or process) class C is represented as follows:



- `inherit` T appearing in a (datatype, object or process) class C is represented as follows:

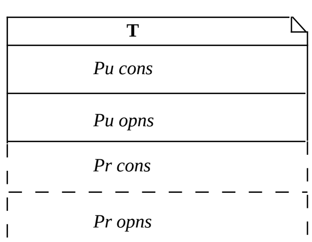


B.0.0.0.9 Datatype

- The visual presentation of a datatype

  T, *uses*, *inherits*, *vardecs*, *Pr_cons*, *Pu_cons*, *Pr_opndefs*, *Pu_opndefs*

  has form



  Notice that *uses* and *inherits* determine the arcs of the class graph, and so they do not appear in this diagram.

- The visual presentation of a constructor definition

  $con((ext_1\colon\ T_1)\ ;\ \ldots;(ext_n\colon\ T_n))$

  is the same as the above textual one.

- The visual presentation of an operation definition

  $(opn, T_1; \ldots; T_n, T_r, opnbody)$

  is

  $opn(T_1, \ldots, T_n){:}\,T_r$

  followed by the representation of *opnbody*.

- The visual presentation of an operation body

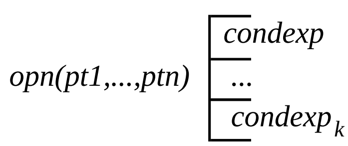
  $condrule_1; \ldots; condrule_n$

  consists of the representations of $condrule_1, \ldots, condrule_n$ one after the other.

- The visual presentation of a conditional rule

  $opn(pt_1; \ldots; pt_n), condexp_1; \ldots; condexp_k$

  is as follows.

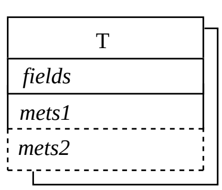
- The visual presentation of patterns and expressions is just the same as the textual one.

B.0.0.0.10 Object Class

- The visual presentation of an object class

  $(T,\ relations,\ vardecs,\ fields, mets_1,\ mets_2)$

  is as follows.

  Notice that *relations* determines the arcs of the class graph, and so it does not appear in this diagram.

- The visual presentation of a list of field declarations is just the same as the textual one.

- The visual presentation of a field declaration

  $(f, T, \mathtt{e})$

  is as follows.

  f: $T = \mathtt{e}$.

- The visual presentation of a list of method definitions is given by the representations of the various definitions, separated by dotted lines.

- The visual presentation of a method definition

  $(\mathtt{m}, T_1; \ldots; T_n,\ T_r, mbody)$

  consists of

  $\mathtt{m}(T_1, \ldots, T_n)$: T_r

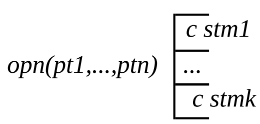
  followed by the representation of *mbody*.

- The visual presentation of a list of method cases is given by the representations of the various cases, one after the other.

- The visual presentation of a method case

  $\mathtt{m}(pt_1; \ldots; pt_n), cstm_1; \ldots; cstm_k)$

  is as follows.

- The visual presentation of a conditional statement

  $(bexp,\ stm)$

  is as follows

  $bexp$ `=>` stm

- The visual presentation of an assignment is the same as the textual one.

- The visual presentation of a method call

  $\mathtt{m}(\mathtt{e}_1; \ldots; \mathtt{e}_n)$

  is as follows

  $\mathtt{m}(\mathtt{e}_1, \ldots, \mathtt{e}_n)$

B.0.0.0.11 Process class

- The visual presentation of a process class

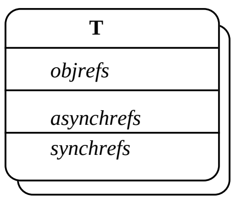
  (T, *relations*, *interface*, *pbody*)

  is given by the diagram determined by *interface*, which will be part of the class diagram, and by the other one determined by *pbody*, which will be given apart. Notice that *relations* determines the arcs of the class graph, and so it does not appear in this diagram.

- The visual presentation of an interface diagram

  (*obrefs*, *asynchs*, *synchs*)

  of a process class named T is as follows.

- The visual presentation of an object reference

  (o_{id}, T)

  is as follows

  o_{id} : T

- The visual presentation of an asynchronous channel reference

  (T,*io*,*ach_id*,T')

  is as follows

  T*ioach_id*(T')

- The visual presentation of a synchronous channel reference

  (*io*, *sch_id*, T')

  is as follows

  iosch_id(T')

- The visual presentation of a behaviour diagram

  (*cons*, *vardecs*, *con*(*exps*), *bgraph*)

  is given by the *bgraph* together with *con*(*exps*).

- The visual presentation of a behaviour graph

  $state\_case_1; \ldots; state\_case_n$

  associated with *exps* is given by combining the representations of each state case, $con(exps)$ determines the initial state that will be marked by an ongoing arrow.

- The visual presentation of a state case

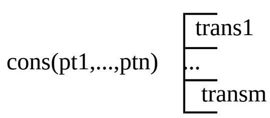
  $transcase_1; \ldots; transcase_n$

  is given by putting inside an oval the representations of each transition case, fusing together possible repeated states.

- The visual presentation of a transition case

  $(con(pt_1; \ldots; pt_n),\ trans_1; \ldots; trans_m)$

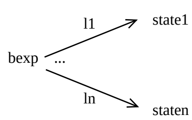
  is as follows.



- The visual presentation of a transition alternative

  $(bexp,\ (l_1, st_1); \ldots; (l_n, st_n))$

  is as follows.



  every $st_1, \ldots, st_n$ is put in the oval that collects all the terms representing the state.

- The visual presentation of the labels is the same as the textual one.

- The visual presentation of a synchronous I/O expression

  $(io, sch\_id(\mathsf{e}))$

  is as follows

  $iosch\_id(\mathsf{e})$

- The visual presentation of an asynchronous I/O expression

  $(\mathsf{e}_1, io, ach\_id(\mathsf{e}_2))$ is as follows

  $\mathsf{e}_1 ioach\_id(\mathsf{e}_2)$

Architecture Diagram

- The visual presentation of an architecture diagram

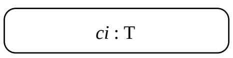
  (*comps*, *coops*)

  is a graph whose nodes are determined by *comps*, and whose arcs are determined by *coops*.

- The visual presentation of a process component

  ($\mathtt{pr}$, ci, T)

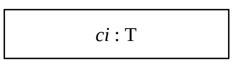
  is as follows.

- The visual presentation of an object component

  ($\mathtt{ob}$, ci, T)

  is as follows.

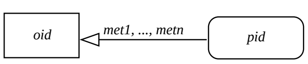
  

- The visual presentation of a list of cooperations is given by the representations of the single ones.

- The visual presentation of an object cooperation

  (o_{id}, pid,ln,$\mathtt{m}_1$; $\ldots$;$\mathtt{m}_n$)

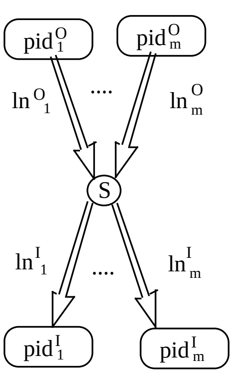
  is as follows.

- The visual presentation of a simple asynchronous cooperation

  ($conn$,T,$\mathtt{s}$,($pid_1{}^I$,$ln_1{}^I$); $\ldots$;($pid_n{}^I$,$ln_n{}^I$),
  ($pid_1{}^O$,$ln_1{}^O$); $\ldots$;($pid_m{}^O$,$ln_m{}^O$))

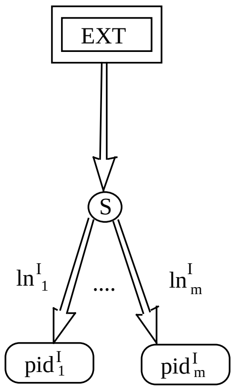
  is as follows.

- The visual presentation of a simple input asynchronous cooperation with the external

  $(conn,T,\mathtt{s},(pid_1{}^I,ln_1{}^I);\ \ldots;(pid_n{}^I,ln_n{}^I),$
  $(\mathtt{EXT},\ \mathsf{Ext}_{\mathsf{id}}))$

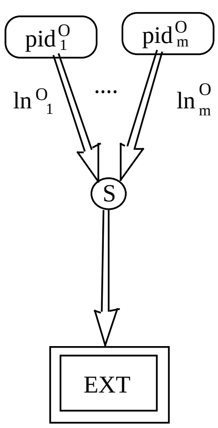
  is as follows.



- The visual presentation of a simple output asynchronous cooperation with the external

  $(conn,T,\mathtt{s},(\mathtt{EXT},\ \mathsf{Ext}_{\mathsf{id}}),$
  $(pid_1{}^O,ln_1{}^O);\ \ldots;(pid_m{}^O,ln_m{}^O))$

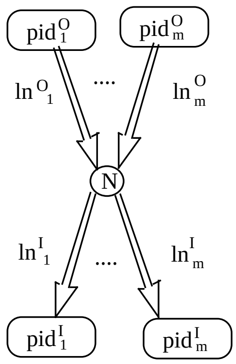
  is as follows.



- The visual presentation of a named asynchronous cooperation

  $(conn,T,\mathtt{n},(pid_1{}^I,ln_1{}^I);\ \ldots;(pid_n{}^I,ln_n{}^I),$
  $(pid_1{}^O,ln_1{}^O);\ \ldots;(pid_m{}^O,ln_m{}^O))$

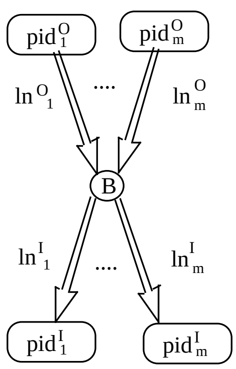
  is as follows.



- The visual presentation of a broadcast asynchronous cooperation

  $(conn, T, \mathtt{b}, (pid_1{}^I, ln_1{}^I); \ldots; (pid_n{}^I, ln_n{}^I),$
  $(pid_1{}^O, ln_1{}^O); \ldots; (pid_m{}^O, ln_m{}^O))$

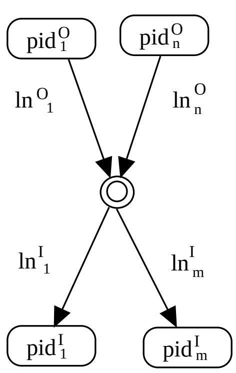
  is as follows.



- The visual presentation of a synchronous cooperation

  $sch\_id, T_m, ((pid_1{}^I, ln_1{}^I); \ldots; (pid_n{}^I, ln_n{}^I)),$
  $((pid_1{}^O, ln_1{}^O)); \ldots; (pid_m{}^O, ln_m{}^O))$

  is as follows.



Sequence Diagram

- The visual presentation of a sequence diagram

  (*vardecs*, *inits*, *interacts*, *anns*)

  is a sequence of vertical lines, one for each different component in *interacts*. On the top of each line there is the icon of the component, with the name and the type inside; the icon of a connector is a circle with the connector name inside. Under each icon, there is the visual presentation of the initial state in *inits*.

  The visual presentations of the interactions in *interacts* are combined with the vertical lines.

  The visual presentation of the annotations in *anns* is put at the bottom of the presentation of the interactions.

- The visual presentation of a list of initial state expressions $(act_1, init_1)\ ; \ldots ;(act_n, init_n)$ is the sequence of the visual presentation of each initial state expression.

- The visual presentation of an initial state expressions

  $(act, con(\mathtt{e}_1; \ldots ;\mathtt{e}_n))$

  is given by putting the textual expression $con(\mathtt{e}_1; \ldots ;\mathtt{e}_n)$ under the icon of *act*.

- The visual presentation of an initial state expressions

  $(act, f_1 = \mathtt{e}_1; \ldots ;f_n = \mathtt{e}_n)$

  is given by putting the $f_1 = \mathtt{e}_1, \ldots , f_n = \mathtt{e}_n$

  under the icon of *act*.

- The visual presentation of a list of interactions

  $(inter_1| \ldots |inter_m)$

  is the visual presentation of each interaction, one above the other, following the order in the list.

- The visual presentation of a method call interaction

  $(pid,\ \mathtt{e}= \mathtt{m}(\mathtt{e}_1; \ldots ;\mathtt{e}_n),\ o_{id})$

  is the arrow

  $\xrightarrow{\mathtt{e}=\mathtt{m}(\mathtt{e}_1;\ldots;\mathtt{e}_n)}$.

  starting from the vertical of the *pid*component and ending on the vertical line of o_{id}.

- The visual presentation of a method call interaction

  $(pid, \mathtt{m}(\mathtt{e}_1; \ldots; \mathtt{e}_n), o_{id})$

  is the arrow

  $\xrightarrow{\mathtt{m}(\mathtt{e}_1;\ldots;\mathtt{e}_n)}$.

  starting from the vertical of the *pid*component and ending on the vertical line of o_{id}.

- The visual presentation of an asynchronous interaction

  $(act_1, !ach\_id(\mathtt{e}), conn)$

  is the arrow

  $\xrightarrow{!ach\_id(\mathtt{e})}$

  starting from the vertical line of the act_1 component and ending on the vertical line of the connector *conn*.

- The visual presentation of an asynchronous interaction

  $(act_1, ?ach\_id(\mathtt{e}_2), conn)$

  is the arrow

  $\xrightarrow{?ach\_id(\mathtt{e})}$

  starting from the vertical line of the *conn* component and ending on the vertical line of act_1.

- The visual presentation of a synchronous interaction

  $(act_1, !sch\_id(\mathtt{e}), act_2)$

  is the arrow

  $\xrightarrow{!sch\_id(\mathtt{e})}$

  starting from the vertical line of the act_1 component and ending on the vertical line of the component act_2.

- The visual presentation of an annotation expression is equivalent to the textual one.

Appendix C

The calculator example

In this section we report a small example of `JTN` specification.

We specify the design of a Java program simulating a pocket calculator that computes and interacts with a keyboard, a display and a printer; think, for example, of a small application simulating a calculator on the desktop of a computer. The functionalities of the calculator are quite obvious: it can receive, by the physical keyboard, numbers and simple commands for performing operations (addition and multiplication) and for printing the display content. We have chose to process only one operation at a time: the user can introduce a number, an operation symbol, another number and then the equal symbol, to calculate the result of the operation application.

C.1 The Class diagram

The calculator has four active components: the keyboard driver reading keys from the keyboard, the computing unit performing the computations, the display driver echoing inputs and results to the display, and the print driver printing the display content. All such processes use an object, which records the content of the display, and three different types of data: digits, commands and lists of digits, enriched by the mathematical operations.

The keyboard driver receives the keys by an asynchronous channel from the keyboard (the external world); the display driver and the print driver send their outputs to, respectively, the display and the printer by two other asynchronous channels. The communications and the synchronizations among the active components are realized by some synchronous channels.

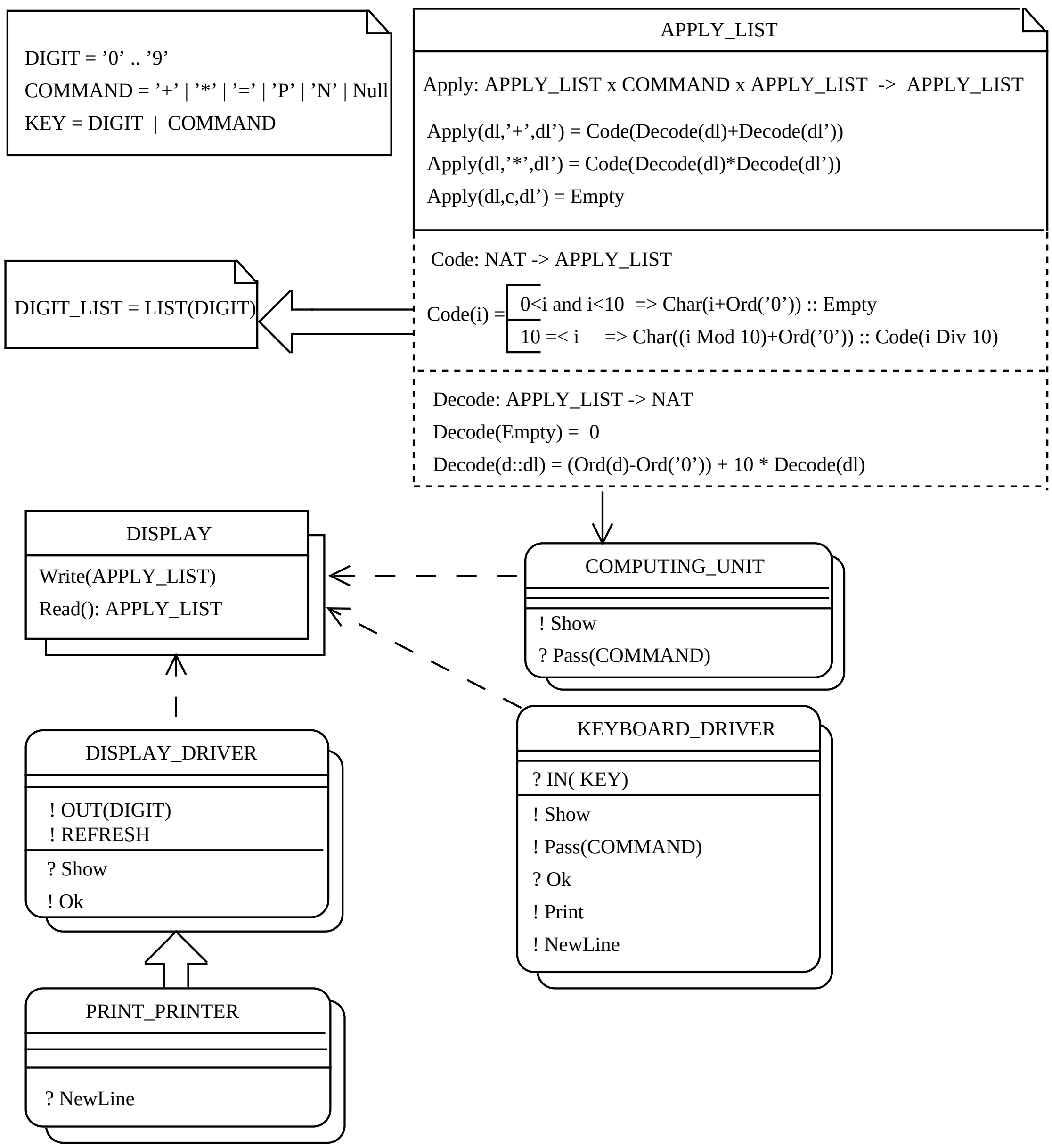


Figure C.1: The Calculator class diagram

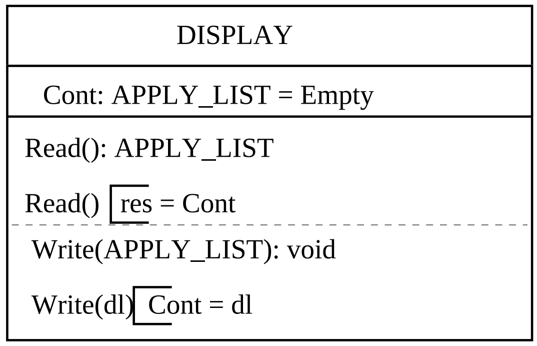


Figure C.2: The DISPLAY_DRIVER behaviour diagram

C.1.1 The datatype classes

A group icon contains the specifications of the `DIGIT`, the `COMMAND` and the `KEY` types.

The datatype `APPLY_LIST` extends the datatype `DIGIT_LIST` by adding a public operation `Apply` for calculating the result of addition and multiplication over two numbers represented by digit lists. Such operation requires two auxiliary (private) operations `Code` and `Decode` to manage the representation of the numbers.

C.1.2 The object class

DISPLAY

The `DISPLAY` object supplies two public methods that are used by the processes to modify the content of its unique field, of type `APPLY_LIST`.

C.1.3 The process classes

KEYBOARD_DRIVER

The `KEYBOARD_DRIVER` in the initial state can read a key (i.e., a digit or a command) from the channel IN connected with the external. Then it checks the value of the key. If it is:

- a newline command, it sends a signal to the `PRINTER_DRIVER` on the `Newline` channel and restores the initial state again;

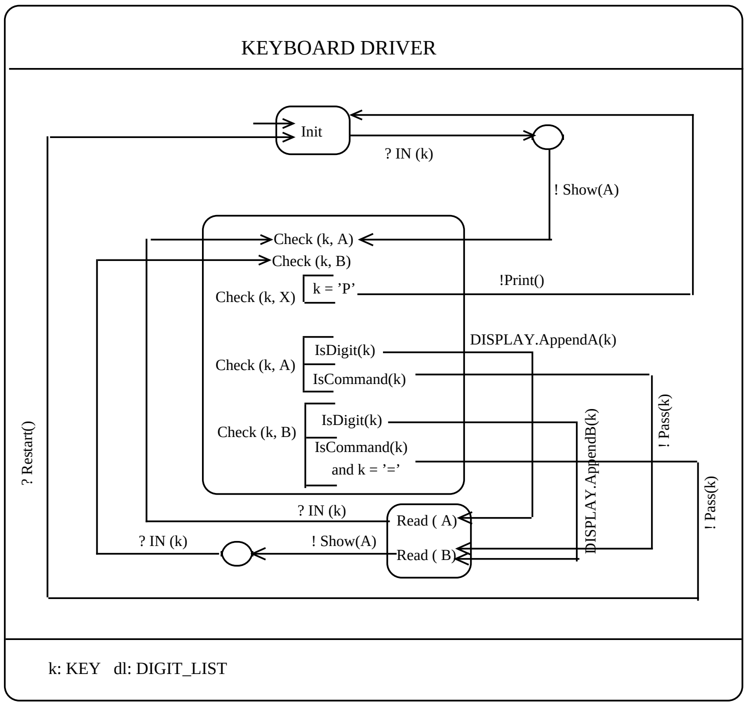


Figure C.3: The KEYBOARD behaviour diagram

- a print command, it first sends a signal on the `Newline` then on the `Print` channels of the `PRINTER_DRIVER` and then returns to the `initial` state;

- a digit, it comes to the `Taking` state; from now on, depending on the value of the key received on `IN`, it just continues the construction of the number, by appending to a local variable each digit it receives, or it sends a signal to the `Ok` channel of `DISPLAY_DRIVER` or not, then asks the `DISPLAY` object to write the new number and return to the checking state;

From the checking state it can send the command on the `Pass` channel connected to the `COMPUTING_UNIT` and returns to the `initial` state.

COMPUTING_UNIT

The `COMPUTING_UNIT` receives the command from the `KEYBOARD_DRIVER` by means of the `Pass` channel, then reads the first number from the `DISPLAY`, asks the `DISPLAY_DRIVER` to display it (by a synchronization on the `Show` channel) and waits for another command.

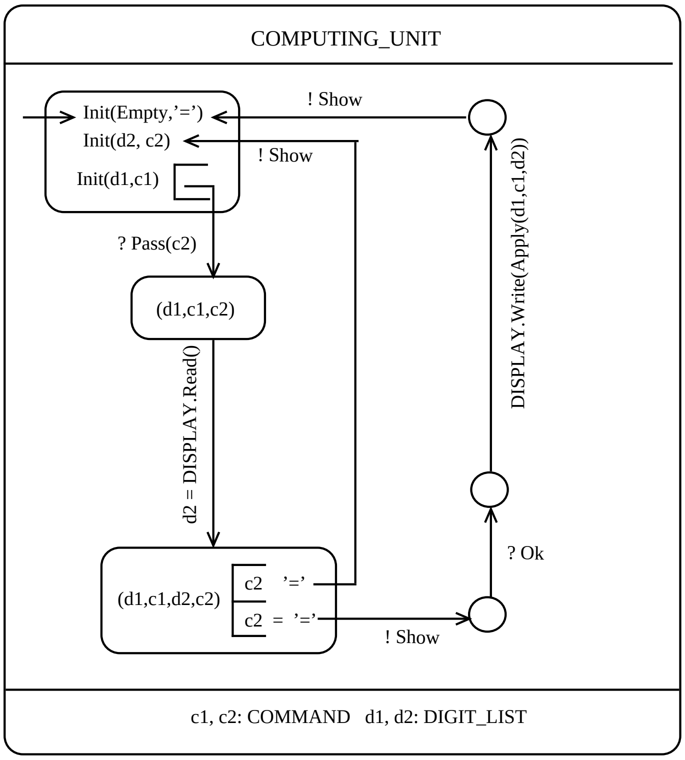


Figure C.4: The COMPUTING_UNIT behaviour diagram

When it receives the new command, reads from the DISPLAY_DRIVER the second number, ask again the DISPLAY_DRIVER to show it, writes on the DISPLAY the result of the computation and asks the DISPLAY_DRIVER to write the result out (again, by a synchronization on Show).

DISPLAY_DRIVER

The DISPLAY_DRIVER sends the content of the DISPLAY_DRIVER buffer on the external output every time it is changed, that is after the KEYBOARD_DRIVER has completely stored a new number or after the COMPUTING_UNIThas calculated the result of an operation.

In the initial state it waits for a synchronization on the Show channel from the COMPUTING_UNIT. Then, in sequence, performs the following activities:

- reads a number on the DISPLAY;
- notifies the KEYBOARD_DRIVER or the COMPUTING_UNIT that the DISPLAY can be mod-

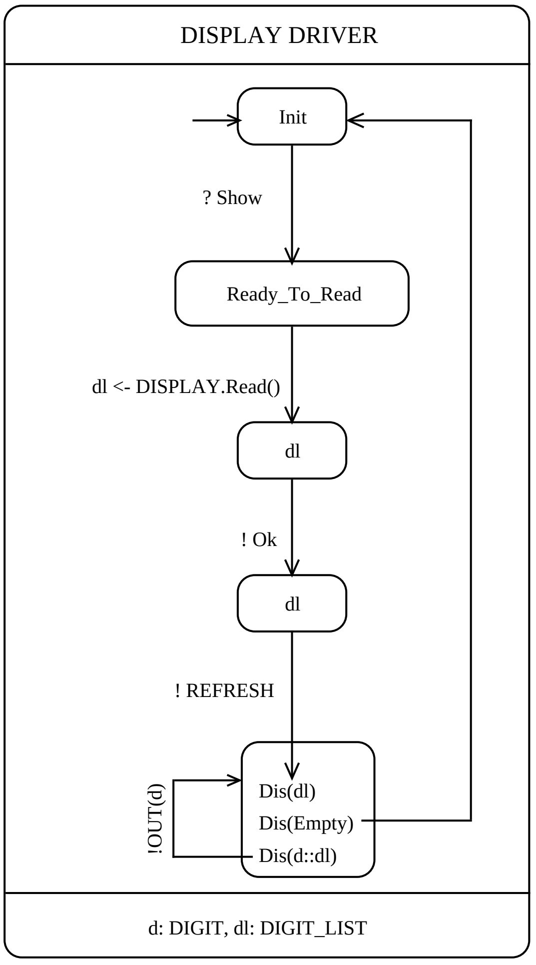


Figure C.5: The DISPLAY_DRIVER behaviour diagram

ified again, by sending a signal on the `Ok` channel;

- sends a message on `REFRESH` ;
- sends the read number to `OUT`, one digit at a time;
- comes back to the `initial` state;

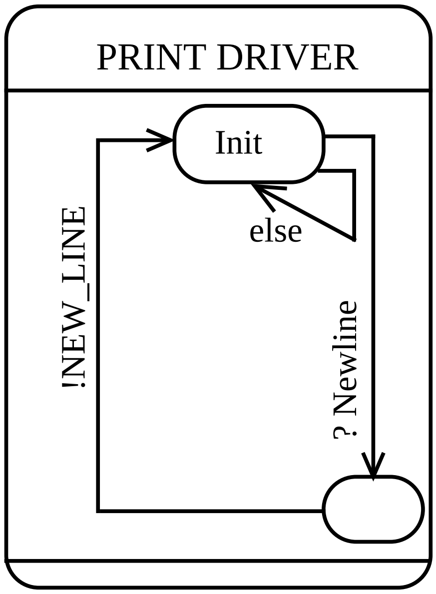


Figure C.6: The PRINTER behaviour diagram

PRINTER_DRIVER

The PRINTER_DRIVER has the same capabilities of the DISPLAY_DRIVER, but the Show channel is renamed into Print and it is connected to the KEYBOARD_DRIVER, thus it starts the reading of the number after a signal from the KEYBOARD_DRIVER. The OUT channel is renamed into OUT_P, that is connected to the external printer. Thus it has the behaviour of the DISPLAY_DRIVER but the output is redirect to another external component.

Finally, the PRINTER_DRIVER has the capability to receive a signal on the Newline channel and to send a message to the NEW_LINE asynchronous channel connected with the external.

C.2 The Architecture diagram

Below there is the diagram depicting the architecture of the Calculator system.

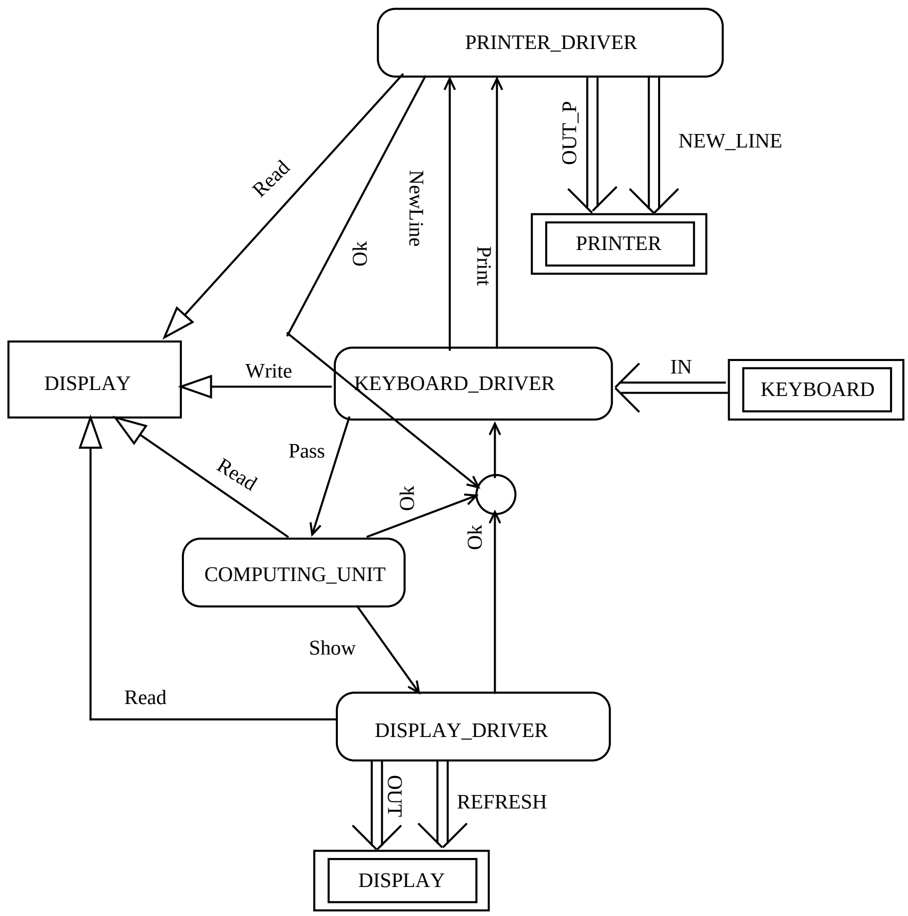


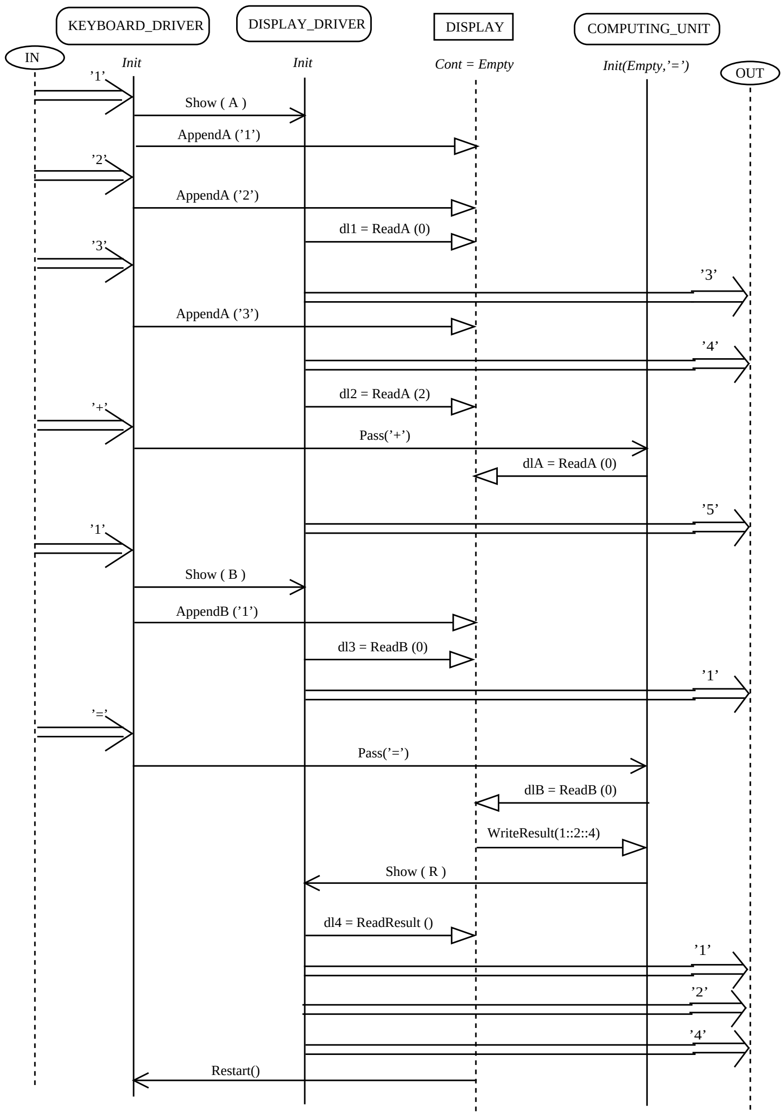
Figure C.7: The architecture diagram

C.3 The Sequence diagrams

The sequence diagram visualize the case in which the Calculator receives from the external the request to calculate "123 + 45".

The depicted components are the `KEYBOARD_DRIVER`, the `COMPUTING_UNIT`, the `DISPLAY_DRIVER`, the channel `IN` used by the `KEYBOARD_DRIVER` to receive the inputs, and the object component `DISPLAY`. The channel used by the `DISPLAY_DRIVER` to visualize the `DISPLAY` content is not represented.

However we can see all the interactions occurring among the components in order to compute the correct result that is read from the `DISPLAY` by the `DISPLAY_DRIVER`.



dl1 = 1::2 dl2 = 3:: Empty dl3 = 1::Empty dl4 = 1 :: 2:: 4   dlA = 1::2::3 dlB = 1:: Empty

dlA = 1;;2;;3

Figure C.8: A Sequence diagram

Appendix D

Technical definitions

In this chapter we recall some notions about *algebraic specifications* which appears in the literature. The chapter is organized by themes.

D.1 The CASL specification language

Here we briefly introduce the CASL ([oLD99]) specification language, mainly focusing on the sub-language that is used in Chap.4.

The acronym CASL stands for the Common Algebraic Specification Language, it is a an expressive algebraic language for specifying requirements and design for conventional software. It has been developed within CoFI, the Common Framework Initiative for algebraic specification and development, that groups together leading researchers in algebraic specifications, among which some people from our department at the University of Genova.

However, it is important to point out that CASL is just one possible language to describe the formal algebraic specifications that are the semantics of `JTN` specifications: different choices are possible, however we have chosen CASL because it is simple but, at the same time, expressive enough and it is aimed to become a widely accepted language. Furthermore we will refer to the CASL-LTL extension (see [E.A99]) of CASL for the specification of *dynamic systems*.

First we present CASL, then its extension CASL-LTL .

D.1.1 Basic Concepts

First, we briefly recall the main concepts underlying the formalization of specifications.

A basic specification framework may be characterized by:

- a class Sig of signatures Σ, each determining the set of symbols $|\Sigma|$ whose intended interpretation is to be specified;
- a class Mod(Σ) of models;
- a set Sen(Σ) of sentences (or axioms), for each signature Σ;
- a relation $\models$ of satisfaction, between models and sentences over the same signature;
- a proof system, for inferring sentences from sets of sentences.

A *basic specification* consists of a signature Σ together with a set of sentences from Sen(Σ).

The *(loose) semantics* of a basic specification is the class of those models in Mod(Σ) which satisfy all the specified sentences. A specification is said to be *consistent* when there are some models that satisfy all the sentences, and *inconsistent* when there are no such models. A sentence is a *consequence* of a basic specification if it is satisfied in all the models of the specification.

The proof system is required to be *sound*, i.e., sentences inferred from a specification are always consequences.

Sentences of basic specifications may include constraints that restrict the class of models, e.g., to reachable ones.

In the following we consider *many-sorted* basic specifications of the CASL specification framework.

D.1.1.0.12 Signature

A many-sorted signature is a 4-tupla $\Sigma = (S, TF, PF, PR)$ consisting of:

- a set S of sorts;
- sets $TF_{w,s}$, $PF_{w,s}$, of total function symbols, respectively partial function symbols, such that $TF_{w,s} \cap PF_{w,s} = \emptyset$, for each function profile (w,s) consisting of a sequence of argument sorts $w \in S^*$ and a result sort $s \in S$ (constants are treated as functions with no arguments);
- sets PR_w of predicate symbols, for each predicate profile consisting of a sequence of argument sorts $w \in S^*$.

Constants and functions are also referred to as operations, following the traditions of algebraic specification.

D.1.1.0.13 Models

For a many-sorted signature $\Sigma = (S, TF, PF, PR)$ a many-sorted model $M \in ModD\Sigma$ is a many-sorted first-order structure consisting of a many-sorted partial algebra:

- a non-empty carrier set s^M for each sort $s \in S$(let w^M denote the Cartesian product $s1^M \times \cdots \times sn^M$ when $w = s1...sn$);
- a partial function f^M from w^M to s^M for each function symbol $f \in TF_{w,s}$ or $f \in PF_{w,s}$, the function being required to be total in the former case;
- a predicate $p^M \subseteq w^M$ for each predicate symbol $p \in PR_w$.

D.1.1.0.14 Sentences

The many-sorted terms on a signature $\Sigma=(S, TF, PF, PR)$ and a set of sorted, non-overloaded variables X are built from:

- variables from X;
- applications of qualified function symbols in $TF \cup PF$ to argument terms of appropriate sorts.

For a many-sorted signature $\Sigma=(S, TF, PF, PR)$ the many-sorted sentences in Sen(Σ) are the usual closed many-sorted first-order logic formulae, built from atomic formulae using quantification (over sorted variables) and logical connectives. An inner quantification over a variable makes a hole in the scope of an outer quantification over the same variable, regardless of the sorts of the variables. Implication may be taken as primitive (in the presence of an always-false formula), the other connectives being regarded as derived.

The atomic formulae are:

- applications of qualified predicate symbols $p \in P$ to argument terms of appropriate sorts;
- assertions about the definedness of fully-qualified terms;
- existential and strong equations between fully-qualified terms of the same sort.

Definedness assertions may be derived from existential equations using conjunction, or regarded as applications of fixed predicates. Strong equations may be derived from existential equations, using implication and conjunction; existential equations may be derived from conjunctions of strong equations and definedness assertions, or regarded as applications of fixed predicates.

D.1.2 Basic Constructs

Below there is the syntax of the principal constructs (the ones used in Chap.4) of many-sorted basic specifications, and describes their intended interpretation. For the complete grammar defining the abstract syntax of the entire CASL specification language see [Mos97]

A well-formed many-sorted basic specification `BASIC-SPEC` in the CASL language is written simply as a sequence of `BASIC-ITEMS` constructs. Each `BASIC-ITEMS` construct determines part of a signature and/or some sentences (except for `VAR-ITEMS`, which merely declares some global variables).

A list of signature declarations and definitions `SIG-ITEMS` determines part of a signature and possibly some sentences. A `FREE-DATATYPE` construct determines part of a signature together with some sentences. A sort-generation construct `SORT-GEN` determines part of a signature, together with some sentences including a corresponding sort generation constraint. A list of variable declaration items `VAR-ITEMS` determines sorted variables that are implicitly universally quantified in the subsequent axioms of the enclosing basic specification; note that variable declarations do not contribute to the signature of the specification in which they occur. A `LOCAL-VAR-AXIOMS` construct restricts the scope of the variable declarations to the indicated list of axioms. (Variables may also be declared locally in individual axioms, by explicit quantification.) An `AXIOM-ITEMS` construct determines a set of sentences.

The grammar uses uppercase words for nonterminal symbols, allowing also hyphens. All other characters stand for themselves, with the following exceptions:

- ::= and | are generally used as meta-notation, as in BNF;
- a string of characters enclosed in double quotation marks '"..."' always stands for the enclosed characters themselves; 'N t...t N' indicates one or more repetitions of the nonterminal symbol N separated by the terminal symbol t (which is usually a comma or semicolon); 'N...N' is simply one or more repetitions of N (occasionally, N here is a sequence of terminal and nonterminal symbols, such as '[SPEC]');
- 'var/vars' indicates that the singular and plural forms may be used interchangeably, and similarly for other keywords; 'end/' indicates that the use of 'end' is optional, and similarly for semicolons: ';/'.

The following nonterminal symbols are for the lexical syntax: `WORDS`, `DOT-WORDS`, `NO-BRACKET-SIGNS`, `DIGIT`, `DIGITS`, `NUMBER`, `QUOTED-CHAR`, `URL`, and `PATH`.

Comments are generally displayed in the same font as ordinary informal text that might appear before and after a CASL specification (but note that this may be overruled by formatting instructions occurring in the comment text); each line of the displayed comment must always start with %%.

```
BASIC-SPEC      ::= BASIC-ITEMS...BASIC-ITEMS  |

BASIC-ITEMS     ::= SIG-ITEMS
                  | freetype/types DATATYPE-DECL ;...; DATATYPE-DECL ;/
                  | var/vars VAR-DECL ;...; VAR-DECL ;/
                  | var/vars VAR-DECL ;...; VAR-DECL
                            "." FORMULA "."..."." FORMULA ;/
                  | axiom/axioms FORMULA ;...; FORMULA ;/

SIG-ITEMS       ::= sort/sorts SORT-ITEM ;...; SORT-ITEM ;/
                  | op/ops OP-ITEM ;...; OP-ITEM ;/
                  | pred/preds PRED-ITEM ;...; PRED-ITEM  ;/
                  | type/types DATATYPE-DECL ;...; DATATYPE-DECL ;/

SORT-ITEM       ::= SORT ,..., SORT

OP-ITEM         ::= OP-NAME ,..., OP-NAME : OP-TYPE
                  | OP-NAME ,..., OP-NAME : OP-TYPE , OP-ATTR ,..., OP-ATTR
                  | OP-NAME OP-HEAD = TERM

OP-TYPE         ::= SOME-SORTS  -> SORT  |  SORT
                  | SOME-SORTS  ->? SORT  |  ? SORT

SOME-SORTS      ::= SORT *...* SORT

OP-HEAD         ::= ( ARG-DECL ;...; ARG-DECL ) : SORT   |  :  SORT
                  | ( ARG-DECL ;...; ARG-DECL ) :? SORT  |  :? SORT
ARG-DECL        ::= VAR ,..., VAR : SORT

PRED-ITEM       ::= PRED-NAME ,..., PRED-NAME : PRED-TYPE
                  | PRED-NAME PRED-HEAD <=> FORMULA
                  | PRED-NAME <=> FORMULA

PRED-TYPE       ::= SOME-SORTS  |  ( )

PRED-HEAD       ::= ( ARG-DECL ;...; ARG-DECL )
```

```
DATATYPE-DECL   ::= SORT "::=" ALTERNATIVE "|"..."|" ALTERNATIVE
ALTERNATIVE     ::= OP-NAME  ( COMPONENT ;...; COMPONENT )
                  | OP-NAME  ( COMPONENT ;...; COMPONENT )?
                  | OP-NAME
                  | sort/sorts SORT ,..., SORT
COMPONENT       ::= OP-NAME ,..., OP-NAME :  SORT
                  | OP-NAME ,..., OP-NAME :? SORT
                  | SORT

VAR-DECL        ::= VAR ,..., VAR : SORT

FORMULA         ::= QUANTIFIER  VAR-DECL ;...; VAR-DECL "." FORMULA
                  | FORMULA / FORMULA /../ FORMULA
                  | FORMULA FORMULA... FORMULA
                  | FORMULA => FORMULA
                  | FORMULA if FORMULA
                  | FORMULA <=> FORMULA
                  | not FORMULA
                  | true | false
                  | def TERM

                  | TERM =e= TERM
                  | TERM = TERM
                  | TERM in SORT
                  | ( FORMULA )
                  | MIXFIX...MIXFIX

QUANTIFIER      ::= forall  |  exists  |  exists!

TERMS           ::= TERM ,..., TERM

TERM            ::= MIXFIX...MIXFIX | LITERAL

MIXFIX          ::= NO-BRACKET-TOKEN | PLACE
                  | MIXFIX : SORT
                  | MIXFIX as SORT
                  | MIXFIX when FORMULA else MIXFIX
                  | ( TERMS )
                  | [ TERMS ] | [ ]
                  |  TERMS  |
```

```
SORT             ::= TOKEN-ID
OP-NAME          ::= ID
PRED-NAME        ::= ID
VAR              ::= SIMPLE-ID

SIMPLE-ID        ::= WORDS
ID               ::= TOKEN-ID | MIXFIX-ID
TOKEN-ID         ::= TOKEN
MIXFIX-ID        ::= TOKEN-ID PLACE-TOKEN-ID ... PLACE-TOKEN-ID
                   |          PLACE-TOKEN-ID ... PLACE-TOKEN-ID
PLACE-TOKEN-ID   ::= PLACE TOKEN-ID
                   | PLACE
PLACE            ::= __

TOKEN            ::= WORDS | DOT-WORDS | DIGIT | QUOTED-CHAR
                   | SIGNS
NO-BRACKET-TOKEN::= WORDS | DOT-WORDS | DIGIT | QUOTED-CHAR
                   | NO-BRACKET-SIGNS

SIGNS            ::= NO-BRACKET-SIGNS | BRACKET-SIGNS
                   | NO-BRACKET-SIGNS BRACKET-SIGNS
BRACKET-SIGNS    ::= BRACKET SIGNS
                   | BRACKET
BRACKET          ::= [ | ] |   |
```

A basic specification consists essentially of a signature Σ (declaring symbols) and a set of sentences (axioms or constraints) over Σ. The semantics of a well-formed basic specification is the specified signature Σ together with the class of all Σ-models that satisfy the specified sentences.

A structured specification is formed by combining specifications in various ways, starting from basic specifications. For instance, specifications may be united; a specification may be extended with further signature items and/or sentences; parts of a signature may be hidden; the signature may be translated to use different symbols (with corresponding translation of the sentences) by a signature morphism; and models may be restricted to initial models.

```
SPEC             ::= BASIC-SPEC
                   | SPEC RENAMING
                   | SPEC RESTRICTION
```

```
                  | SPEC and SPEC and...and SPEC
                  | SPEC then SPEC then...then SPEC
                  | free GROUP-SPEC
                  | GROUP-SPEC
GROUP-SPEC      ::=  SPEC
                  | SPEC-NAME
                  | SPEC-NAME [ FIT-ARG ]...[ FIT-ARG ]

RENAMING        ::= with SYMB-MAP-ITEMS ,..., SYMB-MAP-ITEMS

RESTRICTION     ::= hide SYMB-ITEMS ,..., SYMB-ITEMS
                  | reveal SYMB-MAP-ITEMS ,..., SYMB-MAP-ITEMS

SPEC-DEFN       ::= spec SPEC-NAME = SPEC end/
                  | spec SPEC-NAME SOME-GENERICS = SPEC end/
SOME-GENERICS   ::= SOME-PARAMS | SOME-PARAMS SOME-IMPORTED
SOME-PARAMS     ::= [ SPEC ]...[ SPEC ]

SYMB-ITEMS      ::= SYMB
                  | SOME-SYMB-KIND SYMB ,..., SYMB
SYMB-MAP-ITEMS  ::= SYMB-OR-MAP
                  | SOME-SYMB-KIND SYMB-OR-MAP ,..., SYMB-OR-MAP
SOME-SYMB-KIND  ::= sort/sorts | op/ops | pred/preds

SYMB            ::= ID | ID : TYPE
TYPE            ::= OP-TYPE | PRED-TYPE
SYMB-MAP        ::= SYMB "|->" SYMB
SYMB-OR-MAP     ::= SYMB | SYMB-MAP

SPEC-NAME       ::= SIMPLE-ID
TOKEN-ID        ::= ... | TOKEN [ ID ,..., ID ]
```

The semantics of a well-formed structured specification is of the same form as that of a basic specification: a signature Σ together with a class of Σ-models. Thus the structure of a specification is not reflected in its models: it is used only to present the specification in a modular style.

Specifications may be named, so that the reuse of a specification may be replaced by a reference to it through its name. A named specification may declare some parameters, the union of which is extended by a body; it is then called generic. A reference to a generic specification should instantiate it by providing, for each parameter, an argument specification together with a fitting morphism from the parameter to the argument specification.

D.2 Casl-Ltl

The basic idea underlying CASL-LTL is to model a dynamic system by using a labeled transition system.

D.2.1 Labeled transition systems

A *dynamic system* is a many-sorted first-order structure over a signature having two sorts, *state* and *label* whose elements corresponds to the states and the interaction descriptions of the system, and with a predicate $\rightarrow \subseteq state \times label \times state$ whose elements represent the transitions of the system.

Def. D.2.1 A *labeled transition system* (shortly *lts*) is a triple

$$(State, Label, \rightarrow)$$

where *State* is the set of the *states*, *Label* is the set of the *labels* of the system and $\rightarrow \subseteq State \times Label \times State$ is the *transition relation* of the system. A triple $(st, l, st') \in \rightarrow$ is called a *transition* and is usually written $st \xrightarrow{l} st'$.

A dynamic system D can be modeled by an $lts = (State, Label, \rightarrow)$ and an initial state $st_0 \in State$. The elements in *State* are the intermediate states in the life of D that can be reached from st_0, while the transition relation describes the capabilities of D to pass from an intermediate state to another.

The elements in *Label* provides the information about the interactions performed by the system during a transition, that is about the external conditions that make such capability effective and on the transformation induced on the external by the execution of the transition.

A transition $st \xrightarrow{l} st'$ has the following meaning: the system D, from the state described by st has the *capability* to pass into the state st' by performing the interaction with the external world described by l.

We can associate a *transition tree* with an *lts* and a starting state. The root is st_0 and for each $st \xrightarrow{l} st'$ there is an edge connecting the nodes decorated by st and st_0 and decorated by l.

A concurrent dynamic system, thus a system with dynamic subcomponents, can be modeled by an *lts* obtained by composing the *ltss* of the components.

D.2.2 Generalized Labeled Transition Logic

A *Generalized Labeled Transition System* (shortly *glts*) is a generalization of an *lts*, useful for the modelization of concurrent dynamic systems, where in every transition is enlighten the role of the subcomponents. It can be obtained from an *lts* by modifying the transition predicate and enriching it with some kind of additional information.

Def. D.2.2 A *generalized labelled transition system* (shortly *glts*) is a 4-tupla

$$(State, Label, Info, \rightarrow)$$

where *State* is the set of the *states*, *Label* is the set of the *labels* of the system, *Info* is the set of the additional *information*; $\rightarrow \subseteq State \times Label \times State \times Info$ is the *transition relation* of the system. A 4-tupla $(\mathtt{in}, st, l, st') \in \rightarrow$ is called a *transition* and is usually written $\mathtt{in} : st \xrightarrow{l} st'$

The two sets of *State* and *Label* are the sets of intermediate states and labels, as for the *lts*; the *Info* set is a set of additional information about the role of the subcomponents involved in a system transition.

The effective nature of the elements in *Info* may vary and it is strictly related to the use of the model. In most of the cases, the information are the description of all (or some of) the transitions of the active subcomponents that take place during the global system transition.

The meaning of a $(i, s, l, s') \in \rightarrow$ is that the system D has the capability of performing the transition $st \xrightarrow{l} st'$ if some sub-components in st has the capability to perform the transitions described by i.

D.2.3 Casl-Ltl dynamic signatures

First-order structures corresponding to *ltss* can be formally described by CASL-LTL specifications.

The CASL-LTL extension of CASL offers a special construct to declare the two sorts of the states and labels of an *lts*, plus the special predicate describing its transitions.

Precisely, a CASL-LTL specification is a pair SP=$(D\Sigma, AX)$, where $D\Sigma=(\Sigma, DS)$ is a *dynamic signature*, and AX is a set of CASL-LTL formulae on Σ.

D.2.3.0.15 Dynamic signature A *dynamic many-sorted signature* is a 5-tupla $D\Sigma = (S, DS, TF, PF, P)$ where (S, TF, PF, P) is a many-sorted signature as described above, while DS is the set of *dynamic sorts* s.t. $DS \subseteq S$ and, for all $ds \in DS$, there exists $Label\_ds \in S - DS$ and $\_ \overset{-}{\longrightarrow} \_ \in P_{ds\ Label\_ds\ ds}$;

The models of a dynamic many-sorted signature is defined as for the basic CASL, but the presence of the dynamic sorts and of the transition predicate allow to equip a model with the set of the "paths" for a dynamic sort, describing all the possible behaviours of an element of sort ds.

Consider a dynamic signature $D\Sigma = (S, DS, TF, PF, P)$ and $M \in ModD\Sigma$. For each $ds \in DS$, the set of the *paths* on M of sort ds, denoted by $PATH(M, ds)$, is the set of all the sequences of transitions having the form either (1) or (2) below:

(1) $d_0\ l_0\ d_1\ l_1\ d_2\ l_2\ \ldots$ (infinite path)
(2) $d_0\ l_0\ d_1\ l_1\ d_2\ l_2\ \ldots\ d_n$ $n \geq 0$

where, for all $i \geq 0$, $(d_i, l_i, d_{i+1}) \in \rightarrow^M$, and there do not exist l, d' such that $(d_n, l, d') \in \rightarrow^M$.

Given $\sigma \in PATH(M,)$, $FirstS(\sigma)$ denotes the *first state* of σ; and $FirstL(\sigma)$ denotes the *first label* of σ, if exists, i.e., if σ is not just a state.

If $\sigma = d_0\ l_0\ d_1\ l_1\ d_2\ l_2\ \ldots$ and $h \geq 0$ s.t. there exists d_h, then $\sigma|_h$ denotes the path $d_h\ l_h\ d_{h+1}\ l_{h+1}\ d_{h+2}\ \ldots$, otherwise it is undefined.

D.3 LT temporal logic

Here we briefly report the temporal branching-time logic language used in the CASL specifications of Chap.4.

The *dynamic many-sorted sentences* in **TSen**$(D\Sigma)$ are the usual closed many-sorted first-order logic formulae, built from atomic formulae using quantification (over sorted variables) and logical connectives plus temporal formulae anchored to the elements of dynamic sorts.

The temporal formulae of CASL-LTL (presented in [E.A99]) express the properties of the elements of a dynamic sort ds (dynamic elements) in terms of their paths, i.e., of their possible behaviours.

Temporal formulae are obtained from state formulae by using temporal combinators (such as "henceforth", "eventually", "at the next instant", ...), together with classical propositional connectives and quantifiers.

It is a branching-time logic, instead of the simpler linear one, because it allows to express, in a natural way, properties about the choices available at a given moment of time.

But the logic we use in the axioms of the CASL specifications presented in Chap. 4 is a little bit different w.r.t. the one in [E.A99]: we need formulae enlightening the role of a component in a transition of the whole program, thus we will describe the extension $\mathbf{GTSen}(D\Sigma)$ of $\mathbf{TSen}(D\Sigma)$ where the atomic path-formulae express also properties on the activity of a sub-component of the system.

The model of the sentences in $\mathbf{TSen}(D\Sigma)$ are *ltss*, while the models of $\mathbf{GTSen}(D\Sigma)$ are *gltss*, as we express properties about paths in which also appears the information about the activities of the subcomponents involved in system transition.

In the following, we give the set of temporal formulae depending on a signature and the validity notion for these formulae on a structure.

Such formulae express properties that have to be satisfied along paths.

The dynamic sentences in $\mathbf{TSen}(D\Sigma)$ and the path formulae for the elements of the various dynamic sorts of $D\Sigma$ ($\mathbf{PSen}(D\Sigma, ds)_{ds \in DS}$) are defined as follows:

$\mathbf{TSen}(D\Sigma)$ contains all the atomic formulae of the basic CASL, all those built with the logic combinator of the basic CASL, and the following temporal formulae: for each $ds \in DS$

$in\_any\_case(t, \pi)$ and $in\_one\_case(t, \pi)$

with t term of sort ds and $\pi \in \mathbf{PSen}(D\Sigma, ds)$.

The path formulae for the elements of sort ds, $\mathbf{PSen}(D\Sigma, ds)$, are defined as follows

- $state\_cond(x \bullet F)$ $\qquad$ x variable of sort ds, $F \in \mathbf{TSen}(D\Sigma)$

  This formula holds on a path σ whenever F holds at the first state of σ.

- $label\_cond(x \bullet F)$ $\qquad$ x variable of sort $Label\_ds$, $F \in \mathbf{TSen}(D\Sigma)$

  This formula holds on a path σ whenever σ is not just a single state and F holds at the first label of σ.

- π_1 *until* π_2 $\qquad$ $\pi_1, \pi_2 \in \mathbf{PSen}(D\Sigma, ds)$

  This formula holds on a path σ whenever there exists a point in σ s.t. π_2 holds at such point and π_1 holds until before it.

- *eventually* π $\qquad$ $\pi \in \mathbf{PSen}(D\Sigma, ds)$

  This formula holds on a path σ whenever there exists a point in σ s.t. π holds at such point.

- *always* π $\qquad$ $\pi \in \mathbf{PSen}(D\Sigma, ds)$

  This formula holds on a path σ whenever π holds at any point in σ.

– $\neg\, \pi, \pi \;\Rightarrow\; \pi', \forall\, y \bullet \pi$

y variable of whatever sort, $\pi, \pi' \in \mathbf{PSen}(D\Sigma, ds)$

with the usual meaning

D.4 Satisfaction

Let M be a dynamic model over $D\Sigma$ and v a variable evaluation, then we define by multiple induction:

- the validity of a dynamic formula F in M w.r.t. v (written $M, v \models F$),
- the validity of a path formula $\pi \in \mathbf{PSen}(D\Sigma, ds)$ on a path $\sigma \in PATH(D\Sigma, ds)$ in M w.r.t. v (written $M, v, \sigma \models \pi$),

as follows:

– $M, v \models \mathit{in\_any\_case}(t, \pi)$ iff for each $\sigma \in PATH(M, ds)$ such that $\mathit{FirstS}(\sigma) = t^{M,v}$, $M, v, \sigma \models \pi$

– $M, v \models \mathit{in\_one\_case}(t, \pi)$ iff there exists $\sigma \in PATH(M, ds)$ such that $\mathit{FirstS}(\sigma) = t^{M,v}$, $M, v, \sigma \models \pi$

– $M, v, \sigma \models \mathit{state\_cond}(x \bullet F)$ iff $M, v[\mathit{FirstS}(\sigma)/x] \models F$

– $M, v, \sigma \models \mathit{label\_cond}(x \bullet F)$ iff
$\mathit{FirstL}(\sigma)$ is defined and $M, v[\mathit{FirstL}(\sigma)/x] \models F$

– $M, v, \sigma \models \pi_1 \;\mathit{until}\; \pi_2$ iff there exists $j \geq 0$ such that for all h, $0 < h < j$, $M, v, \sigma|_h \models \pi_1$ and $M, v, \sigma|_j \models \pi_2$

– $M, v, \sigma \models \mathit{eventually}\; \pi$ iff there exists $j \geq 0$ such that $M, v, \sigma|_j \models \pi$

– $M, v, \sigma \models \mathit{always}\; \pi$ iff for all $j \geq 0$ such that $\sigma|_j$ is defined, $M, v, \sigma|_j \models \pi$

– $\neg\, F, F \;\Rightarrow\; F', \forall\, x \bullet F, \;\neg\, \pi, \pi \;\Rightarrow\; \pi', \forall\, x \bullet \pi$ as usual.

A *generalized dynamic signature* is a dynamic signature $D\Sigma = (S, DS, TF, PF, P)$ where for all $ds \in DS$ there exist $\mathit{Label\_ds}, \mathit{Info\_ds} \in S - DS$ and $\_ : \_ \xrightarrow{\_} \_ \in P_{\mathit{Info\_ds} ds\; \mathit{Label\_ds}\; ds}$; and $\mathit{Info\_ds}$ is the sort of the *information* associated to each transition.

The models of a generalized dynamic many-sorted signature is a *glts*, and the presence of the *Info_ds* sorts and of the new transition predicate that has an additional argument of sort *Info_ds* modifies the set of the "paths" for a dynamic sort as follows:

for each $ds \in DS$, the set of the *generalized paths* on M of sort ds, denoted by $GPATH(M, ds)$, is the set of all the sequences of transitions having the form either (1) or (2) below:

(1) $i_0\ d_0\ l_0\ i_1\ d_1\ l_1\ i_2\ d_2\ l_2\ \ldots$ (infinite path)
(2) $i_0\ d_0\ l_0\ i_1\ d_1\ l_1\ i_2\ d_2\ l_2\ \ldots\ d_n$ $n \geq 0$

where for all $i \geq 0$, $(i_i, d_i, l_i, d_{i+1}) \in \rightarrow^M$, and there do not exist l, i_n, d' such that $(i_n, d_n, l, d') \in \rightarrow^M$.

Now we define the extension of the sentences.

The dynamic sentences in $\mathbf{GTSen}(D\Sigma)$ and the path formulae for the elements of the various dynamic sorts of the generalized dynamic signature $GD\Sigma$ are simply enriched by adding the atomic path formula that describe properties over the information component of a transition.

The new path formulae for the elements of sort ds, are defined as the one in $\mathbf{PSen}(D\Sigma, ds)$, plus

- $\mathit{info\_cond}(x \bullet F)$ x variable of sort *Info_ds*, $F \in \mathbf{TSen}(D\Sigma)$
  This formula holds on a path σ whenever F holds at the first information element of σ.

The validity of a dynamic formula over a model of a generalized dynamic model is slightly modified, to take into account the new kind of paths.

D.5 Observational Semantics

D.5.1 Strong Bisimulation

The *strong bisimulation* semantics corresponds to the idea that two process behaviours can be identified if an external observer, that can only see the actions labeling the transitions, cannot distinguish them. This semantics is formally defined by means of a congruence on the models, called the *maximum strong bisimulation relation*, that is the union of the *strong bisimulation relation* ([Mil90, D.P81]).

If we consider the models $glts_1$= ($State_1$, $Label_1$, $\rightarrow_1$, $Info_1$) and $glts_2$= ($State_2$, $Label_2$, $\rightarrow_2$, $Info_2$), described by two *gltss*, a relation $\mathcal{R} \subseteq State_1 \times State_2$ is a *(strong) bisimulation relation* among iff:

whenever $st_1\ \mathcal{R}\ st_2$ then:

- for all $st'_1 \in State_1$, $l_1 \in Label_1$, $\mathtt{in}_1 \in Info_1$ if $\mathtt{in}_1 : st_1 \xrightarrow{l_1} st'_1$ then there exists $st'_2 \in State_2$, $l_2 \in Label_2$, $\mathtt{in}_2 \in Info_2$ s.t. $\mathtt{in}_2 : st_2 \xrightarrow{l_2} st'_2$ $st'_1\ \mathcal{R}\ st'_2$.
- for all $st'_2 \in State_2$, $l_2 \in Label_2$, $\mathtt{in}_2 \in Info_2$ if $\mathtt{in}_2 : st_2 \xrightarrow{l_2} st'_2$ then there exists $st'_1 \in State_1$, $l_1 \in Label_1$, $\mathtt{in}_1 \in Info_1$ s.t. $\mathtt{in}_1 : st_1 \xrightarrow{l_1} st'_1$ $st'_1\ \mathcal{R}\ st'_2$.

The theory of bisimulation has some properties, among which:

$$\mathcal{S} = \cup\{R|\mathcal{R} \ \ \textit{is a bisimulation}\}$$

is an equivalence relation.

The existence of $\mathcal{S}$ comes from standard results concerning largest pre-fixed-points ([Mil90, D.P81])

If we call $x \xrightarrow{l} st'$ an *experiment* for $glts_1$, we can rephrase the definition above as follows:

R is a bisimulation iff:

whenever $st_1\ \mathcal{R}\ st_2$ then:

- for all the experiment obs_1, if st_1 *passes* obs_1, then exists an experiment obs_2 *similar* to obs_1s.t. st_2 *passes* obs_2.
- for all the experiment obs'_2, if st_2 *passes* obs'_2, then exists an experiment obs'_1 *similar* to obs'_2s.t. st_1 *passes* obs'_1.

If obs= $x \xrightarrow{l} st'$, then st passes obs iff $st \xrightarrow{l} st'$.

In the case of a strong bisimulation relation, we say that an experiment $x \xrightarrow{l_1} st_1$ is *similar* to all and only the experiments of the form $x \xrightarrow{l_2} st_2$ with $st_1\ \mathcal{R}\ st_2$ and $l_1 \mathcal{L}\ l_2$.

Thus the similarity relation depends on $\mathcal{R}$ and we can introduce a relation C($\mathcal{R}$) among experiments such that: $x \xrightarrow{l_1} st_1$ C($\mathcal{R}$) $x \xrightarrow{l_2} st_2$ iff $st_1\ \mathcal{R}\ st_2$ and $l_1 \mathcal{L}\ l_2$.

D.5.2 Weak Bisimulation

If we need a semantic equivalence relation that does not distinguish processes if they differs only in the number of non-observable, internal transitions, we can replace the previous definition of experiments by using a new predicate $\Rightarrow$: *State* $\times$ *Label* $\times$ *State* $\times$ *Info* defined by the following inductive rules:

$$\frac{i : st_1 \xrightarrow{l} st_2}{st_1 \underset{S}{\overset{l}{===>}} st_2}$$

$$\frac{i : st_1 \xrightarrow{\tau} st_2 \quad st_2 \underset{S}{\overset{l}{===>}} st_3}{st_1 \underset{S}{\overset{l}{===>}} st_3}$$

$$\frac{st_1 \underset{S}{\overset{l}{===>}} st_2 \quad i : st_2 \xrightarrow{\tau} st_3}{st_1 \underset{S}{\overset{l}{===>}} st_3}$$

The new experiments have the form $x \underset{S}{\overset{l}{===>}} st$. The similarity relation is analogous to the one used in the case of strong bisimulation.

Bibliography

[AR87a] E. Astesiano and G. Reggio. SMoLCS-Driven Concurrent Calculi. In *Proc. TAPSOFT'87, Vol. 1*, chapter 249 LNCS. Springer Verlag, Berlin, 1987.

[AR87b] E. Astesiano and G. Reggio. The SMoLCS Approach to the Formal Semantics of Programming Languages – A Tutorial Introduction. In *System Development and Ada, Proc. of CRAI Workshop on Software Factories and Ada, Capri*, chapter 275 LNCS. Springer Verlag, Berlin, 1987.

[AR96] E. Astesiano and G. Reggio. Labelled Transition Logic: an Outline. Technical Report DISI-TR-96-20, DISI, Università di Genova, 1996.

[AR97] E. Astesiano and G. Reggio. *Formalism and Method*. In M. Bidoit and M. Duchet, editors, *Proc. TAPSOFT'97*, pages 93–114, Berlin, 1997. Springer Verlag.

[BE98] Schulte W. Beorger E. Programmer Friendly Modular Definition of the Semantics of Java. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java,*, pages 175–181, 1998. LNCS 1998.

[CR99] E. Coscia and G. Reggio. JTN: A Java-Targeted Graphic Formal Notation for Reactive and Concurrent Systems. In *Proc. FASE'99: Fundamental Approaches to Software Engineering*, chapter 1577 LNCS. Springer Verlag, Berlin, 1999. Available at `http://www.disi.unige.it/CosciaE/papers/JTNFase99`.

[D.H87] D.Harel. Statecharts: a Visual Formalism for Complex Systems. *Science of Computer Programming*, (8), 1987.

[DH98] W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. Technical Report CS98-09, Faculty of Mathematics and Computer Science The Weizmann Institute of Science, May 1998.

[D.P81] D.Park. Concurrency and Automata on Infinite Sequences. In *Proc. 5th GI Conference*, page LNCS 104. Sprimger Verlag, 1981.

[EA96] G. Reggio E. Astesiano. A Dynamic Specification of the RPC-Memory Problem. In S.Merz M. Broy and K.Spies, editors, *Formal System Specification. The RPC-Memory Specification Case Study*, pages 67–108. Springer Verlag, 1996. LNCS 1169.

[E.A99] G.Reggio E.Astesiano, C.Choppy. CASL-LTL: A CASL Extension for Dynamic Reactive Systems. Technical report, CoFI Document, December 1999. available at: ftp.disi.unige.it/person/ReggioG/CASLTL/CASLTL.ps.

[EB98] W. Schulte E. Boerger. Defining the Java Virtual Machine as Platform for Provably Correct Java Compilation. In J. Zlatuska L. Brim, J. Gruska, editor, *Proc. MFCS'98*, pages 17–35, LNCS 1450, 1998. Springer Verlag.

[EG94] E.Astesiano and G.Reggio. Formally-Driven Friendly Specification of Concurrent Systems: a Two-Rail Approach. Technical Report DISI-TR-94-20, DISI-Universitá di Genova, Italy, 1994. Presented at ICSE'17-Workshop on Formal methods,1995.

[EG95] E.Coscia and G.Reggio. Deontic Concepts in the Algebraic Specification of Dynamic Systems: the Permission Case. In M. Haveraaen, O.Owe, and O-J. Dahl, editors, *Recent Trends in Data Type Specifications*, pages 161–181, Oslo, Norway, September 1995. Springer Verlag.

[EG98] E.Coscia and G.Reggio. *A Proposal for an Abstract Semantics of Shared Objects in Multi-threaded Java.* 1998. submitted to the *OOPSLA Formal Underpinnings of Java* workshop.

[EG99] E.Coscia and G.Reggio. *An Operational Semantics for Java.* Technical report, DISI, July 1999.

[ER99] E.Coscia and G. Reggio. *An Abstract Operational Semantics for a Subset of Java Programs.* Technical report, DISI, 1999. in preparation.

[F.P94] G.Reggio F.Parodi. METAL: a metalanguage for SMoLCS. Technical Report DISI-TR-94-13, DISI¡ University of Genova, 1994.

[GG87] G.Costa and G.Reggio. *Specification of Abstract Dynamic Data Types: a Temporal Logic Approach. TCS*, (173), 1987. to appear.

[G.R94a] A.Morgavi G.Reggio, D.Bertello. The Reference Manual for the SMoLCS Methodology. Technical Report DISI-TR-94-17, University of Genova, DISI, 1994.

[G.R94b] E.Crivelli G.Reggio. Specification of an Hydroeletric Power Station: Revised Tool-Checked Version. Technical Report DISI-TR-1994-17, University of Genova, DISI, 1994.

[HMM86] R. Harper, D. MacQueen, and R. Milner. Standard ml. Technical Report ECS-LFCS-86-2, 1986.

[ITU93] *Z.120: Message Sequence Chart (MSC)*. Technical report, ITU, Geneva, 1993.

[J.G96] G. Steele J.Gosling, B. Joy. The Java Language Specification. The Java Series. Addison Wesley, 1996.

[Mil80] R. Milner. A Calculus of Communicating Systems, volume 92 of *Lecture Notes in Computer Science*. Springer Verlag, Berlin, 1980.

[Mil90] R. Milner. *Operational and Algebraic Semantics of Concurrent Processes.* In *Handbook of Theroretical Computer Science, Vol B: Formal Models and Semantics*. Elsevier and MIT Press, 1990.

[ML97] G. Reggio M. Larosa. *A Graphic Notation for Formal Specifications of Dynamic Systems.* In Cliff B. Jones J.Fitzgerald and P.Lucas, editors, *FME '97: Industrial applications and Strenghtned Foundations of Formal Methods*, Graz, Austria, September 1997. 4th. International Symposium of Formal Methods Europe, Springer Verlag.

[Mos97] P.D. Mosses. *CoFI: The Common Framework Initiative for Algebraic Specification and Development.* In *Proc. TAPSOFT '97*, Berlin, 1997. 1214 LNCS, Springer Verlag.

[oLD99] The CoFI Task Group on Language Design. Casl: The common algebraic specification language summery. Technical Report Version 1.0, CoFI WG, November 1999. Available at: *http://www.bricks.dk/Project/CoFI/Documents/CASL/Summary.*

[PBM97] A. Knapp P.Cenciarelli, B.Reus, and M.Wirsing. *From Sequential to Multi-Threaded Java: An event Based Operational Semantics.* In M.Johnson, editor, *Algebraic Methodology and Software Technology.* AMAST '97, Springer, July 1997.

[Plo81] Gordon D. Plotkin. *A Structural Approach to Operational Semantics (Lecture Notes).* Technical report, DAIMI FN-19, Aarhus University, 1981.

[PRM98] A.Knapp P.Cenciarelli, B. Reus, and M.Wirsing. *An Event-Based Structural Operational Semantics of Multi-Threaded Java.* In Formal Syntax and Semantics of Java. Springer, 1998.

[Rat97a] Rational. *UML Notation Guide Version 1.1.* http://www.rational.com/uml/html/notation/, September 1997. On-line documentation.

[Rat97b] Rational. *UML Semantics.* http://www.rational.com/uml/html/semantics/, September 1997. On-line documentation.

[Reg98] G. Reggio. *A Method to Capture Formal Requirements: the INVOICE Case Study.* In *Int. Workshop Comparing Specification Techniques.* Universite de Nantes, 1998.

[Rei82] Wolfgang Reisig. *Petri Nets : an introduction.* EATCS Monographs on Thoretical Computer Science. Springer Verlag, 1982.

[Ros] Rational rose. information available on-line at: http://www.rational.com/products/rose.

[Rus98] M. Russo. *Formal Executable Semantics for Java.* Vancouver, Canada, October 1998. proceedings of OOPSLA'98 workshop: Formal Underpinnings of Java.

[S.D97] S. Eisenbach S.Drossopoulou. *Java is Type Safe- Probably. 11th European Conference on Object Oriented Programming*, June 1997.

[SDL93] Z.100 ITU: Specification and Design Language SDL. Technical report, ITU, Geneva, 1993.

[U.S94] U.S. Naval Postgraduate School, Monterey, California. *1994 Monterrey Workshop:* Increasing the Pratical Impact of Formal Methods for Computer-Aied Software Development: Software Evolution., September 1994.

[W3C98] W3C. *Document Object Model (DOM): Level 1 Specification.* Technical report, W3C, October 1998. Available at: http://www.w3.org/TR/REC-DOM-Level-1.