A Graph-based Operational Semantics for Context-oriented Programming

Hans Schippers, Tim Molderez and Dirk Janssens
Dept. of Mathematics and Computer Science
University of Antwerp
Antwerp, Belgium
{hans.schippers,tim.molderez,dirk.janssens}@ua.ac.be

ABSTRACT

Context-oriented programming can be regarded as a technique aiming for an improved (multi-dimensional) separation of concerns (MDSOC). The delMDSOC (virtual) machine model describes a common target platform for a range of high-level MDSOC approaches. As it is based only on the well-known concepts of objects, messages and delegation, it provides a means to express the semantics of context-oriented programming using these same notions as well. An approach based on structured operational semantics (SOS) exists, but it has a number of drawbacks, including its implicit representation of program state and the lack of a simulation mechanism. In this paper we introduce a graph-based semantics for context-oriented programming built on top of a concurrent actor-based semantics of delMDSOC. The semantics consists of a number of graph rewrite rules which can be applied to sample graphs using the AGG tool. This allows for the visual simulation of context-oriented programs, which we demonstrate by means of an example.

1. INTRODUCTION

Context-oriented programming (COP) is a recent technique aiming at improving (multi-dimensional) separation of concerns (MDSOC) by organizing code in so-called layers which can be activated and deactivated in dynamic scopes. A couple of attempts have been undertaken at the specification of a formal operational semantics for COP, typically using a structured operational semantics (SOS) approach [2, 7]. An SOS semantics provides a set of syntax-based inference rules for steps between program states; the states are usually expressed by the values of a set of variables, meaning that relationships between these values are not explicitly visible. Moreover, tool support is rather limited, meaning that SOS specifications are often only manually verified and thus prone to errors.

Graph transformation [5] techniques take a different approach at the specification of formal semantics. In a graph transformation system, program states are graphs, enabling one to express the various relationships between basic entities such as objects and messages explicitly by means of labeled edges between the nodes representing them. These graphs can be transformed by means of so-called graph rewrite rules. Hence, by representing program states as graphs, it becomes possible to formally express the operational semantics of an arbitrary program through a set of graph rewrite rules.

Graph transformation techniques have been used in an MDSOC context before [1], in order to describe a formal operational semantics for an aspect-oriented extension to featherweight Java. Apart from exhibiting an intuitive syntax in which all entities involved are represented explicitly (i.e., run-time state is represented as a graph), the formalism is also useful in the context of verification algorithms, since the state space consists of a collection of graphs which could all be checked for certain properties.

The approach in [1] is specifically tailored towards the pointcut-and-advice flavor of aspect-oriented programming in that it introduces dedicated concepts such as a proceed stack. However, the benefits of a graph-based approach are relevant for the MDSOC paradigm in general, including context-oriented programming. The delMDSOC machine model [3, 7] describes a target platform for a range of MDSOC techniques by means of a very limited set of concepts: objects, message sending and delegation. Possibly dynamic (un)deployment of functionality, be it in the form of aspects, layers or other modularization mechanisms, is expressed by allowing objects to delegate to other objects, thus forming chains of delegating objects, and dynamic manipulation of these delegation chains. By expressing the semantics of high-level MDSOC approaches in terms of these common concepts, it may become possible to compare and combine or even translate e.g., an aspect-oriented program into a context-oriented one and vice versa.

This paper makes a contribution towards this goal by specifying a formal operational semantics for context-oriented programming on top of the delMDSOC machine model. It is organized as follows: Section 2 briefly reviews the essentials of delMDSOC, followed by an overview of its graph-based semantics in Section 3. Next, we present the graph-based operational semantics for COP in Section 4. Finally, Section 5 summarizes the paper.

2. A MACHINE MODEL FOR MDSOC

In this section we briefly describe the delMDSOC machine model originally introduced in [3].

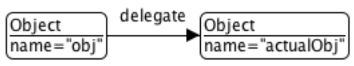

Figure 1: delMDSOC object representation.

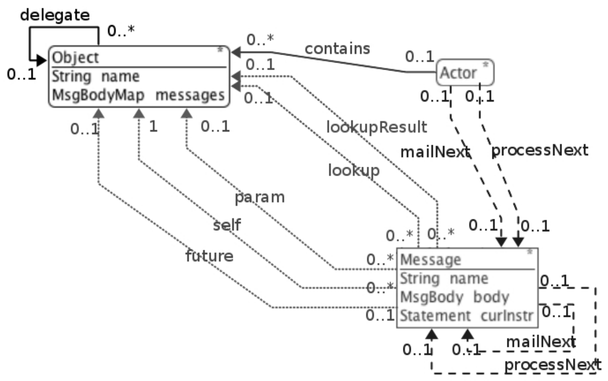

Figure 2: delMDSOC type graph.

The model assumes a prototype-based object-oriented environment in which each object is visible to others via a *proxy*, which is a placeholder determining the object's identity. Messages sent to an object are received by its proxy and are *delegated* to the actual object, as displayed in Figure 1. It is important to realize that the actual object is not necessarily fixed, and may depend on information such as the receiver or sender of a message, the invoked method, etc. In general, the *delegate* arrow should be considered a function of arbitrary contextual information rather than a pointer. However, in this paper the pointer interpretation is sufficient.

The model allows dynamic modification of the initial configuration by inserting and removing additional proxy objects in between the proxy and its delegate object. This results in a chain of proxy objects organized in a delegation chain, which collectively constitute the whole object, also called the *composite object*. A crucial property of delegation is that *self* remains bound to the original receiver object, i. e., the proxy at the head of the delegation chain.

3. A GRAPH-BASED OPERATIONAL SEMANTICS FOR delMDSOC

A summary of a graph-based semantics for the delMDSOC machine model was first described in [4]. It consists of a number of graph rewrite rules, specified using the AGG tool [8], which encode the possible transitions of program state, assuming the latter is represented as a graph. Such graphs consist of a number of attributed nodes and edges. Figure 2 shows the corresponding *type graph*, which lists the different kinds of nodes which may occur, as well as the edges potentially connecting them.

Actor nodes are concurrent entities and basically act as containers of *Object* nodes. Each actor has a mail queue where it stores all *Messages* sent by objects contained within other actors. An actor processes the messages from its mail queue sequentially, by removing them from the mail queue and pushing them onto the process stack. By definition, a message on the process stack must always target an object contained by the corresponding actor. The message is connected to this object by means of a *self* edge. In delMDSOC, the *self* object usually is a proxy, which by definition does not understand any messages. Hence, a lookup algorithm determines which object in *self*'s delegation chain holds an appropriate message implementation in its *messages* attribute, which acts as method dictionary. This object is marked by a *lookupResult* edge. The looked up message implementation is then copied to the *body* attribute of the message node. It consists of a list of instructions, which determine the message's effect. Some instructions may cause new messages to be sent, which is reflected by the creation of new *Message* nodes. If such a message targets an object within the same actor, it is pushed on the process stack. If it targets an object within a different actor, it ends up in that actor's mail queue. In the context of this paper, we only consider single-actor scenario's, while we refer to [6] for more information on actor-based concurrency in delMDSOC.

So-called graph rewrite rules describe how a graph may evolve, and hence capture the operational semantics of delMDSOC. They take a graph representing a certain initial program state as input, and transform this graph until it reflects the program state at the end of the calculation which the program represents. In general, a graph rewrite rule consists of a *left hand side* (LHS) and a *right hand side* (RHS). The LHS describes a graph pattern, a (small) graph which, if it is present as a subgraph of the current host graph, is replaced by the subgraph described by the RHS. Potentially, additional constraints as to when a rule should or should not be applicable, may be specified as *attribute conditions* (AC), or *negative application conditions* (NAC). A NAC is a graph pattern which must not be present in the host graph for the rule to be deemed applicable. AC's are boolean expressions which may appear in order to express constraints on attribute values.

For example, Figure 3 shows the graph rewrite rule which removes the first message from the mail queue and pushes it onto the process stack. To the right of the dotted line is a NAC, ensuring that the rule will not be applied in case a message is already present on the process stack. To the left of the dotted line are the LHS and RHS of the rule, separated by a thick black arrow. The pattern specified in this particular LHS will be matched in the host graph if at least two messages are present in the mail queue[1]. Since the *self* object (labeled by the number prefix 3 in Figure 3) of the first message is also relevant for transforming LHS into RHS, it is also present in the LHS. However, its presence does not further constrain the match as, by construction, a message is always connected to a self object. This holds for the message labeled 4 as well, but since its self object is not involved in the transformation, there is no need for it to be specified. The labels prefixing the nodes and edges are used for identification purposes, meaning a node (or edge) bearing the same label in LHS and RHS (and possibly NAC) refers to the same node (or edge) in the host graph. Nodes (or edges) appearing on the LHS but not on the RHS will be deleted, while the inverse situation results in creation of new nodes or edges.

In this particular RHS, several changes compared to the LHS can be perceived. For example, the two *mailNext* edges

[1]The case where only one message is present in the mail queue is handled by a variation of this rule, and the whole is actually specified in AGG as a so-called *amalgamated rule.*

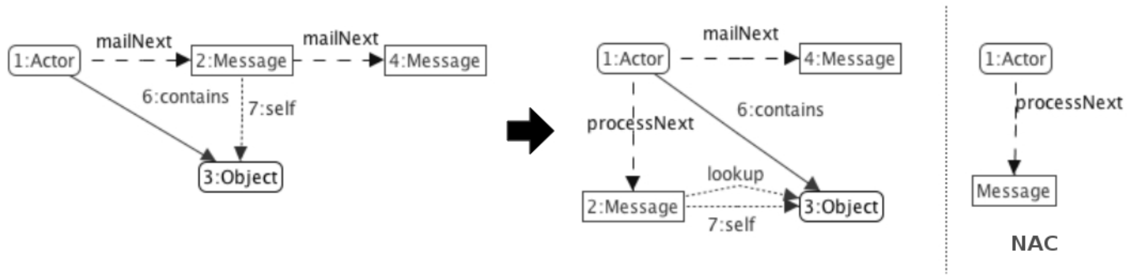

Figure 3: Process message rewrite rule.

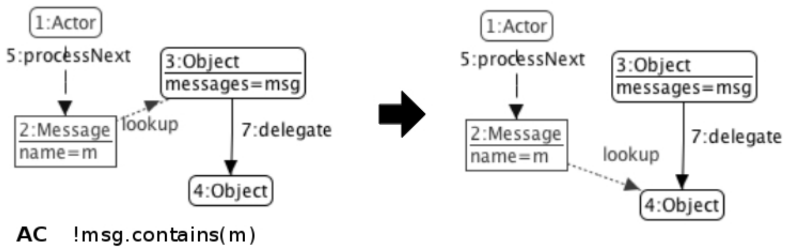

Figure 4: Rewrite rule traversing a delegation chain during message lookup.

storing messages 2 and 4 in the actor's mail queue, are removed and replaced by one *mailNext* edge connecting the actor to message 4, effectively removing message 2 from the mail queue. Instead, message 2 is pushed onto the actor's process stack (which is guaranteed to be empty by the NAC) by means of a *processNext* edge. At the same time, a *lookup* edge is created between message 2 and its self object, indicating that the lookup process can be initiated and should start in the self object. The lookup process is realized by the graph rewrite rule in Figure 4, which is only applicable in case a *lookup* edge is present for the message at the top of the process stack, as can be derived from its LHS. Additionally, an attribute condition (AC) is present, which enforces the additional constraint on the LHS that *msg*, which is a local variable holding the current value of the *messages* attribute of the object labeled 3, does not contain a definition for the name *m* of the message currently at the top of the process stack. If the LHS can be matched in the host graph, it is replaced by the RHS, meaning that the *lookup* edge is removed and replaced by a new *lookup* edge, this time connecting the message and the object which follows object 3 in the latter's delegation chain. The rule may now be applicable again, if object 4 does not contain a message implementation for *m* either, etc. The intuition behind this rule is that, by repetitive application, it looks up a method implementation in the method dictionary of an object and its delegates. As soon as an implementation is found, the AC will fail, and the rule will no longer apply. In this situation, the LHS of the rule in Figure 5 will match, and replace the *lookup* edge by a *lookupResult* edge in order to denote the end of the search. Furthermore, it extracts the message implementation from the message dictionary, and stores it in the *body* attribute of the message.

Note that several expressions, including AC's and attribute initializers exhibit a Java-like syntax. This is because node attribute types are Java types in AGG. It is important to realize that this escape to Java does not imply that the formal character of the whole approach is tainted. Technically, all datatypes could be represented as graphs too, rendering the use of attributes unnecessary. However, this would impair the readability of the graphs. Hence, Java datatypes are borrowed in order to provide a more concise notation for data storage.

A message implementation consists of a list of textual instructions. Once a message has been associated with such implementation, these instructions are processed one by one by being moved from the *body* attribute of the message to its *curinstr* field. Several rewrite rules, essentially one for each type of instruction, may now apply. We will now look at the SEND and DEPLOY instructions in more detail. The LHS of the rewrite rule in Figure 6 includes an AC which constrains the *curinstr* to be of the form

```
SEND(msgName, receiverName)
```

Moreover, the LHS looks for an object (labeled 4) with a *name* attribute equal to receiverName. The RHS causes this object to be employed as the self object of a newly created message node which is pushed on the process stack (and will hence be processed before continuing the current message).

Analogously to handling SEND, the AC in the LHS of the rewrite rule in Figure 7 recognizes instructions of the form

```
DEPLOY(proxyName, targetName)
```

and the corresponding objects are matched accordingly. The RHS causes the object denoted as proxy to be inserted in the delegation chain of the target object, immediately after the target object itself.

Undeploying a proxy from a delegation chain is achieved similarly, by means of an UNDEPLOY instruction and a corresponding rewrite rule.

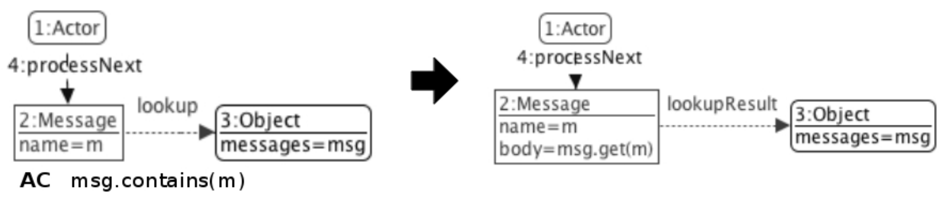

Figure 5: Rewrite rule extracting a message implementation after message lookup.

4. A GRAPH-BASED SEMANTICS FOR CONTEXT-ORIENTED PROGRAMMING

In [7], a delegation-based operational semantics for dynamically scoped layer activation was provided by means of an SOS approach. For instance, the semantics for the evaluation of an expression e surrounded by a *withlayer* block activating layer L is described by the following reduction rule:

$$\frac{\begin{array}{rl} & getMethods(P,L) = M_1 \dots M_n \\ & \forall i : M_i = T_i\ C_i.m_i(T_i'\ x)\{e_i\} \\ & \forall k \in [1,n] : \iota_{lp,k} \notin dom(h) \\ h' = & h[\iota_{lp,1} \mapsto [\![m_1 : e_1]\!]][Del_{\iota_{lp,1}} \mapsto Del_{h(C_1)}] \\ & \qquad [Del_{h(C_1)} \mapsto \iota_{lp,1}] \\ & \dots \\ & [\iota_{lp,n} \mapsto [\![m_n : e_n]\!]][Del_{\iota_{lp,n}} \mapsto Del_{h(C_n)}] \\ & \qquad [Del_{h(C_n)} \mapsto \iota_{lp,n}] \\ & [Lyr(\iota_{lp,1}) \mapsto L] \dots [Lyr(\iota_{lp,n}) \mapsto L] \\ & P \vdash e, h', s \leadsto_\delta \iota, h'' \\ h''' = & h''[Del_{h(C_n)} \mapsto Del_{\iota_{lp,n}}] \dots [Del_{h(C_1)} \mapsto Del_{\iota_{lp,1}}] \end{array}}{P \vdash withlayer(L)\{e\}, h, s \leadsto_\delta \iota, h'''}$$

The expression $withlayer(L)\{e\}$ is evaluated in an environment containing a given context-oriented program P, and the current program state, determined by the heap h and the stack s. What happens is that for each method m_i redefined by L, a new proxy object is created containing L's definition of m_i. These proxies are then inserted in the delegation chains of the objects representing the classes which originally defined them. As such, invocations of these methods during the evaluation of e will now be intercepted by these proxies and result in the execution of L's definition instead. Finally, as soon as e has been evaluated, the original delegation links are restored.

Although a formal semantics specified by means of reduction rules such as this one is useful to increase the understanding of the behavior of COP programs, its application to a particular COP program is often a tedious, manual task. Moreover, during this process, program state is only present implicitly in the variables h and s.

In a graph-based semantics, all necessary information is encoded in the graph representation of the program. Looking at the above reduction rule, this requires every layer to be represented in such way that a proxy can be constructed upon layer activation for every method redefined by that layer. We propose to have these proxies available at program start, so that layer activation merely comes down to inserting them in the appropriate delegation chains.

Consider the simple context-oriented program in Listing 1, written in ContextJ-like pseudocode. It consists of two classes

```
1  class A {
2      int f;
3      void p() { this.f = 100; }
4      void m() {
5          A a = ... ; B b = ... ;
6          with(L) { a.p(); b.n(); }
7      }
8  }
9  class B {
10     int g;
11     void n() {this.n = 500; }
12 }
13 layer L {
14     void A.p() { this.f = 1; }
15     void B.n() { this.g = 4; }
16 }
```

Listing 1: Example context-oriented program.

A and B and a layer L, which overrides method p of class A and method n of class B. The layer is activated during part of the execution of A's method m, in which the overridden methods p and n are called (line 6). Figure 8 shows the graph representation of the state of this program at a point where L is not active and the message m has been sent to an object $a1$ and is waiting to be processed. Given that delMD-SOC assumes an object-based environment, classes are represented as objects, just like their instances. Instances hold fields corresponding to the attributes of their class, and delegate to the class object, which holds a method dictionary. Recall from Section 2 that all objects are represented as a combination of a proxy and the actual object. Hence, all information regarding a high-level object is captured by four machine-level objects: An object representing the instance, an object representing the class and a proxy for each of these. Class objects can be reused, which is why $a1$ and $a2$, which are both instances of A, partially share the same delegation chain in Figure 8. Apart from $a1$ and $a2$, there is also an instance $b1$ of class B, and an object L representing the layer L as well as two proxies LAp and LBn which will, upon layer activation, be used for overriding the methods specified in the layer definition. Since we are not dealing with a concurrent program, only one actor node is present, which contains all objects (by design, only objects which can receive messages need to be connected by a *contains* edge). The actor's mail queue holds a message m, which will be processed by removing it from the mail queue onto the process stack, looking up a corresponding method body in the delegation chain of the receiver $a1$, and executing it.

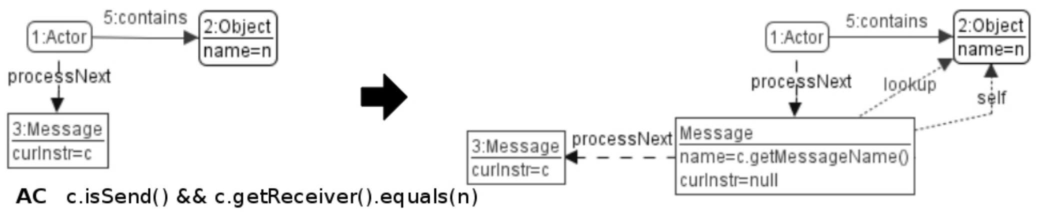

Figure 6: Rewrite rule for sending a new message.

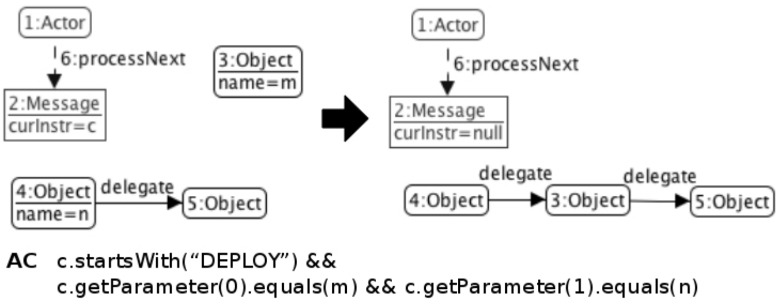

Figure 7: Rewrite rule for proxy deployment.

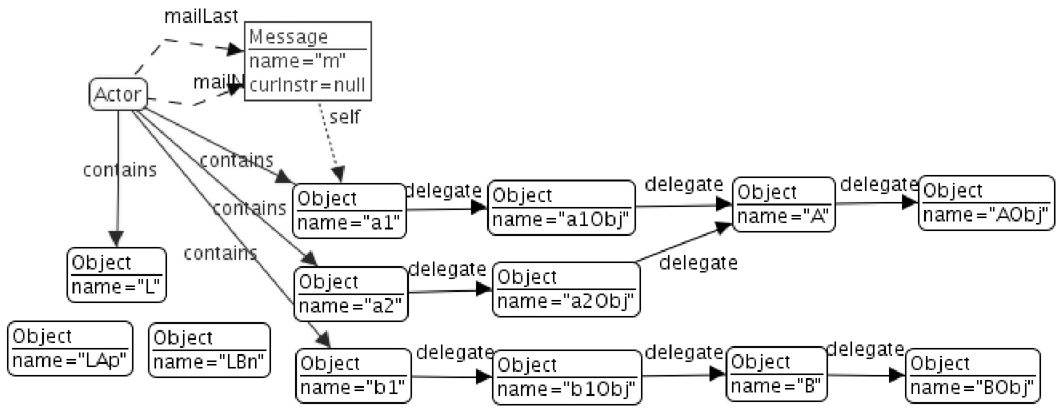

Figure 8: Initial program state.

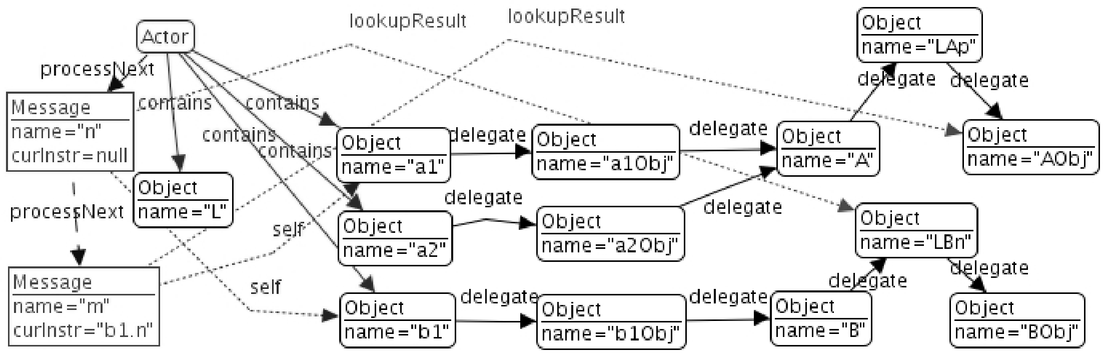

Figure 9: Program state after layer activation.

Applying the graph rewrite rules relevant for this lookup process (cf. Section 3) will result in the following definition of m being found in the class object *AObj*:

```
...
SEND( activate ,L)
SEND(p , a1 )
SEND(n , b1 )
SEND( deactivate ,L)
```

Essentially, the *with* construct from Listing 1 was translated into two message sends to the layer object L: An *activate* message upon entrance of the scope, and *deactivate* upon exit. A definition for these two messages is indeed provided in L's method dictionary, where *activate* is defined as:

```
DEPLOY(LAp,A)
DEPLOY(LBn,B)
```

The instructions will cause the graph rewrite rule for deployment (cf. Section 3) to be applied, resulting in the two proxies being inserted in the delegation chains immediately after the object provided as second parameter. The definition of *deactivate* simply reverses this effect:

```
UNDEPLOY(LAp)
UNDEPLOY(LBn)
```

Figure 9 shows the situation where L has already been activated, the message n has been looked up in the delegation chain of $b1$ and a definition has been found in LBn, as can be inferred from the *lookupResult* edge being present between the message and LBn. The latter, just like LAp, has been inserted in the delegation chain of the appropriate class object, as a consequence of L's activation. Further processing of the message n will result in the value of field g of $b1$ to be set to 500. Finally, L is deactivated again, and the delegation chains are restored to the state in Figure 8.

As the graph rewrite rules defining the operational semantics of delMDSOC have been specified using the AGG tool [8], the example outlined above can be simulated automatically. This effectively visualizes the operational semantics of a context-oriented program in terms of objects, messages and delegation. Comparing this approach to the SOS reduction rule at the beginning of this section, it should be noted that the SOS rule captures the general case, i.e., is applicable for any arbitrary context-oriented program, whereas the graph-based approach concentrates on one particular (albeit abstract) program. Hence, the latter should be accompanied by an algorithm describing the encoding of an arbitrary context-oriented program into a graph representation which can serve as input to the graph grammar describing the delMDSOC semantics. Essentially this comes down to mapping all high-level COP constructs onto entities in the delMDSOC model. From the above example we derive that classes are mapped onto normal low-level objects (including a proxy), as are their instances, layers are represented by a single object and a proxy for each method they override, and the *with* construct is mapped onto the sending of an *activate* and a *deactivate* message respectively upon entrance and exit of the corresponding dynamic scope. In other words, the essentials of this mapping remain exactly the same as for an SOS-based approach. A more elaborate description can be found in [7].

5. CONCLUSION

This paper presented a graph-based semantics for context-oriented programming on top of a graph-based operational semantics for the delegation-based delMDSOC machine model. The latter consists of a number of graph rewrite rules which describe the dynamics of a delegation-based message sending system. These rules may be applied to graph structures representing program state, in order to simulate program execution. By representing context-oriented programs in this manner, the delMDSOC rewrite rules are capable of describing the semantics of dynamically scoped layer activation and hence (a flavor of) context-oriented programming in general. As delMDSOC supports actor-based concurrency, an interesting path of future work comprises the description of a concurrent semantics for context-oriented programming.

6. REFERENCES

[1] Mehmet Aksit, Arend Rensink, and Tom Staijen. A graph-transformation-based simulation approach for analysing aspect interference on shared join points. In *Proceedings of the 8th ACM international conference on Aspect-oriented software development*, pages 39–50, Charlottesville, Virginia, USA, 2009. ACM.

[2] Dave Clarke and Ilya Sergey. A semantics for context-oriented programming with layers. In *International Workshop on Context-Oriented Programming - COP '09*, pages 1–6, Genova, Italy, 2009.

[3] Michael Haupt and Hans Schippers. A machine model for aspect-oriented programming. In *ECOOP 2007 - Object-oriented Programming, 21st European Conference, Berlin, Germany, July 30 - August 3, 2007, Proceedings*, volume 4609 of *Lecture Notes in Computer Science*, pages 501–524. Springer, 2007.

[4] Tim Molderez, Hans Schippers, and Dirk Janssens. A graph-based operational semantics for a machine model with actor-based concurrency. In *8th BElgian-NEtherlands software eVOLution seminar (BENEVOL 2009)*, Louvain-la-Neuve, Belgium, 2009.

[5] Grzegorz Rozenberg, editor. *Handbook of graph grammars and computing by graph transformation*, volume 1-3. World Scientific Publishing Co., Inc., 1997.

[6] Hans Schippers, Tom Van Cutsem, Stefan Marr, Michael Haupt, and Robert Hirschfeld. Towards an actor-based concurrent machine model. In *ICOOOLPS'09 workshop at ECOOP'09*, 2009.

[7] Hans Schippers, Dirk Janssens, Michael Haupt, and Robert Hirschfeld. Delegation-based semantics for modularizing crosscutting concerns. In *OOPSLA 2008 - Conference on Object-Oriented Programming Systems, Languages and Applications, Nashville, TN, USA, oct 19 - oct 23*, page 525–542, New York, NY, USA, 2008. ACM.

[8] Gabriele Taentzer. AGG: a graph transformation environment for modeling and validation of software. In *Applications of Graph Transformations with Industrial Relevance*, pages 453, 446, 2004.