

# DO-Casl: an Observer-Based Casl Extension for Dynamic Specifications

Matteo Dell’Amico and Maura Cerioli

DISI–Dipartimento di Informatica e Scienze dell’Informazione  
Università di Genova, Via Dodecaneso, 35, 16146 Genova, Italy  
dellamico@disi.unige.it, cerioli@disi.unige.it

**Abstract.** We present DO-CASL, a new member of the CASL family of specification languages. It is an extension of CASL-LTL and it supports a methodology for conveniently writing loose specifications of observers on dynamic sorts. The need for such constructs arose during the development of a CASL library for distributed systems. Indeed, we have frequently used the same pattern of specification, in order to solve a generalization of the frame problem while using observers. The constructs we propose make the resulting specifications more readable, concise and maintainable. The semantics of our extension is given by reduction to standard CASL-LTL, which is, in turn, reducible to standard CASL whenever temporal logic is not used. A small prototype of the pre-processor from DO-CASL to CASL-LTL has been implemented as well.

## Introduction

Middleware is widely and successfully used to support programmer productivity, by providing solutions to the most common tasks and abstractions of low-level concepts. Many software projects are developed by writing mainly “glue code”, connecting the functionalities made available from middleware. Thus, the usefulness of a programming language or environment is greatly influenced by the quality and quantity of available libraries.

We argue that this is also true for specification languages. That is, besides libraries for standard datatypes (such as lists, sets or integers), software specifications also need libraries for middleware primitives. This way, the developer is lifted from the burden of axiomatization of components that can sometimes be very complex. Moreover, they are part of the context, not to be implemented anew for each system development.

Our work stems from the definition of a CASL-LTL library for decentralized distributed systems.

We decided to use one of the CASL family of languages, because of its large acceptance in the scientific community. Among them, we selected CASL-LTL, as it provides an intuitive representation of label transition systems and hence of dynamic systems.

Our library supports a model driven development<sup>1</sup>: our specifications abstract away from concrete middleware to propose coherent sets of functionalities which are implemented, in most cases, by several infrastructures. Technically, this means that our specifications are loose. In order to allow different implementations and to get flexibility for further extensions of each given specifications by other primitives, we adopt an observer oriented style of specification. Observers are used to hide details of the internal states, which are only partially known in this layer of abstraction. Since we assume our knowledge of the needed observers to be incomplete, we do not deduce equality of two states from all the current observers yielding the same result on both.

Though our specifications are loose, we want to state the observable properties of the primitives, for the user to build upon them. Therefore, in this context loose does not mean underspecified.

For instance, in a specification of the primitives for connection to, and disconnection from the network, we want to axiomatize not only that immediately after a connection (disconnection) the system is connected (disconnected), but also that each connection is standing till the next (explicit) disconnection.

This is a very simple example of a common problem. It is, indeed, often the case that only a small subset of the transitions may affect the result of an observer. Therefore, the specifier should explicitly axiomatize the *frame assumption* (which, roughly speaking, is “nothing changes unless explicitly required”) for the observers, in order to be able to deduce that their results are unaffected by most transitions. This problem is further complicated by the need for flexibility. Indeed, the definition of an observer may be in a specification, while that of some specific transitions affecting its result may be in another specification, extending or using the first one.

In this paper, we present an extension of CASL-LTL providing constructs to solve these problems, by automatically adding the axioms to capture the frame assumption. Thus, it enables a readable and compact style for the development of observer-oriented dynamic specifications. Our proposal is highly modular, because the information needed to state that some transition possibly affects the result of an observer are colocated with the definition of the transition itself. Thus, adding new transitions do not require to change the specification of the observer.

The semantics of the language we propose, DO-CASL (for Dynamic Observer-based CASL), is given by reduction to CASL-LTL. Among the different possible way to translate the language, we selected one that yields standard CASL if the input DO-CASL specification does not contain temporal logic axioms. We think this choice to be more readable for the average user and it surely allows to reuse the existing tools for CASL in most cases, while no tools for the temporal logic extension exist so far.

---

<sup>1</sup> Actually, the library includes specifications in different layers of abstraction, in order to support more detailed design, even committed to a specific technology. But, here we focus on the more abstract layer.

**Paper structure** Section 1 introduces the preliminaries about CASL, Sec. 2 describes the style of specifications adopted, Sec. 3 presents DO-CASL, and finally Sec. 4 shows how the language may be used in an extended example.

## 1 Casl and Casl-Ltl

The CASL algebraic specification language has been developed as central part of the CoFI initiative<sup>2</sup>. It provides constructs for declaring basic and structured specifications, whose semantics is a class of *partial* first-order models, and architectural specifications, whose semantics is, roughly speaking, a (higher-order) function over the semantics of basic specifications. Thus, the natural semantics of CASL specifications is the *loose* one: all the partial first-order structures satisfying its axioms are models of a basic specification. However, the models may be restricted to the initial (free) ones, by means of a structuring construct, so that methods based on initial semantics may be accommodated as well.

The building blocks of basic specifications are declarations of (sub)sorts, operations and predicates, giving a signature, and axioms on that signature. Operations may be total or *partial*, denoted by a question mark in the arity. CASL also accommodates subsorting; here, anyway, we do not explicitly use it.

The structuring operators are the usual in algebraic specification languages, like sum, (free) extension, renaming and hiding. We will use mostly union, extension and generic specifications. The latter being less standard, let us discuss a bit its semantics and usage. A generic specification is named and consists of

- a list of *formal parameters*, which are place-holder specifications to be replaced, in the instantiation phase, by more detailed specifications, the actual parameters, possibly using a *fitting morphism* to connect the symbols used in the formal parameters to those in the actual parameters;
- a list of *imports*, which are specifications to be used as they are, for instance that of integer numbers;
- a *body* specification, describing the features to be added to the parameters and the imports by the specification.

The result of an instantiation is, roughly speaking, the enrichment of the union of actual parameters and imports with the body, where body and parameters are translated using the fitting morphisms, if they exist.

For a complete description of CASL, we refer to [1].

**Casl-Ltl and Generalized Labeled Transition Systems** It is important to note that CASL is one of a *family* of languages, sharing common constructs and their semantics. For instance, there are restrictions of CASL without partial functions, and/or subsorting, and/or predicates, so that the resulting languages may be translated in other less rich languages in order to reuse specialized tools. On the converse, there are extensions of CASL by constructs and corresponding

---

<sup>2</sup> See <http://www.brics.dk/Projects/CoFI> and <http://www.cofi.info/>.

semantics to deal with specific problem. For instance, there is higher-order CASL (see e.g. [2]) and state-based CASL (see e.g. [3]).

In the sequel we will use CASL-LTL (see [4]), which is designed to describe *generalized labeled transition systems* (glts from now on).

A glts may be used to represent the evolution of a dynamic system. It consists of a set of *states* of the system, one of *labels*, one of *information* and finally the *transition relation*, representing the evolution capabilities of the system. Any element of the transition relation is a tuple consisting of the starting and the final states, a label, capturing what is relevant to the external world, and an information, for what is relevant only to the system itself. For instance, if a system is keeping track of the number of sent messages, the transition corresponding to sending all the messages in a queue will have the message list coded in the label and the number of sent messages in the info part, to be used to update the internal counter. Any state of the system corresponds to the process having an evolution tree determined by the transition system itself, where each branch is given by a transition of the system and represents a *capability* of moving of the parent state.

A glts may be specified by using CASL-LTL. Indeed, CASL-LTL allows to declare dynamic sorts, using the construct **dsort** *ds* **label** *l\_ds* **info** *i\_ds*. This CASL-LTL construct semantically corresponds to the declaration of the sorts *ds*, *l\_ds*, and *i\_ds* for the states, the labels and the information of the glts, and of the transition predicate **preds**  $-- :: -- \xrightarrow{\quad} -- : i\_ds \times ds \times l\_ds \times ds$ , as well.

Thus, each element *s* of sort *ds* in a model *M* (an algebra or first-order structure) of the above specification corresponds to a process modelled by a transition tree with initial state *s* determined by the glts  $(i\_ds^M, ds^M, l\_ds^M, -- :: -- \xrightarrow{\quad} --^M)^3$ .

The most important extension of CASL-LTL w.r.t. CASL is the enrichment of the logic by constructs from a branching-time CTL-style temporal logic, which effectively increase the expressive power of the language (see [5] and [4]).

In the sequel we will use an obvious shortcut when either the label part or the information one are irrelevant, dropping any reference to the immaterial aspect using transition predicates such as  $-- \xrightarrow{\quad} -- : ds \times l\_ds \times ds$  and  $-- :: -- \rightarrow -- : i\_ds \times ds \times ds$ . The general case is computed from the shorter version, by adding a sort with just one element for the missing component and decorating all the transitions with that element.

## 2 Lessons Learned from Developing a Library for Distribution

Our CASL-library for distribution, as motivated in the Introduction, is hierarchically organized in a directed acyclic graph of refinements<sup>4</sup>, where all the nodes are

<sup>3</sup> Given a  $\Sigma$  algebra *A*, and a sort *s* of  $\Sigma$ ,  $s^A$  denotes the interpretation of *s* in *A*; similarly for the operation and predicates of  $\Sigma$ .

<sup>4</sup> Here, *refinements* has the traditional meaning of model-class inclusion. This property is guaranteed in our specifications by the use of the extension construct.

loose specifications, having as models all those middleware implementing some set of functionalities. Thus, the refinement in this context corresponds mostly to adding functionalities, not to making implementation decisions (though we also provide a few detailed specifications representing an individual middleware).

In order to achieve this result, we adopt an observational style, in the sense that we introduce functions and predicates to extract from the elements (representing internal states of the subsystems) the values of some of their aspects, which we regard as relevant for the applications to be built upon our infrastructure. However, we are not relying on the observers to distinguish elements, as in most observational approaches.

Indeed, by the nature of our library, the set of observers is continuously extended, as new aspects of the nodes and networks are introduced by the library specifiers and end-users. Thus, the fact that the current set of observers cannot distinguish between two elements is not a clue of their equality; it could as well be an indication of some aspect still to be taken into account. Therefore, our approach has in common with more traditional observational approaches (e.g., the pioneering [6]; we defer to [7] for further references) only the intuition of the black-box approach and the use of the word observer.

For instance let us consider the case of the most basic specification in our library, the one of *peer*, as a paradigmatic example of the difficulties we encountered in the development of the library and the solution we propose.

A *peer* is the abstraction of any node in a net. It has a persistent identity, the capability to connect to a net using a given address and to disconnect from the net. Thus, we leave underspecified how elements of the peer sort are made and introduce functions extracting the identity, address (possibly undefined if the peer is not connected), and online status from such elements, as in the following signature<sup>5</sup>:

```

sort  PeerId, Address
dsort Peer label PeerL
ops  online : Address → PeerL
       offline :→ PeerL
       id : Peer → PeerId
       addr : Peer →? Address
preds isOnline : Peer

```

where we have (static) sorts, describing data types, like for instance the (totally unspecified) sort for peer identifiers, *PeerId*, or that for the labels of their transitions, *PeerL*. Moreover, we also have the dynamic sort *Peer*, representing the states of the nodes. Analogously, we have operations building some sort, like for instance *online* and *offline*, which denote particular labels, and we have observers, both operations like *id* and *addr*, and predicates like *isOnline*, used to extract, or observe, aspects of the peer states.

Now, we need to state two different kinds of axioms. First of all, we have the standard axioms, describing the effects of operations and transitions, such as asserting that after going online with an address *a*, the peer is actually online and its address is *a*. But, we also have to state that no transition is affecting

---

<sup>5</sup> Notice the obvious adaptation to the case with silent information.

the value of  $id$ , as the identity is persistent, that the only transitions affecting *isOnline* are those actually taking the peer on and off line, and that the address is persistent for each connection, so that it can change only if some connection or disconnection has taken place. These are quite different from the previous ones, from a logical point of view, because they express a property that the users usually implicitly assume: *each aspect of the status of the system changes only if forced to, by a transition explicitly modifying it*. But, there is no such a thing as an *implicit* assumption in specifications. Unless some axiom is imposed to guarantee it, there are models which do not satisfy it. In other words, we have to state a sort of *frame assumption* (see e.g. [8]) for some observers.

However, there are two important differences from standard frame assumption in our approach. First of all, we want to explicitly state that some properties of the system change and some do not for a given transition, but leave most of them underspecified, changing or not depending on the individual models. Thus, the frame assumption would hold only for a subset of the functions and predicates on the dynamic sorts, those we call *observers*.

Moreover, we need a flexible way to state the frame assumption to accommodate further refinements of the specification. Let us for instance consider the problem of stating the persistency of the address in each continuous connection. If we simply add the axiom<sup>6</sup>

$$\forall l : PeerL; \forall p, p' : Peer; \bullet \text{ addr}(p) = \text{addr}(p') \text{ if} \\ (p \xrightarrow{l} p' \wedge (\forall a : Address. \neg l = \text{online}(a)) \wedge \neg l = \text{offline})$$

then we forfeit the capability to add in an extension of this specification a label representing another way of connecting, for instance the connection without explicit address, for those cases where the address is dynamically provided by a server. Indeed, even if we add such labels, they cannot change the address of the peer, being different from *online* and *offline*.

This would be clearly unacceptable in our approach, where new refinements of the specifications can be added to represent richer middleware or to support more demanding applications. Following our observational approach, instead of stating that only transitions using some individual labels may affect an observer, we describe abstract properties on the labels and require that only the labels satisfying them can affect the predicate. For instance, in our example, the property of being online may be influenced by all the labels representing a connection or a disconnection, but by no others. Thus, we use again predicates on the labels and info to describe the category of actions they are representing and use these predicates in turn to state our weak form of frame assumption. In this way we achieve a separation of concerns and a higher level of modularity: at the moment when observers are introduced, the specifier has to decide with categories of transitions may change the observation. But, it is only at the moment of the definition of each individual transition, that is, of its label and info, that the decision about which are its categories has to be made.

It is worth noting that the actual labels become superfluous and can be dropped from the specifications of the abstract behavior of the middleware

---

<sup>6</sup> This approach is similar to the expansion of “not changed by other events” in [9].

classes. The actual labels appear, together with the axioms categorizing them w.r.t. the observers, in the lower level specifications, those representing individual middlewares and in the end-user specifications, where they are used to describe the moves of the system at the applicative level.

Thus, our toy example should be changed as follows

```

sort PeerId, Address
dsort Peer label PeerL
ops id : Peer → PeerId
      addr : Peer →? Address
preds goesOnline : PeerL
        goesOffline : PeerL
        isOnline : Peer
∀ l : PeerL; p, p' : Peer
  • id(p) = id(p') if p  $\xrightarrow{l}$  p'
  • addr(p) = addr(p') if (p  $\xrightarrow{l}$  p' ∧ ¬goesOnline(l) ∧ ¬goesOffline(l))
  • def(addr(p)) if p  $\xrightarrow{l}$  p' ∧ goesOnline(l)

```

Though, technically, this is satisfactory, from a methodological point of view it is not. Indeed, the end user is required to add lots of trivial axioms of the form

- $obs(d) = obs(d')$  if  $(i : d \xrightarrow{l} d' \wedge \neg cat_1(l, i) \wedge \dots \wedge \neg cat_n(l, i))$  to state that the transition (s)he is introducing does not affect the result of most observers. But, the user should be more encouraged to focus on the pairs “transition + observer” where the transition is relevant to that observer, than on those where, being no relationship between the two components, things are not going to change and hence the corresponding axiom has to be issued.

We need a mechanism to clearly separate the axioms stating which category of transitions affects which observer, from those describing the effects and, moreover, to automatically add the axiomatization of the default behavior, where the observer values are not changing unless some transition affecting the corresponding aspect takes place.

### 3 DO-Casl

Let us introduce a syntactic short-cut, which does not require any change in the semantics, because the terms introduced by this new construct reduces to terms in standard CASL-LTL, and then CASL in the case no temporal logic is used in the specification. In the choice of the restrictions for such a construct, we have been guided by pragmatic considerations, choosing a generality sufficient to deal with all the cases in our library and, at the same time, not so extreme to make the translation in standard CASL difficult.

We start by adding a new production for the SPEC non-terminal of the grammar of the abstract syntax. The terms generated by this production will correspond to the dynamic specifications using observers, that is a special case of CASL-LTL basic specifications including one or more *observer blocks*.

Any observer block refers to a dynamic sort and encapsulates the definition of observers on that sort, together with categories of transitions which may affect them and axioms to express this dependency.

**Definition 1.** *The context free grammar of DO-CASL adds to that of CASL-LTL the following productions (terminal and non-terminal symbols):*

SPEC	::= ...   DSPEC
DSPEC	::= dyn-spec DBASIC-ITEMS*
DBASIC-ITEMS	::= BASIC-ITEMS   OBS-BLOCK
OBS-BLOCK	::= obs-block DSORT-DECL OBS-ITEMS*
OBS-ITEMS	::= OP-ITEMS   PRED-ITEMS   CATEGORY-ITEMS   VAR-ITEMS   CAT-AXIOM-ITEMS
CATEGORY-ITEMS	::= category PRED-DECL+
CAT-AXIOM-ITEMS	::= axiom-items CAT-FORMULA+
CAT-FORMULA	::= CAT-QUANTIFICATION   CAT-IMPLICATION
CAT-QUANTIFICATION	::= quantification universal VAR-DECL CAT-IMPLICATION
CAT-IMPLICATION	::= implication PREDICATION AFFECTS   implication AFFECTS AFFECTS
AFFECTS	::= affects VAR VAR PREDNAME   affects VAR VAR OPNAME

The concrete syntax is as close to CASL-LTL as possible.

**Definition 2.** *The concrete syntax for DO-CASL is the same as that of CASL-LTL for the terms common to both languages. For the newly added terms, the concrete syntax is as follows:*

- a DSPEC starts with the keyword **dspec** and ends with **end\_dspec**<sup>7</sup>;
- an OBS-BLOCK starts with the keyword **observe** and ends with **end\_obs**;
- a CATEGORY-ITEMS starts with the keyword **cats**;
- the concrete syntax for CAT-AXIOM-ITEMS, CAT-FORMULA, CAT-QUANTIFICATION and CAT-IMPLICATION is the same as for the corresponding languages in CASL;
- $(l, i)$  affects  $o$  is the concrete syntax for a term of the form *affects l i o*.

Besides the static correctness of standard CASL, we also impose a few requirements.

**Definition 3.** *For an observer block referring to a dynamic sort **dsort**  $ds$  label  $l\_ds$  **info**  $i\_ds$  to be statically correct, we require that:*

- all the enclosed operations and or predicates must have  $ds$  as (unique) source; in the following we will call them observers on  $ds$ ;
- all declarations of predicates in a CATEGORY-ITEMS section must have  $l\_ds \times i\_ds$  as source; in the following we will call them categories on  $l\_ds \times i\_ds$ ;
- in each axiom,
  - the predicate symbols in a PREDICATION must be declared as categories within the same block and their arguments must be local variables of appropriate sort;

<sup>7</sup> Mandatory, to simplify parsing.

- the operation and predicate names in an *AFFECTS* must be declared as observers within the same block;
- the variables in an *AFFECTS* must be of sort *l\_ds* and *i\_ds*, respectively.

Since the source for observers and categories must agree with the static requirements, they could be deduced from the observer block head. Though requiring them to be explicitly stated is then redundant, and a possible source of error, we prefer to use this verbose syntax, because it is closer to the standard CASL declarations of functions and predicates and hence, supposedly, easier for the end-users. Specialized editors could be easily devised, which automatically supply the deducible sorts in the declaration of observers and categories, to save the users the pain of writing them.

Let us consider as an example the peer specification, using the syntactic sugar introduced so far to represent the observers. Notice that the operations *online* and *offline* have been dropped, because their role is filled by the corresponding predicates. Moreover, we give here the full specification, with also the axioms external to the block. Finally, note that the specification is parametric over the definition of the addresses (e.g., IPv4, IPv6, JXTA or Chord identifiers, etc.)

```
spec PEER [sort Address]=
  dspec
  sort PeerId
  observe Peer label PeerL
  ops id : Peer → PeerId
      addr : Peer →? Address
  preds isOnline : Peer
  cats  goesOnline : PeerL
      goesOffline : PeerL
  ∀ l : PeerL
    • goesOnline(l) ⇒ l affects isOnline
    • goesOffline(l) ⇒ l affects isOnline
    • l affects isOnline ⇒ l affects addr
  end_obs
  ∀ l : PeerL; p, p' : Peer
    • isOnline(p') if p  $\xrightarrow{l}$  p' ∧ goesOnline(l)
    • ¬isOnline(p') if p  $\xrightarrow{l}$  p' ∧ goesOffline(l)
    • def(addr(p)) if isOnline(p)
  end_dspect
```

Now, let us define the semantics of our constructs, by reduction to CASL-LTL. The basic intuition is to translate categories to standard predicates and to add a predicate *ad hoc* to represent the capability of affecting a given observer. Moreover, we want to keep only those models where such *ad hoc* predicates are minimal with respect to the axioms given in the observer block and the interpretation of the categories on the individual model. To get this result, we will intersect the models of the translated specification with those of a free, and hence minimal, specification for the *ad hoc* predicates.

**Definition 4.** For a correct observer block  $obs\_blk$ <sup>8</sup>

```
observe ds label l_ds info i_ds
  ops f1 : ds →? s1; ... fn : ds →? sn;
  preds p1, ..., pm : ds;
  cats pt1, ..., ptk : l_ds × i_ds
  vars x11, ..., x1n1 : s1; ...; xk1, ..., xknk : sk;
  φ1 ... φh
```

**end\_obs**

its expansion, denoted  $DOCASL2CASL(obs\_blk)$ , is the following basic specification

```
dsort ds label l_ds info i_ds
ops f1 : ds →? s1; ... fn : ds →? sn;
preds p1, ..., pm : ds; pt1, ..., ptk : l_ds × i_ds
      _aff_f1, ..., _aff_fn, _aff_p1, ..., _aff_pm : l_ds × i_ds
∀ l : l_ds; i : i_ds; d, d' : ds;
  %% transitions not affecting f1 ... fn, p1 ... pm leave the observer result unchanged
  (¬_aff_f1(l, i) ∧ i : d  $\xrightarrow{l}$  d' ⇒ f1(d) = f1(d'))
  ...
  (¬_aff_pm(l, i) ∧ i : d  $\xrightarrow{l}$  d' ⇒ (pm(d) ⇔ pm(d'))
```

Moreover we will call  $free\_aff(obs\_blk)$  the 4-tuple

$$(SSp(obs\_blk), CSp(obs\_blk), OSp(obs\_blk), ASp(obs\_blk))$$

where

- $SSp(obs\_blk)$  is **sorts**  $l\_ds, i\_ds$
- $CSp(obs\_blk)$  is **preds**  $pt_1, \dots, pt_k : l\_ds \times i\_ds$
- $OSp(obs\_blk)$  is **preds**  $\_aff\_f_1, \dots, \_aff\_f_n, \_aff\_p_1, \dots, \_aff\_p_m : l\_ds \times i\_ds$
- and  $ASp(obs\_blk)$  is  $\forall x\_l\_ds : l\_ds; x\_i\_ds : i\_ds; trans(\varphi_1) \dots trans(\varphi_h)$ , where  $trans$  drops any variable quantification and transforms each occurrence of  $(l, i)$  affects  $o$  into  $\_aff\_o(x\_l\_ds, x\_i\_ds)$ , where  $x\_l\_ds$  and  $x\_i\_ds$  are fixed variable names.

A correct dynamic specification including observer blocks  $obs\_blk_1 \dots obs\_blk_n$  translates to the specification given by leaving all the **BASIC-ITEMS** unaffected, by replacing each  $obs\_blk_i$  by  $DOCASL2CASL(obs\_blk_i)$  and adding at its end, the following fragment:

```
and {
  SSp(obs\_blk1) ... SSp(obs\_blkn)
  CSp(obs\_blk1) ... CSp(obs\_blkn)
then free {OSp(obs\_blk1) ... OSp(obs\_blkn)
  ASp(obs\_blk1) ... ASp(obs\_blkn)}}
```

The models of the final fragment are first-order structures on a signature with as set of sorts all the sorts for labels and info from some observer block, no operations, and as predicates both those representing categories and those introduced to translate the **AFFECTS** atoms. The models of this fragment may have any

<sup>8</sup> We are using only partial functions for simplicity, but total functions are allowed as well, of course.

interpretation for the sorts and the predicates representing categories because these sorts and predicates are declared within the block with no axioms or operations. But, by definition of freeness, it has the minimal interpretation of the predicates representing the AFFECTS atoms compatible with the explicit axioms stated in the observer blocks. Since the axioms used in the free construct are all positive conditional (see e.g. [10]), this block is guaranteed to be consistent. By making the intersection of this model class with that of the expansion of the specification (which could be empty if the axioms imposed by the users are inconsistent), we get only those models where the common algebraic structure is interpreted in the same way. Thus, the label and info sorts and the category predicates satisfy the explicit axioms in the specification.

Let us see what is the expansion of our running example.

```

spec PEER [sort Address]=
sort PeerId
dsort Peer label PeerL
ops id : Peer → PeerId
      addr : Peer →? Address
preds isOnline : Peer
      goesOnline, goesOffline : PeerL
      _aff_id, _aff_addr, _aff_isOnline : PeerL
∀ l : PeerL; ∀ p, p' : Peer
  • ¬_aff_id(l) ∧ p  $\xrightarrow{l}$  p' ⇒ id(p) = id(p')
  • ¬_aff_addr(l) ∧ p  $\xrightarrow{l}$  p' ⇒ addr(p) = addr(p')
  • ¬_aff_isOnline(l) ∧ p  $\xrightarrow{l}$  p' ⇒ (isOnline(p) ⇔ isOnline(p'))
  • isOnline(p') if p  $\xrightarrow{l}$  p' ∧ goesOnline(l)
  • ¬isOnline(p') if p  $\xrightarrow{l}$  p' ∧ goesOffline(l)
  • def(addr(p)) if isOnline(p)
end
and {
sorts PeerL
preds goesOnline, goesOffline : PeerL
then free {
preds _aff_id, _aff_addr, _aff_isOnline : PeerL
∀ l : PeerL;
  • goesOnline(l) ⇒ _aff_isOnline(l)
  • goesOffline(l) ⇒ _aff_isOnline(l)
  • _aff_isOnline(l) ⇒ _aff_addr(l) } }

```

Note that atoms of the form  $(l, i)$  *affects*  $o$  are well-formed only inside the observer block(s) where  $o$  is introduced. Thus, the information about which category of actions may change the value of an observer must be collected in the same block where the observer is declared. If the same observer **obs** is declared in two or more blocks (which must refer, then, to the same dynamic sort) within the same dynamic specification, then the resulting semantics is equivalent to having the two blocks merged in one. Indeed, the minimality of the predicate *\_aff\_obs* is described by the collection of all the *free\_aff(obs\_blk<sub>i</sub>)* and hence it is immaterial in which block a category, or an axiom is given. On the contrary, if the same

observer `obs` is declared in two or more blocks in different dynamic specifications, then the first occurrence completely defines the semantics of `_aff_obs` and the further definitions are either useless or harmful, if introducing inconsistencies.

What does not need to be co-located with the observer first declaration, is the definition of the validity of the category predicates. Thus, labels and info introduced further on in the specification can affect an observer already defined.

A simple tool implementing the semantics given in Def. 4 is available on the web<sup>9</sup>. Such tool does not check syntax requirements; thus, it is only guaranteed to work on statically correct DO-CASL input. If the input specification does not contain any temporal logic formula, the result is a standard CASL specification<sup>10</sup>, which can be handled by CASL tools such as HETS<sup>11</sup>.

## 4 Writing DO-CASL Specifications

In this section we will extend the already seen PEER specification, in order to give an example on how specifications in DO-CASL can be written.

We will show a simple specification of a P2P application where nodes can send messages each other. Received messages will be accessible in an “inbox” until they are deleted. We are making use of the CASL standard library (see e.g. [1]) for the *Set* and *Map* structured data types.

```
spec BASEPEER = PEER [sort Address ]
```

The BASEPEER specification is just a shortcut. It is used in parametric specifications, which could otherwise get quite unwieldy when using a parametric specification as a parameter. We will systematically use this scheme in the following.

```
spec NET [BASEPEER ]=
  SET [sort PeerId fit Elem  $\mapsto$  PeerId ] and
  MAP [sort PeerId fit S  $\mapsto$  PeerId ] [sort Peer fit T  $\mapsto$  Peer ] and
  MAP [sort PeerId fit S  $\mapsto$  PeerId ] [sort PeerL fit T  $\mapsto$  PeerL ]
```

```
then dspec
sort Net = Map [PeerId, Peer ]; NetI = Map [PeerId, PeerL ];
observe Net info NetI
  ops dom : Net  $\rightarrow$  Set[PeerId]
```

```
end_obs
 $\forall id : PeerId; n, n' : Net; i : NetI$ 

- $dom(i) \subseteq dom(n)$  if  $i :: n \longrightarrow n'$  %% shortened notation with no label
- $lookup(id, n) \xrightarrow{lookup(id, i)} lookup(id, n')$  if  $i :: n \longrightarrow n' \wedge id \in dom(i)$
- $lookup(id, n) = lookup(id, n')$  if  $i :: n \longrightarrow n' \wedge \neg id \in dom(i)$

```

<sup>9</sup> <http://www.disi.unige.it/person/DellamicoM/do-casl>

<sup>10</sup> The tool also expands the dynamic sort declarations from observer blocks. Thus, using the equivalent form **observe** *ds* **label** *l\_ds* **info** *i\_ds* **end\_obs** to represent *ds* **label** *l\_ds* **info** *i\_ds*, it may be used also as a preprocessor for CASL-LTL, which is currently not supported at all, for specifications without temporal axioms.

<sup>11</sup> [http://www.informatik.uni-bremen.de/agbkb/forschung/formal\\_methods/CoFI/hets/](http://www.informatik.uni-bremen.de/agbkb/forschung/formal_methods/CoFI/hets/)

**end\_dspect**

**spec** BASENET = NET [BASEPEER ]

The NET specification models a *network* of peers which are part of it even if transiently offline.

Since a network is a closed system with no interactions with the outside world, its transitions are decorated with info only.

A *Net* is a mapping from peer identifiers to peers; in this way, it is guaranteed that no two nodes can have the same identifier within the same network. Analogously, *Net* transitions are mappings from node identifiers to peer transition labels, associating each moving node to the label decorating its local transition. Idle nodes do not belong to the domain of the *Net* transition.

We describe the *dom* operation on *Net* as an observer that is not influenced by any category of labels. This succinctly ensures that the identifiers represented in a network never change during the evolution of a network.

Our library handles a more general case, with nodes dynamically entering and leaving the network, by defining categories that influence the peers in the network.

**spec** MESSAGES [BASENET ]=  
SET [sortMsg fit Elem  $\mapsto$  Msg ]

**then dspect**

**observe** Peer label PeerL

**op** inbox : Peer  $\rightarrow$  Set[Msg]

**cats** receives, deletes : PeerL

$\forall l : PeerL$

- receives( $l$ )  $\Rightarrow$   $l$  affects inbox
- deletes( $l$ )  $\Rightarrow$   $l$  affects inbox

**end\_obs**

**ops** orig, dest : Msg  $\rightarrow$  Address;

sent, received : PeerL  $\rightarrow$  Set[Msg]

$\forall m : Msg; p, p' : Peer; l : PeerL; id : PeerId; n, n' : Net; i : NetI$

- received( $l$ ) = {} if  $\neg$ receives( $l$ )
- deleted( $l$ ) = {} if  $\neg$ deletes( $l$ )
- inbox( $p'$ ) = (inbox( $p$ ) - deleted( $l$ ))  $\cup$  received( $l$ ) if  $p \xrightarrow{l} p'$
- isOnline( $p$ ) if  $p \xrightarrow{l} p' \wedge isNonEmpty(received(l) \cup sent(l))$
- dest( $m$ ) = addr( $p$ ) if  $p \xrightarrow{l} p' \wedge m \in received(l)$
- orig( $m$ ) = addr( $p$ ) if  $p \xrightarrow{l} p' \wedge m \in sent(l)$
- $m \in received(lookup(id, i))$   
if  $i :: n \rightarrow n' \wedge l \in range(i) \wedge m \in sent(l) \wedge id \in dom(n)$   
 $\wedge addr(lookup(id, n)) = dest(m)$

**end\_dspect**

**spec** BASEMESSAGES = MESSAGES [BASENET ]

This specification enriches the peers with the capability of sending and receiving messages. Messages carry information about addresses of the sender and

the recipient, respectively via the *orig* and *dest* operations. Whenever a peer sends a message, the destination immediately receives it (the recipient node has to be online, otherwise the sending operation cannot be executed). The *sent* and *received* operations are meant to extract the sent and received messages in a transition.

Moreover, an *inbox* observer has been defined. Received messages remain in it until they are erased, possibly after a command from a user. While a node can only send and receive messages while it is online, it can erase messages from its own inbox at any time.

In the MESSAGES specification we have adopted a different style in order to define the value of observers w.r.t. PEER. Indeed, in PEER the observers were defined with axioms of the form  $obs(p') = \dots$  if  $p \xrightarrow{l} p' \wedge cat_i(l)$ , for each  $cat_i$  affecting  $obs$ . This style is safe, in the sense that no inconsistency with the minimality of the  $\_aff\_obs$  predicates having some  $cat_i(l)$  in the premises may be introduced, but it is quite verbose.

In MESSAGES, the *inbox* observer is specified differently. Indeed, the axiom giving its value is not guarded by a *receives*( $l$ ), nor by a *deletes*( $l$ ). Thus, it could, potentially, introduce an inconsistency. However, since *received* and *deleted* yield the empty set on label not satisfying *receives* nor *deletes*, for such labels the axiom is equivalent to stating that the value of the observer in the source and target state is unchanged, as required by the frame assumption. This style of specification is more readable and concise, though more error prone.

Both styles are convenient in different settings and DO-CASL allows to use both.

**spec** FINALMESSAGES = BASEMESSAGES  
**then**

**free generated type** *PeerL* ::=

*online*(*Address*) | *offline* | *send*(*Msg*) | *recv*(*Msg*) | *del*(*Msg*)

$\forall l : PeerL; a : Address; p, p' : Peer; m : Msg$

- $goesOnline(l) \Leftrightarrow (\exists a : Address \bullet l = online(a))$
- $addr(p') = a$  if  $p \xrightarrow{online(a)} p'$
- $goesOffline(l) \Leftrightarrow l = offline$
- $sent(send(m)) = \{m\}$
- $receives(l) \Leftrightarrow (\exists m : Msg \bullet m = recv(m))$
- $received(recv(m)) = \{m\}$
- $deletes(l) \Leftrightarrow (\exists m : Msg \bullet m = del(m))$
- $deleted(del(m)) = \{m\}$

**end**

We have concluded this section with an oversimplified example of a final definition of the transition labels. In this case, a transition can only be a simple operation like going online, offline, sending, receiving or deleting a message.

Notice that all the tedious axioms, such as  $received(offline) = \{\}$  or  $addr(p) = addr(p')$  if  $p \xrightarrow{del(m)} p'$ , are inherited from the interplay between the, so to speak, *global* statements of BASEMESSAGES and the *local* definition of the category validity.

## Conclusions and Further Work

We have introduced a specification style and some syntactic sugar supporting it in the CASL language, motivated by our experience with the definition of loose dynamic specifications.

The use of the resulting language, DO-CASL, is supported by a small prototype, compiling it in CASL, so that standard tools can be used on the resulting specifications.

Though we have fully developed the semantics of DO-CASL and, we hope, given convincing examples of DO-CASL usefulness in interesting realistic examples, the language itself has not been submitted formally as a CASL extension yet. We plan to do it if it is well received by the scientific community.

## Acknowledgements

We thank Gianna Reggio for spending her valuable time in stimulating discussions on the subject of specification of dynamic system and for her kind support.

## References

1. CoFI (The Common Framework Initiative): CASL Reference Manual. LNCS2960 (IFIP Series). Springer Verlag (2004)
2. Schröder, L., Mossakowski, T.: HASCASL: Towards integrated specification and development of Haskell programs. In Kirchner, H., Ringeissen, C., eds.: AMAST 2002 Proceedings. LNCS2422, Springer Verlag (2002) 99–116
3. Baumeister, H., Zamulin, A.V.: State-based extension of CASL. In Grieskamp, W., Santen, T., Stoddart, B., eds.: IFM 2000, Dagstuhl Castle, Germany, Proceedings. LNCS1945, Springer Verlag (2000) 3–24
4. Reggio, G., Astesiano, E., Choppy, C.: CASL-LTL: A CASL extension for dynamic reactive systems – Version 1.0 – Summary. Technical Report DISI-TR-03-36, Univ. of Genova (2003)
5. Costa, G., Reggio, G.: Specification of abstract dynamic-data types: A temporal logic approach. TCS **173**(2) (1997) 513–554
6. Reichel, H.: Behavioural equivalence — a unifying concept for initial and final specification methods. In: Proc. 3rd. Hungarian CS Conference. (1981) 27–39
7. Cerioli, M.: Basic concepts. In: Algebraic System Specification and Development: Survey and Annotated Bibliography. 2nd edition, 1997. Number 3 in Monographs of the Bremen Institute of Safe Systems. Shaker (1998)
8. Borgida, A., Mylopoulos, J., Reiter, R.: On the frame problem in procedure specifications. IEEE Trans. Softw. Eng. **21**(10) (1995) 785–798
9. Choppy, C., Reggio, G.: Using CASL to specify the requirements and the design: A problem specific approach. In Bert, D., Choppy, C., Mosses, P.D., eds.: Recent Trends in Algebraic Development Techniques, 14th International Workshop, WADT’99, Château de Bonas, France, 1999, Selected Papers. LNCS 1827, Springer-Verlag (2000) 104–123
10. Cerioli, M., Mossakowski, T., Reichel, H.: From total equational to partial first-order logic. In Astesiano, E., Kreowski, H.J., Krieg-Brückner, B., eds.: Algebraic Foundations of Systems Specification. IFIP State-of-the-Art Reports. Springer-Verlag (1999)