DynOCoLa: Enabling Dynamic Composition of Object Behaviour

Chitra Babu, Wenonah Jaques and D Janakiram
Distributed Object Systems Lab, Dept. of Computer Science & Engg.,
Indian Institute of Technology Madras,
Chennai - 600 036, India.
Email: {chitra, wenonah, d.janakiram}@cs.iitm.ernet.in
URL: http://lotus.iitm.ac.in

Abstract

Aspect Oriented Programming (AOP) has emerged as a promising paradigm for modularly capturing systemic cross-cutting concerns in modern software applications. This approach has enhanced separation of concerns and has minimized code tangling to a large extent. However, while developing applications, in order to characterize the dynamic and evolving object behaviour effectively, a single object itself may have to be viewed as a composition of several aspects. Towards this objective, this paper proposes a **Dyn***amic* **O***bject* **Co***mposition* **La***nguage (***DynOCoLa***), which views object composition itself as a combination of various aspects that abstract the variant behaviour of objects. This behavioural variation is based on the contractual conditions that are fulfilled in different contexts. Instead of viewing objects as being rigidly defined during compile-time, the DynOCoLa run-time system weaves the various aspects appropriately with the objects. Thus, DynOCoLa enables dynamic customization of an object's behaviour based on the state of the object. This paper discusses the design and implementation of DynOCoLa. An application scenario, involving customized service delivery to mobile devices based on the location of the devices, has been studied to illustrate the power and flexibility offered by DynOCoLa.*

Keywords: Dynamic Behavioural Composition, Dynamic Object Behavioural (DOB) Aspect, Aspect Oriented Programming (AOP), State Based Filtering(SBF), Surrogate Object

I. Introduction

Separation of concerns is a highly desirable goal in large-scale software development. Aspect Oriented Programming (AOP)[1] paradigm has facilitated better separation of concerns by providing a cleaner abstraction for capturing concerns which cut across different classes. This has minimized code tangling and has enabled easy code maintenance. However, presently the focus of AOP is confined to system-level concerns such as logging, synchronization and persistence.

Viewed from an alternative perspective, the behaviour of an individual object itself can be composed out of multiple aspects specific to an object which abstract the customizable behaviour of that object. These aspects have been termed as Dynamic Object Behavioural(DOB) aspects. The DOB aspects encompass the variant behaviour, the contractual conditions that capture the corresponding state of the object and the appropriate *self* rebinding. For modeling the evolving object behaviour, the strong coupling between abstraction and encapsulation, which is characteristic of traditional class-based OO programming languages, need to be relaxed in a systematic type-safe way. Encapsulation should be maintained among a group of fragmented objects which are logically part of a single object, projecting a unique identity. This is the key design philosophy behind the proposed language DynOCoLa which is based on the Method Driven Model (MDM)[2]. Such an approach enables dynamic customization of object behaviour depending upon its internal state. Further it provides a way of rebinding the *self* pointer appropriately whenever an object extends its behaviour during run-time.

DynOCoLa facilitates filtering of messages received by objects based on the current state of the object. Although there are a few filter models[3], [4] currently available, they do not semantically ensure whether the object is in a suitable state before executing the corresponding filtering actions. Thus, DynOCoLa is unique in providing State Based Filtering (SBF) of messages. It also facilitates introduction of new DOB aspects during run-time for any unanticipated condition that needs to be dealt with. This

is crucial for handling several issues that arise in dynamically configurable systems. A sample scenario where the flexibility offered by DynOCoLa can be beneficial is in providing customized service delivery to mobile devices based on the location of the devices.

The rest of the paper is organized as follows. Section 2 briefly explains the language, DynOCoLa. It also elaborates on the design and implementation of the compiler and run-time system for DynOCoLa. A case study that illustrates the effectiveness of DynOCoLa is examined in Section 3. Section 4 discusses related work while Section 5 concludes and provides future research directions.

II. The Dynamic Object Compositional Language

This section describes the **Dyn**amic **O**bject **C**ompositional **L**anguage - **DynOCoLa**. DynOCoLa is a realization of the Method Driven Model. MDM allows the definition of a set of objects whose behaviour can change dynamically based on their state. The state of the object is dictated by the member variables encapsulated within the object. These objects are instantiated from *partial class* definitions. The variable behaviour is abstracted into first-class entities known as *DOB aspects*. At runtime, depending on the state of the object, an appropriate DOB aspect is weaved with the object.

DynOCoLa has been developed in Java. Each of the objects exhibiting variable behaviour is defined as a set comprising of a partial class definition and the definition of the set of aspects associated with the partial class. The DynOCoLa compiler converts this set of definitions into java class definitions. The aspect runtime system of DynOCoLa is responsible for suitably weaving the DOB aspects with the objects at runtime.

Each partial class is defined with the *.apc* extension. It is similar in definition to a java class definition. The class can be part of a package declaration and can import any number of packages. The class definition has an additional keyword *partial* which is specific to DynOCoLa. Similar to any class definition in java, the partial class consists of member variable and member function declarations. Apart from these, it also includes some additional declarations specific to DynOCoLa.

The *composable-aspects* definition specifies the set of DOB aspects which are associated with the partial class. The *aspect-view* defines the visibility of the member variables to the DOB aspects. The current implementation of the compiler allows the aspects to access *all* of these member variables.

The *TypeMarker(TM)* is an abstraction of the set of related methods exhibiting variable behaviour. These methods are related in terms of the member variables which dictate the weaving of DOB aspects. A set of methods which are governed by the same set of variables can be within a single TM definition. Each partial class can comprise of one or more TM definitions. A TM definition starts with the keyword TM, followed by the name of the TM. The TM definition in turn comprises of one or more TM method declarations. The definition of the TM method is similar to that of any method in java. However, it has an additional optional *plug* section. This section is essentially for reflecting the input parameters of the method into the member variables of the class. This *plug* section is required if the weaving of aspects is dictated by the input parameters.

If the partial class corresponding to this aspect is part of a package declaration, the aspects corresponding to the class should also be part of the same package. An additional keyword *aspect* signifies the definition of the DOB aspect. Each aspect definition comprises of four main sections:

- The *unit-type* section declares the partial class to which the DOB aspect belongs. It also defines the specific category of the TM("in" or "partof") to which this aspect is associated.
- The *unit-encap* section encapsulates the definitions of the corresponding TM methods of the partial class which are implemented by the DOB aspect. The method definitions are preceded by the *replace, before* or *after* keyword. The *replace* keyword is used for aspect definitions which are of type "part-of". The *before* and *after* keywords are used for "in" type aspects which behave as filters. The signature of these methods should match the TM methods they represent in the corresponding partial class.
- The *unit-contract* section of the aspect defines the condition necessary for the aspect to be weaved. This condition will be in terms of the member variables of the partial class corresponding to the aspect. The condition statement is preceded by the keyword *require*.
- The *unit-identity* section is not always mandatory. It is required only if the DOB aspect refers to any member variable or function from its corresponding partial class. In that case, the *self* pointer is properly rebound to the object instantiated from the partial class.

The next section discusses in detail the design issues and implementation of the compiler and runtime system of DynOCoLa.

A. Design and Implementation

DynOCoLa enables programmers to develop applications which comprise of objects that exhibit behavioural changes. It has been built using the Java

programming language. It provides language level abstractions namely partial classes and DOB aspects to programmers for defining customizable objects and their behavioural changes. The DynOCoLa compiler functions as a pre-processor which converts the application code written using partial classes and DOB aspects into plain java code. These Java programs are finally interpreted by the Java Virtual Machine (JVM). During execution of the application, a separate instance of an aspect runtime system is created, for each object instantiated from a partial class. This is because, the DOB aspects are specific to each object. The aspect runtime system is responsible for weaving the appropriate aspect based on the state of the object.

1) The DynOCoLa Compiler: The compiler for DynOCoLa has been built using the automatic compiler construction tools that are specific to the Java language, JFLEX[5] and CUP[6].

Jflex is a scanner generator which will tokenize the keywords of the language. CUP is a LALR parser generator tool, which generates a parser that verifies the grammar for the language constructs. The grammar for DynOCoLa is presented elsewhere[7].

The compiler takes as its input the set of partial class files *(.apc)* and the DOB aspect files *(.as)*, and generates a set of .java files. Besides checking the input files for syntactic errors, it also does type checking by matching the signatures of the TM methods of the DOB aspects with the corresponding methods in the partial class. Any violation in type checking generates an error, and terminates the file generation process.

The compiler also generates a .java file which defines the class encapsulating the state variables of the partial class. The state object instantiated from this class is used for determining the appropriate DOB aspect to be weaved at runtime.

One of the important features of the DynOCoLa language is that it allows the late definition of DOB aspects. This is essential because whenever the behaviour of an object evolves in an unanticipated manner, it needs introduction of a new DOB aspect. The newly added DOB aspects can be compiled along with the partial class without having to recompile the other DOB aspects.

2) Aspect Runtime: At runtime, the user instantiates an object from the partial class depending upon his requirements. Each partial class is associated with an aspect runtime. The aspect runtime is responsible for managing the DOB aspects corresponding to the partial class. Whenever a call is made to a TM method of the partial class, the following steps are executed at the partial class and the aspect runtime.

- Some of the parameters passed to the TM method may have an effect on the state of the object. In such cases, the values of these parameters have to be reflected into the state variables before the DOB aspect is weaved. This is done within the 'plug' section.
- The next step is to weave the appropriate DOB aspect based on the state of the object. The process of weaving is performed by the aspect runtime.
  - A state object is created by the aspect runtime. This state object represents the values of the state variables at that instance of time.
  - The state object is then sequentially passed to every DOB aspect for the evaluation of the expression in the contract section to verify whether the aspect needs to be weaved. Since this process is sequential, the first DOB aspect whose contract section is satisfied, is chosen as the DOB aspect to be weaved. Currently it has been assumed that the state space covered by the rules specified in alternative DOB aspects corresponding to a given partial class is disjoint. This implies that at any time, no more than one rule in the set of alternatives will simultaneously hold. However, if none of the contract sections of the DOB aspects are satisfied, the aspect runtime raises an exception.
  - Once the DOB aspect to be weaved is determined, the aspect runtime creates a new instance of the DOB aspect to be weaved. In case of stateful aspects, the aspect runtime returns a previously created instance of the DOB aspect.
  - Depending on the type of the DOB aspect i.e. 'part-of' or 'in', the appropriate methods defined in the 'unit-encap' section of the DOB aspect (replace/before and after) are invoked from the partial class. The sequence in which the DOB aspect methods have to be invoked is specified in each TM method of the partial class during the time of compilation. Once the appropriate DOB aspect is selected, its type is checked. Based on its type, the appropriate methods of the DOB aspect are executed. For example, in case of 'part-of', the *replace* method specified in the DOB aspect is executed. In case of 'in', the *before* filter method of the DOB aspect is executed followed by the code of the partial class and the *after* filter method.

Two different strategies can be followed while deciding the time at which the DOB aspects should be weaved with the object.

- One way is to capture all state variable changes and convert them into set() method invocations during compile time. The runtime system can set a flag indicating the state variable in which a change has

occurred and accordingly only those rules involving these variables are checked in deciding the DOB aspects to be weaved. Once the appropriate DOB aspect is chosen, it can be weaved along with the object immediately without waiting for the TM method call. However this strategy incurs a lot of unnecessary overhead because it is possible that after a particular aspect gets weaved in, the state variables change again before a TM method invocation happens. Hence, this approach has not been adopted in the current implementation.

- Another approach which is favoured by the current implementation is to evaluate the rules only when a TM method is invoked. The current state of the object is constructed and is compared against the previous state of that object. If both of these are identical, the TM method invocation happens on the currently weaved aspect. If the two states differ, then the corresponding aspect to be weaved for the modified state is determined. The existing aspect is unweaved, the new aspect is weaved and the TM method is invoked on that aspect.

III. Case Study

DynOCoLa is unique in that it inherently facilitates behavioural modification of objects based on their state. This section illustrates this feature of DynOCoLa through an application scenario where service delivery to mobile devices can be customized based on the location of the devices.

A. Surrogate Object Model

The Surrogate Object Model (SOM) proposed in [8] models each mobile device in a cellular network as an object. This object, termed as the Surrogate Object(SO), resides at the Mobile Service Station(MSS). It maintains the location of the mobile device along with other relevant information resident in the mobile device. The mobile devices modelled by the SO are constrained in terms of battery, computational power and bandwidth. Due to these constraints as well as the mobility factor itself, adaptation of mobile systems has become extremely important[9].

Mobile users currently access a number of web services through their mobile devices as they travel to various locations. These services may involve complex tasks like on-line reservations, or even simple SMS messages delivered by mobile service providers to users in roaming mode. At different locations, this information provided may be in the local language. However, unless the user is able to understand the language in which he receives the information, it would be useless. This section discusses the customization of the SO to deliver the information to the mobile users in a language understood by him.

1) Location Based Customization of Services: The SO is the representation of the mobile device at the MSS. Messages are delivered to the mobile hosts through their corresponding surrogate objects. The SO, therefore has to be customized based on its state at the time of the service delivery. In this case, the state of the SO is actually the location of the mobile device which is maintained by the SO. Further, the SO can also maintain other information such as the home network of the user, type of mobile device etc. The customization is the association of appropriate filters with the object, which translate text messages or web based information into a language understandable by the mobile user.

The blueprint of SO is defined as a *partial class*. The set of adaptable methods are defined within a *TM*, *DeliverData*. There are two *TM methods*, associated with delivery of text messages(SMS) and web based information respectively to the mobile devices. Each of these methods have language translation filters associated with them. These filters translate the information from the language in which the data is delivered to the local language of the user. The appropriate DOB aspect is selected based on the location of the user at the time of information delivery. For example, consider a person of Indian origin travelling to Japan. In such a scenario, the information in the form of SMS alerts from cellular providers in Japan, or web pages, is most likely to be in Japanese. At this point in time, the state information in the SO would indicate that the user is in Japan. Therefore whenever a TM method is invoked the DOB aspect with the contract condition

```
location.equals("Japan");
```

will be invoked. This DOB aspect will translate the data into English, which would then be delivered to the user. As stated previously, two separate TM methods are used for delivery of web based information and SMS data respectively. The motivation for this design decision is that the complexity associated with the delivery and translation of the data in these cases is different. It is assumed here that appropriate algorithms are available for the translation. Figure 1 illustrates the design of the SO.

2) Additional Enhancements : The previous section discussed a simple form of customization of the SO. This involved association of appropriate language translation filters to the SO based on the location of the mobile device. Another major issue in providing services to mobile users is the limited display and networking capacity of these devices[10]. Therefore, the SO can be customized to render information to devices not only based on their location, but also the device types and network parameters. The additional state information which would dictate the be-

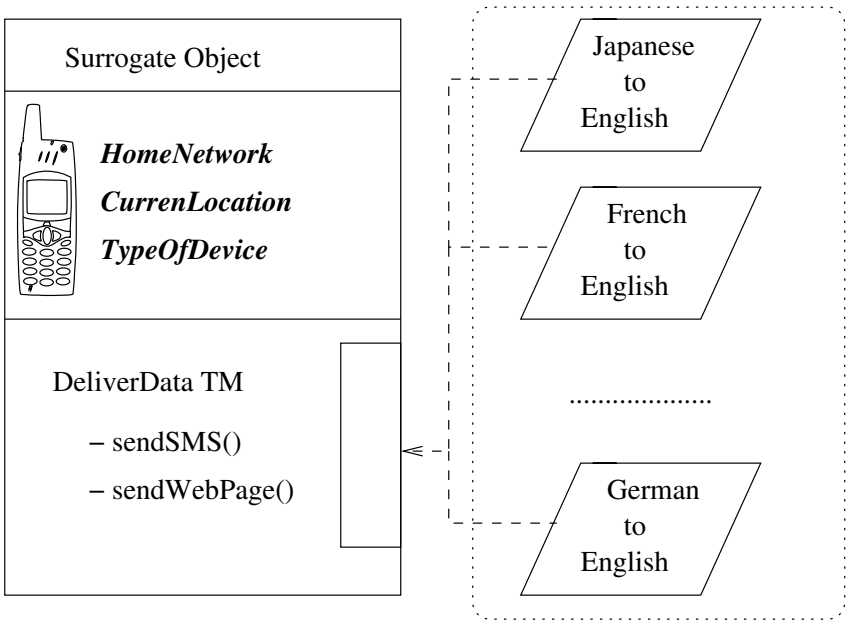

Fig. 1. Design of the Surrogate object

haviour of the object would be the device type and network parameters.

Consider a mobile user accessing services either through a laptop or his mobile phone. In case his mobile phone is not Wireless Access Protocol(WAP) enabled, he will not be able to obtain web information directly. In such a case, the filter at the SO level would have to filter out the relevant information from the web pages. Only this relevant information can then be communicated to the user in the form of plain text messages. Further, sometimes the network bandwidth available to a user's mobile device may be limited. In such scenarios it may be more appropriate to render the web pages to a user's laptop without the images/multimedia content, and with only the required amount of relevant information. In such a case too, the filters associated with the SO will be designed to filter out the irrelevant information. The state variables which dictate the appropriate filters to be weaved will be the device type and network bandwidth, in addition to the location of the device. The device type associated with a SO will always remain the same, since the SO will always be representing a single individual entity from the time it joins the cellular network. The network bandwidth however, may keep varying. The underlying infrastructure used to capture this parameter is assumed, and is beyond the scope of this paper.

A further enhancement to the type of customizations possible for the SO is, to filter the messages that should be delivered to the mobile device. The delivery of the messages would be dictated by the location of the mobile device and the role of the entity sending the message. For example, when a mobile user is in the roaming mode, he may want to receive only important messages on his mobile phone. The DOB aspects associated with the SO would then have to filter out the relevant messages and forward them. The other messages can be stored by the SO, to be forwarded to the user when he is back within his home network. The parameters which would govern the actions taken when messages arrive in this case would include the location and the role of the entity from which messages arrive. Similarly the type of services that the user would want to access from a particular location can also be specified.

In each of the above scenarios, for each mobile user, it would be very difficult to anticipate at the beginning itself all the possible scenarios for which the filters have to be applied. For example, it may not be possible to anticipate all the locations to which a user may travel to, the services he may use, or even the persons who send him messages. Besides, the preferences of the user may keep changing over a given time. DynOCoLa can accommodate all these changing requirements of the user. Unanticipated behaviour of an object can be accommodated by adding a new set of aspects encapsulating the new behaviour.

IV. Related Work

Kiczales et al. proposed the language AspectJ[11] which adheres to the paradigm of AOP. This language captures only global concerns that cut across the boundaries

of various classes. The aspect weaver weaves the aspects at the specified join points in the class during compile-time. On the other hand, the key focus of DynOCoLa is to compose the behaviour of objects dynamically based on their internal state captured through contracts.

Hyper/J[12] is a language proposed by Peri Tarr et al. based on the Multi Dimensional Separation Of Concerns (MDSOC) model. This language considers composing a class out of multiple dimensions. However, the class composition happens at compile time. Demeter/Java is a language developed by Lieberherr et al. based on the Adaptive Programming paradigm[13]. This is analogous to AspectJ except for the fact that the class structure is captured in the form of a graph and traversal strategies specify the join points. However, both these languages do not enable dynamic composition of object behaviour based on its state.

Sina[14] is a language designed based on the Composition Filters (CF) object model[4]. Sina language provides input and output composition filters that intercept the received and sent messages respectively. However these filters are specified as part of the class itself. Hence, introducing filters for unanticipated conditions during run-time is not possible in this model as opposed to DynOCoLa.

V. Conclusions

This paper presented the language DynOCoLa, whose objective is to dynamically compose the overall behaviour of an object from the various DOB aspects that abstract the customizable context-dependent behaviour of that object. This approach facilitates dynamic modification of object behaviour based on the current state of the object. These DOB aspects also provide a way of properly rebinding *self* whenever an object extends its behaviour during run-time. A sample application scenario that involves customized service delivery to mobile devices based on their location has been studied. It is evident from the case study that DynOCoLa can be effectively used for developing applications that mandate dynamic state-based changes in the behaviour of objects. The language constructs provided by DynOCoLa enable run-time customization of object behaviour while developing applications for pervasive computing environments[15].

In the current implementation of the language, the state object captures the values of all the variables that belong to a given partial class. The language can be modified to allow the programmer to specify a relevant subset of variables which should constitute the state object. Extending the language to support the maintenance of the state within the aspect can also be explored.

References

[1] G. Kiczales, J. Lamping, A. Mendheker, C.Maeda, C. Lopes, J. M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of ECOOP 1997*, pages 220–242. Jyvaskyla, Finland, June 1997.

[2] Chitra Babu and D. Janakiram. Method Driven Model: A Unified Model for an Object Composition Language. *ACM SIGPLAN Notices*, 39(8):61–71, August 2004.

[3] R. K. Joshi and D. Janaki Ram. Message Filters for Object-Oriented Systems. *Software-Practice and Experience*, 27(6):678–699, June 1997.

[4] M. Aksit, K. Wakita, J. Bosch, L. Bergmans, and A. Yonezawa. Abstracting Object Interactions Using Composition Filters. In R. Guerraoui, O. Nierstrasz, and M. Riveill, editors, *Proceedings of the ECOOP Workshop on Object-based Distributed Programming*, pages 152–184. LNCS, Springer-Verlag, Kaiserslautern, Germany, 1993.

[5] G. Klein. Jflex users manual. Version 1.4, April 2004.

[6] S. E. Hudson. Cup users manual. Graphics Visualization and Usability Center, Georgia Institute of Technology, July 1999.

[7] Chitra Babu, Wenonah Jaques, and D. Janakiram. DynOCoLa: Dynamic Composition of Object Behaviour Through Aspects . Technical Report IITM-CSE-DOS-04-10, Indian Institute of Technology, Madras, India, 2004.

[8] D. Janakiram, M. A. M. Mohamed, and M. Chakraborty. Surrogate Object Model: A New Paradigm for Distributed Mobile Systems. In *Proceedings of the Fourth International Conference on Information Systems Technology and its Applications(ISTA)*. Massey University, New Zealand, 2005.

[9] Jin Jing, A. Helal, and A. Elmagarmid. Client-Server Computing in Mobile Environments. *ACM Computing Surveys*, 31(2), 1999.

[10] S. Poslad, H. Laamanen, R. Malaka, A. Nick, and A. Zipf. CRUMPET: Creation of User-friendly Mobile Services Personalised For Tourism. In *Proceedings of the 2nd International Conference on 3G Mobile Communication Technologies, London, UK*, 2001.

[11] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W.G. Griswold. An Overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP 2001)*, pages 327–353, 2001.

[12] Hyper/j. http://research.ibm.com/hyperspace/ HyperJ/HyperJ.htm.

[13] K. J. Lieberherr, I. Silva-Lepe, and C. Xiao. Adaptive Object-oriented Programming Using Graph-based Customization. *Communications of the ACM*, 37(5):94–101, May 1994.

[14] M. Aksit. *On the Design of the Object-Oriented Language Sina*. PhD thesis, University of Twente, The Netherlands, 1989.

[15] Chitra Babu, Wenonah Jaques, and D. Janaki Ram. Dynamic Customization in Pervasive Environments. In *Proceedings of the IEEE Conference on Enabling Technologies for Smart Appliances (ETSA) 2005, Hyderabad, INDIA*, January 2005.