In Proceedings of the Third International Workshop on Academic Software Development Tools and Techniques (WASDeTT-3) 2010,
co-located with the Conference on Automated Software Engineering (ASE) 2010,
pages 129-145, Antwerp, Belgium, September 20-24, 2010.

A Platform for Experimenting with Language Constructs for Modularizing Crosscutting Concerns

Tim Molderez, Hans Schippers, Dirk Janssens

Antwerp Systems and Software Modeling (Ansymo)

University of Antwerp, Belgium

Michael Haupt, Robert Hirschfeld

Software Architecture Group

Hasso Plattner Institute, Potsdam, Germany

Abstract

When implementing a new programming language construct, it is important to consider and understand its implications on program semantics. Simply hacking compiler code, even in combination with the use of a debugger, does not allow for easily keeping track of the global picture of overall execution semantics. We present a graph-based implementation of the delMDSOC virtual machine (VM) model in AGG, as a platform for experimenting with programming language constructs. More specifically, given the nature of the delMDSOC model, it is aimed primarily at languages supporting the modularization of crosscutting concerns, such as aspect-oriented or context-oriented languages. Our delMDSOC implementation visualizes programs as graphs at the VM level, in terms of well-known and intuitive concepts: objects, messages, delegation and actors. Implementing new high-level language constructs involves expressing them in terms of these concepts. Since delMDSOC is implemented as a graph rewriting system, program execution can be visually simulated and program state can be inspected at all times, providing insight in the implications of the new language construct on execution semantics. We demonstrate our approach by means of two language constructs: the context-oriented layer construct and a new aspect-oriented construct, the "concurrent cflow" pointcut.

Keywords: virtual machine model, programming language development, modularizing crosscutting concerns, graph rewriting

1. Introduction

New constructs are being added to programming languages regularly, sometimes leading to language extensions or even completely new programming languages. For the language devel-

Email addresses: `tim.molderez@ua.ac.be` (Tim Molderez), `hans.schippers@ua.ac.be` (Hans Schippers), `dirk.janssens@ua.ac.be` (Dirk Janssens), `michael.haupt@hpi.uni-potsdam.de` (Michael Haupt), `hirschfeld@hpi.uni-potsdam.de` (Robert Hirschfeld)

oper, as well as the language user, it is crucial to understand the semantics of such new constructs, and their implications on the execution semantics of programs written in this language.

New constructs are often simply "hacked in" by modifying compiler or interpreter software, the semantics being established by their implementation. The language user then either relies on high-level documentation, which is often lacking, or on the compiler or interpreter itself, using it as a black box to figure out the semantics of such constructs.

This paper presents a platform that can be used to experiment with object-oriented language constructs. This platform is based on the delMDSOC (delegation-based Multi-Dimensional Separation Of Concerns) virtual machine (VM) model [1] and its implementation in the AGG graph transformation tool [2]. In essence, delMDSOC is a VM model that is designed as a compilation target for programming languages aiming to increase modularization and enhance (multi-dimensional) separation of concerns (MDSOC). Examples of such languages include aspect-oriented programming [3], context-oriented programming [4] and role-based programming [5]. Much attention and effort is devoted to experimenting with new language constructs within this fairly young area of programming language research [6, 7, 8, 9].

Our AGG implementation of delMDSOC visualizes programs as graphs at the VM level. The behavior of the programs represented by these graphs is determined by a set of graph rewrite rules [10] which constitute the operational semantics of delMDSOC. As such, AGG allows for visual simulation of program execution by repeatedly transforming the program graph through application of these rules. In between rule applications, execution can be paused and the entire program state can be inspected and modified, including the VM code that each object implements, and even the runtime stack. Because program state is represented by a graph, it also is possible to make backup copies thereof, such that execution can always be resumed from a backup.

What makes this platform useful for experimentation is the fact that, unlike most intermediate languages, delMDSOC's abstraction level is much closer to a high-level programming language than it is to a typical machine instruction set. Because the distance between the programming language and the VM instruction set is decreased, it becomes easier to reason about the programming language. More concretely, languages that are implemented on top of delMDSOC will be expressed in terms of the following well-known high-level concepts: prototype objects, message sending, delegation and actors. A wide range of language constructs can be created using these building blocks [1, 11], due to the delMDSOC model's dynamic nature: All message sends are late-bound and delegation relations between objects can be modified at runtime.

The remainder of this paper is structured as follows: Sec. 2 provides an overview of the delMDSOC machine model. Next, Sec. 3 presents the model's operational semantics in depth. In Sec. 4, we apply our semantics to two examples of language constructs: the existing context-oriented layer construct and a new aspect-oriented construct, the "concurrent cflow" pointcut. Sec. 5 discusses related work, and 6 concludes and briefly outlines future work.

2. Overview of the delMDSOC model

The delMDSOC machine model was originally introduced in [1]. In this section, we give an overview of the model's operation.

As the model is a prototype-based object-oriented environment, the model's basic entities are objects, which can communicate with each other by means of message passing. We opted for prototype objects in favor of a class-based object-oriented model because we wanted the model

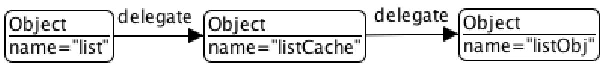


Figure 1: An example of a composite object

to consist of a minimal amount of concepts, keeping it as simple as possible. As there is no concept of classes in prototype-based languages, this was an easy choice to make.[1]

Objects can be connected to each other using delegation, resulting in a chain of objects, also called a *composite object*. An example of such a composite object is shown in Fig. 1. During message lookup, if an object receives a message it cannot understand, the message is delegated to the next object in the chain, and so on, until a matching implementation is encountered in one of the objects' message dictionaries.

In delMDSOC, the first object in a composite object's delegation chain is a so-called *proxy*, which does not understand any messages at all. Its purpose is to serve as a fixed access point, establishing the identity of the composite object. This is important because delegation chains are not fixed, but may be modified at runtime. This mechanism allows for dealing with dynamic deployment of crosscutting modules. In such a scenario, objects may be inserted into or removed from delegation chains in order to dynamically modify a composite object's behavior in response to certain messages. The `listCache` object in Fig. 1 is an example of a crosscutting module that is currently enabled, as it is inserted between the `list` proxy and the `listObj` object. During message lookup, the *self* pseudovariable always remains bound to the receiver of the message, i.e., the proxy, ensuring that lookup for messages sent by a composite object to itself always starts at the front of the delegation chain.

In order to introduce concurrency into delMDSOC, we have opted for the actor-based model of concurrency. Actors were chosen in favor of threads, also because of simplicity reasons: Because actors do not allow for shared state, issues such as deadlocking are avoided for the most part. However, we do not adhere to the pure actor model, but rather take an approach similar to the one employed in the E language [12], in which a distinction is made between actors and objects: Actors act as containers of objects. An object can send an asynchronous message to an object belonging to another actor, which results in this message being appended to the other actor's *mail queue*. The mail queue essentially is a buffer where messages are kept until the message that is currently being processed has finished its execution. When this happens, the message at the front of the mail queue is removed and is pushed onto the *process stack*. Although an actor may at all times receive additional messages in its mail queue, it will not process them until it has dealt with the messages currently residing on the process stack. Furthermore, executing a message on the process stack may result in other message sends. If these messages target objects within the same actor, they immediately end up on the process stack, which means they are processed synchronously. In short, communication between objects within one actor (intra-actor communication) occurs synchronously, whereas communication involving two different actors (inter-actor communication) occurs asynchronously by default. Asynchronous inter-actor communication may result in a *future* object [13] being returned. Upon accessing a future's value, the actor is blocked until the value has been calculated. This blocking mechanism can be also used in order to simulate synchronous inter-actor communication.

[1]If needed, classes can always be emulated using prototype objects in combination with delegation [1].

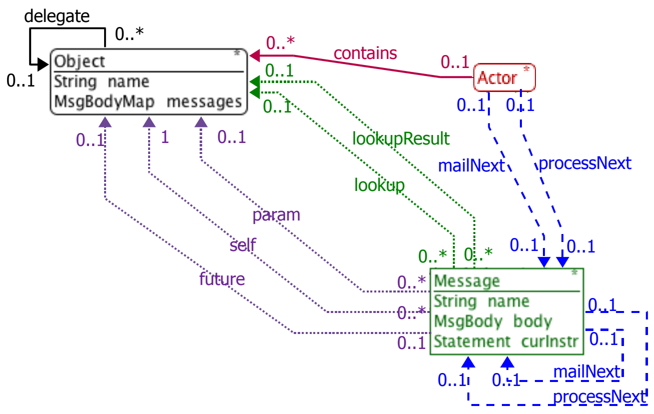


Figure 2: Type graph of the delMDSOC model

2.1. *The model's operation*

The delMDSOC model is described as a set of graph rewrite rules. These rules can be subdivided according to the cycle of steps that each actor goes through:

1. Move the first message from the mail queue to the process stack. (This step is detailed in Sec. 3.1.)
2. Perform the lookup procedure to find the message's implementation, also called the message body. (Sec. 3.2)
3. While the message body has not been executed completely:
   (a) Fetch the next instruction from the message body.
   (b) Perform some preprocessing on this instruction, if any. (Sec. 3.3)
   (c) Execute the instruction. (Sec. 3.4)
4. Pop the message from the process stack and start over at step 1.

While the model's graph rewrite rules can be matched in a non-deterministic order, they are set up such that each actor will indeed follow the above sequence of steps. It should also be mentioned that, while delMDSOC supports a whole set of different instructions, this paper will only focus on a few of these instructions. Several instructions, such as integer addition, do not modify the graph structure, but only attribute values, which is of limited interest in this context. Instead, message sends and object (un)deployment will be discussed in depth in Sec. 3.4. These are more interesting because the mapping of several high-level MDSOC constructs onto the delMDSOC machine model will involve these instructions.

2.2. *Type graph*

Fig. 2 shows the type graph associated with our model, which specifies the model's different kinds of nodes and edges, and how both connect to each other. Three kinds of nodes are available: actors, objects and messages. Node attributes are typed by Java classes.[2] Object nodes contain

[2]It is important to note that we only make use of a minimal subset of Java's functionality. Java expressions are only used in order to read and update attribute values, but they are not allowed to cause side-effects to the graph structure.

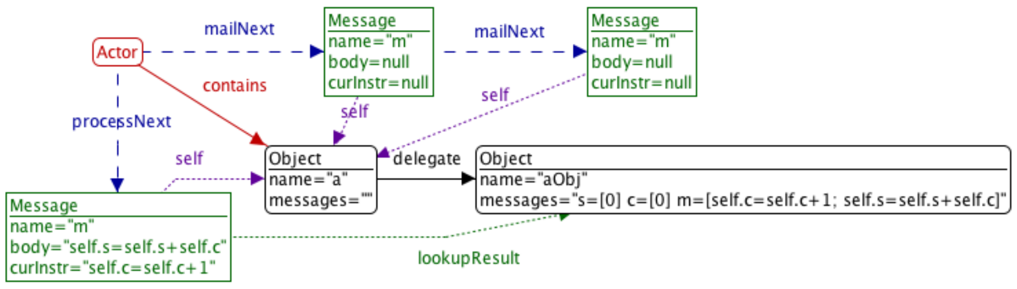


Figure 3: An intermediate state of a simple program

two attributes: `name` and `messages`. The `name` attribute is a `String` acting as a unique identifier. The `messages` attribute is a message dictionary containing implementations for the messages it understands. It has the `MsgBodyMap` type, which is a wrapper class around a hash map, mapping each message name to a list of instructions.

Message nodes contain three attributes: `name`, `curInstr` and `body`. Once a message has been looked up, the message body is stored in the `body` attribute as a `MsgBody`, a linked list of `Statements`, the latter being a wrapper around a `String`. When a statement is executed, it is removed from `body` and is then stored in the `curInstr` attribute, which represents the instruction currently being executed.

Next to the three types of nodes, there also are several types of edges:

- The `contains` edge connects an object to the actor containing it.
- The `delegate` edge indicates the next object in a composite object's delegation chain.
- The `mailNext` edges form an actor's mail queue. If the source node of a `mailNext` edge is an actor, the edge indicates the first message in an actor's mail queue. Otherwise, it indicates the next message in a mail queue.
- Analogous to the `mailNext` edge, the `processNext` edges are used to form an actor's process stack.
- The `lookup` edge indicates the object currently being checked in the message lookup procedure. A `lookupResult` edge then indicates the object where a message was understood.
- The `self` edge indicates a message's self object. Analogously, the `param` edge indicates a message's parameter object, used to pass parameters along with a message send.
- The `future` edge indicates that a message's return value should be stored inside a future object.

The graph in Fig. 3 illustrates an example that complies with the above type graph. It shows an intermediate state of a running program that will calculate the value of $\sum_{c=1}^{3} c$ and store the result in `aObj.s`. It incorporates several of the model's elements: There is one actor that contains one composite object `a`. The actor has two pending messages in its mail queue and is currently executing the instruction `self.c=self.c+1`, as shown in the message that is on the actor's

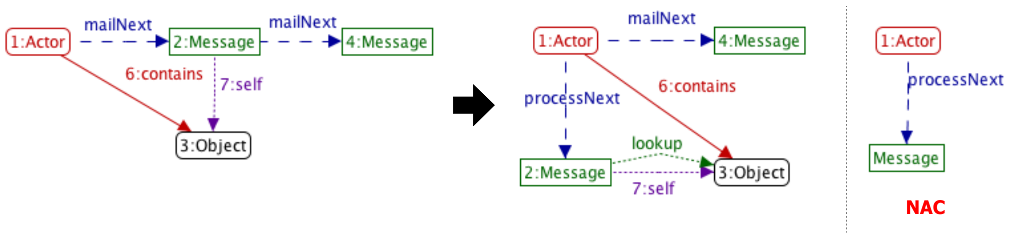


Figure 4: Process message rewrite rule

process stack. The rewrite rules described in the following section will give a general idea on how the execution of this program will continue.

3. Graph rewriting semantics of the delMDSOC model

This section presents the essentials of delMDSOC's semantics, in the form of a set of single-pushout graph rewrite rules.[3] Each graph rewrite rule contains at least two graphs: a left-hand-side (LHS) and a right-hand-side (RHS). The application of a rule then consists of searching for the rule's LHS within the graph that we wish to transform. If a match was found, it will be replaced by the RHS and the rule application will have succeeded; otherwise, the rule application has failed. Optionally, a rule can also have multiple attribute conditions and negative application conditions. An attribute condition (AC) is a Boolean expression that must evaluate to true in addition to matching a rule's LHS. A negative application condition (NAC) specifies a subgraph that may not occur when matching a rule.

3.1. Processing messages in the mail queue

The graph rewrite rule in Fig. 4 comes into play when an actor has an empty process stack, which is ensured by the rule's NAC. If an actor's process stack is empty, this means it is waiting for new messages to process. The rule's RHS removes the first message from the mail queue and moves it onto the actor's process stack. The message also receives a `lookup` edge, indicating that the lookup procedure should start, which is handled by the rules in Sec. 3.2. It should be noted that this rule will not work in the case where only one message is in the actor's mail queue. This issue was solved by making use of AGG's amalgamated rules, which makes it easier to create, maintain and combine variations on rules, rather than having to specify each variant as a separate rule. Using this technique, a variant (not shown here) was created that handles the case with one message in the mail queue.

3.2. Message lookup

The lookup procedure described in this section behaves exactly like the lookup procedure discussed in Sec. 2. In case the desired message implementation is not found in the object currently indicated by the `lookup` edge, the rule in Fig. 5 matches. This is ensured by attribute condition `!msg.contains(m)`, which checks whether the object contains an implementation

[3]The complete set of graph rewrite rules as described in AGG is available at `http://fots.ua.ac.be/delmdsoc/`.

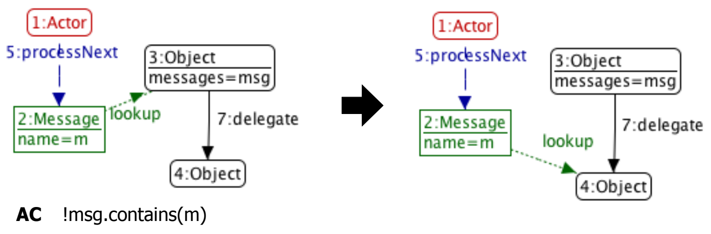


Figure 5: Iteration rule of the lookup procedure

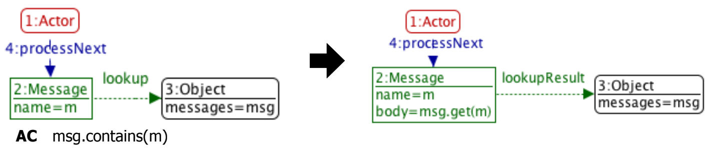


Figure 6: The end of the lookup procedure

for a message named `m`. Its RHS moves the `lookup` edge to the object's delegate, such that the lookup procedure continues there.

In the other case, if the desired implementation was found in the current object, the rule in Fig. 6 matches. Its RHS copies the implementation into the message's `body` attribute and the `lookup` edge is changed into a `lookupResult` edge, indicating that the lookup procedure has finished and that the body can now be executed.

3.3. *Preprocess the current instruction*

Before an instruction can be executed, some preprocessing may be required. Any references to the `self` pseudovariable in an instruction are replaced by the name of the self object, as indicated by the current message's `self` edge. Similarly, references to formal parameters and future objects can be resolved in the preprocessing phase.

Instructions that contain subexpressions that haven't been evaluated yet are also handled during preprocessing: There is a rule that will push a new message on the process stack that is meant to evaluate this subexpression. Once this is finished and the message can be popped, the subexpression's resulting value is resolved in the original instruction.

3.4. *Execute the current instruction*

The intermediate language implemented by our machine model supports several different types of high-level instructions:

There are the usual assignments, if statements, print statements and (implicit) return statements, but there also are more interesting instructions for sending messages, resending messages (analogous to the aspect-oriented proceed statement), creating new actors, cloning objects and (un)deploying objects in delegation chains. This section concentrates only on sending messages and object deployment, as these form the essence of the examples in Sec. 4.

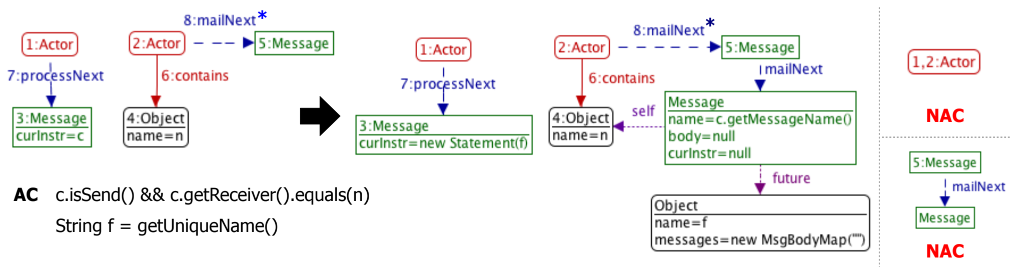


Figure 8: Inter-actor communication

3.4.1. Intra-actor communication

The rule handling communication between objects within the same actor, i.e., intra-actor communication, is shown in Fig. 7. The first AC makes sure the current instruction has the syntax of a message send and that its receiver equals the object with name `n`. (For example, the instruction `listObj.get(3)` sends the message `get` to receiver `listObj` with parameter `3`.) The second AC will check for the presence of the actual parameter object that is passed along with the message send. For simplicity reasons, the delMDSOC model can currently only pass one parameter; multiple parameters can be simulated by using this one parameter object as a container. It also is possible to perform a message send without any parameters; amalgamated rules are used here once again to define the variant of this rule without parameters.

Once this rule matches, the new message that will be sent is pushed onto the actor's process stack. As this message is now at the top of the stack, the other messages on this stack have to wait until the new message has been looked up and executed. In other words, this is a synchronous message send.

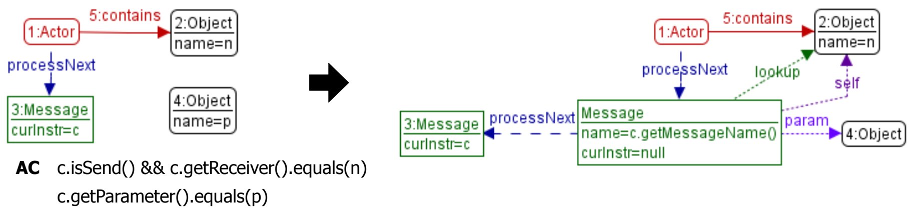
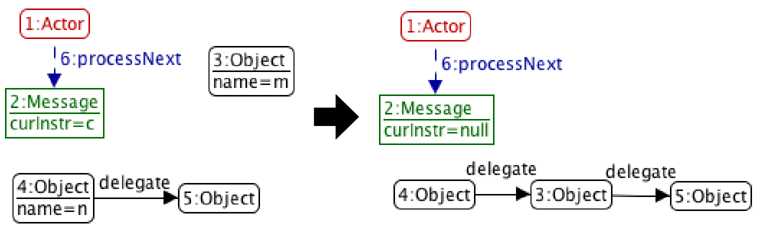


Figure 7: Intra-actor communication

3.4.2. Inter-actor communication

Sending messages between two different actors is handled by the rule in Fig. 8. This rule is one of the few occasions where it becomes apparent that our graph rewriting rules do not use injective matching, in which each node in a rule must be different from each other. This means it could occur that both actors, node 1 and 2, are the same actor. To prevent this situation, a NAC was added that ensures the two actor nodes may not be the same.

In contrast to intra-actor communication, the new message does not immediately end up at the top of the receiving actor's process stack, but it is added to the back of its mail queue. To



AC c.startsWith("DEPLOY")
c.getParameter(0).equals(m) && c.getParameter(1).equals(n)

Figure 9: The deployment instruction

retrieve the last message in the mail queue, transitive closure[4] (marked by the asterisk on edge 8) is used in combination with another NAC. The actor that initiated the message send can continue immediately, meaning that message sends between different actors are asynchronous by default.

To be able to move the return value back to the sending actor once it is available, we make use of futures [13]. In the rule's RHS, a new object is created with a unique name `f` and a `future` edge pointing to it; this is a future object. Once the message that was sent has been executed, the return value will be stored inside this future object. In the meantime, the message that caused the message send (node 3) will receive a reference to the future object as a temporary return value. This reference can be assigned to fields and be passed around as a parameter without a problem, even though the future object may still be empty. When accessing the future object however, the actor will block until the return value is present in the future object. Once the return value is available, another rewrite rule will make sure that accesses to the future object are replaced by the return value contained within. All of this happens in a transparent manner, in the sense that someone making use of delMDSOC does not need to be aware of the existence of future objects to be able to use them. Synchronous inter-actor communication can be simulated using the future's blocking mechanism, which is done by making an asynchronous call and immediately attempting to access its return value. An explicit `SYNC` instruction is available that provides this functionality.

3.4.3. Deployment

The deploy instruction is a key component that allows the model to be used in an MDSOC context. This instruction is used to insert an object within another specified delegation chain. For example, when executing the statement "`DEPLOY(objectA,objectB)`", `objectA` will be inserted between `objectB` and its delegate. The rule handling this instruction is shown in Fig. 9. The attribute conditions of the rule express that the current instruction must be a deploy instruction and that both of its parameters should match with the objects respectively named `m` and `n`.

[4]AGG currently does not support transitive closures, so an additional type of edge is used to explicitly mark the last message in a mail queue.

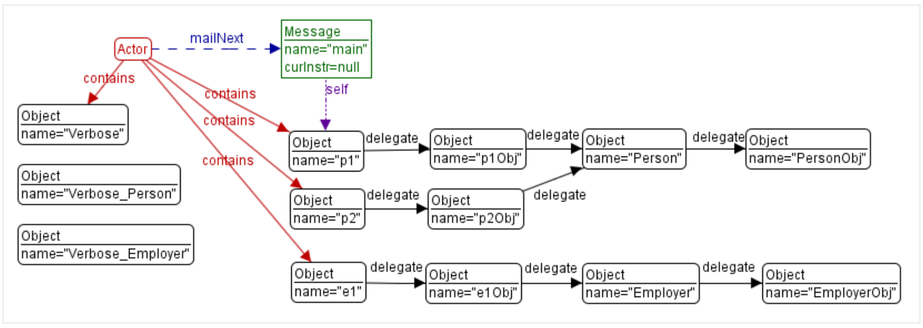


Figure 10: Initial program state

4. Examples

We will now, by means of two examples, illustrate how the set of graph rewrite rules described in the previous section, as well as their simulation in AGG, can be a useful tool while experimenting with language constructs. The first example involves an existing construct that forms the basis of context-oriented programming: layers. The second example introduces a new variation on the aspect-oriented cflow pointcut, called concurrent cflow, or `ccflow`.

4.1. The context-oriented layer construct

Before diving into any details of the example, we will first give a brief introduction to context-oriented programming [4]. The main idea behind context-oriented programming is that some functionality may behave differently when executed in a different context. To this end, the layer construct was introduced; a layer contains the behavioral variations of certain methods in a particular context. More concretely, alternate definitions of any method can be written within a layer. These alternate definitions can then be explicitly activated, i.e. replace their original definitions, whenever a particular dynamic scope is entered.

Listing 1 shows a simple context-oriented example that we wish to map to the delMDSOC model. It is written in a ContextJ-like language [4], which is a context-oriented extension of Java. In this example, two classes are present, `Person` and `Employer`; each of these defines a `toString` method. We redefine the `toString` method of each class within the `Verbose` layer. Also note the use of the `proceed` statement within this layer definition; this represents a call to the original method definition. In `Person.main`, the layer is activated using the `with` block construct. In other words, all code executed in the dynamic scope of the `with` block will make use of the redefined `toString` methods.

Fig. 10 shows the corresponding representation of the initial state of this program in delMDSOC. (The `messages` and `body` attributes are hidden, as they would clutter the graph.) At this point, the `Verbose` layer is not active yet. The `main` method is called by sending a `main` message to object `p1`. Given that delMDSOC 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 Sec. 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

```
class Person {
    String name; Employer emp;
    String toString() {return name;}
    void main() {
        Employer e=new Employer("Cosmo Spacely",
            "Sprocketlane 23, Orbit city");
        Person p=new Person("George Jetson", e);
        with(Verbose) {
            System.out.println(p.toString());
        }
    }
}

class Employer {
    String name; String addr;
    void toString() {return name;}
}

layer Verbose {
    void Person.toString() {
        return proceed() + "Employer:" + emp.toString();
    }
    void Employer.toString() {
        return proceed() + "Address:" + addr.toString();
    }
}
```

Listing 1: Example context-oriented program

objects can be reused, which is why `p1` and `p2`, which are both instances of `Person`, partially share the same delegation chain in Fig. 10. Apart from `p1` and `p2`, there is also an `Employer` instance `e1`. The `Verbose` layer is represented by the objects `Verbose`, `Verbose_Person` and `Verbose_Employer`. The first implements the logic of layer activation and deactivation; the other two contain the layer's redefined `toString` methods. Because we are not dealing with a concurrent program, only one actor node is present, which contains all objects (by design, only objects that can receive messages need to be connected by a `contains` edge). The actor's mail queue holds a message `main`, which will be processed by removing it from the mail queue and pushing it onto the process stack, looking up a corresponding method body in the delegation chain of the receiver `p1`, and executing it.

Applying the graph rewrite rules relevant for this lookup process (cf. Sec. 3.2) will result in the following definition of `main` being found in the `Person` class object:

```
...
Verbose.activate
PRINT(p2.toString)
Verbose.deactivate
```

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

```
DEPLOY(Verbose_Person,Person)
DEPLOY(Verbose_Employer,Employer)
```

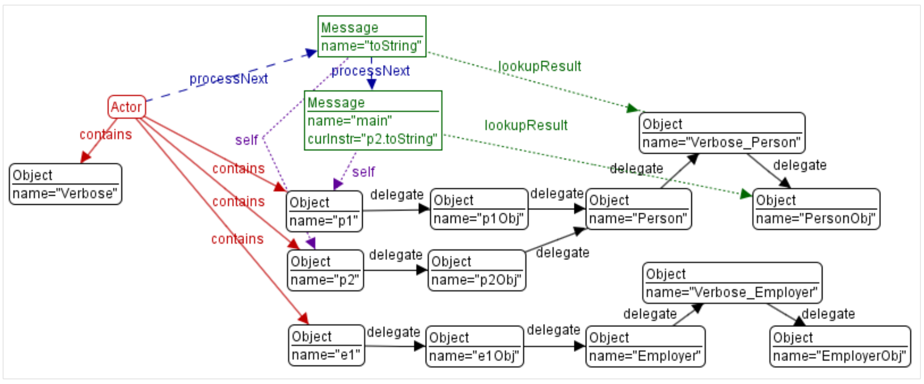


Figure 11: Program state after activation

The instructions will cause the graph rewrite rule for deployment (cf. Sec. 3.4) to be applied, resulting in the two objects being inserted in the delegation chains immediately after the object provided as the second parameter. The definition of `deactivate` simply reverses this effect:

```
UNDEPLOY(Verbose_Person)
UNDEPLOY(Verbose_Employer)
```

Fig. 11 shows the situation where the layer has already been activated, the message `toString` has been looked up in the delegation chain of `p1` and a definition was found in `Verbose_Person`, as can be inferred from the `lookupResult` edge between the message and `Verbose_Person`. The latter, just like `Verbose_Employer`, has been inserted in the delegation chain of the appropriate class object, as a consequence of the layer's activation. After executing the `toString` message, the `Verbose` layer is deactivated again and the delegation chains are restored to the state in Fig. 10.

As the graph rewrite rules defining the operational semantics of delMDSOC have been specified using the AGG tool [2], 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. From the above example we derive that classes are mapped onto normal low-level objects, as are their instances, layers are represented by a set of objects for each method they override and an object that implements the layer's activation and deactivation, respectively upon entrance and exit of the corresponding dynamic scope.

For language developers, being able to break down program execution in basic simulation steps while visualizing object relations and program state is a useful tool in order to verify whether a new language construct behaves according to desirable semantics.

For the language user, it may help in truly understanding the impact of a particular language construct or in fact a whole programming paradigm.

4.2. The concurrent cflow pointcut construct

Consider the sample code in Listing 2, written in an imaginary aspect-oriented language with support for actor-based concurrency. In this example, two actors are present, an `Admin` and a `Client`, both of which can manipulate a database, independent from each other. At any point however, the administrator may initiate a backup of the database. This can cause a synchronization issue, in the sense that updating the database while the backup is created

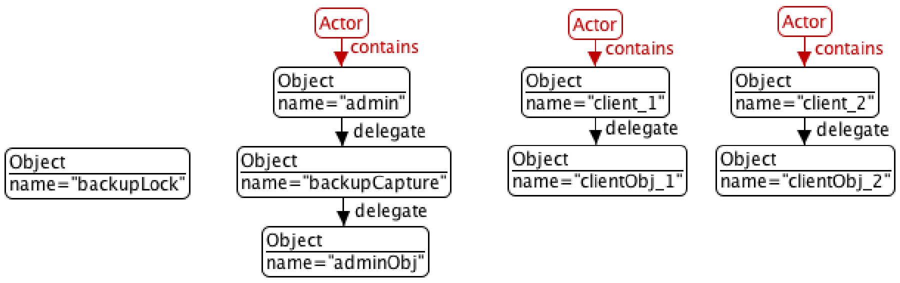


Figure 12: Initial object configuration

could render this backup file corrupt. A simple strategy to prevent this from happening is to block any attempts to update the database during the backup operation. This is expressed by the `BackupLock` aspect, which makes use of a special "concurrent cflow" construct, or `ccflow`, in its pointcut. This construct is a concurrent variant of the aspect-oriented `cflow` pointcut construct as present in AspectJ [3]. The intuition behind this aspect is that, for the duration of the `backup` call in the `Admin` actor, all calls to the `update` method in the `Client` actor should be intercepted and result in an error message. The expressiveness of the existing `cflow` construct is insufficient in this scenario, since it can only be applied within a single thread of control. Any `update` call will be in another control flow than the one of `backup`, as they are executed by different actors. In Sec. 4, we will show how delMDSOC can help in establishing the semantics of this new `ccflow` construct.

```
actor Admin {                       actor Client {
  DbConnection conn;                  DbConnection conn;
  void backup() {                     List read(...) {
    conn.backup();                      return conn.query(...);
  }                                   }
}                                     void update(...) {
                                        conn.query(...);
                                      }
                                    }
aspect BackupLock {
  around():ccflow(execution(Admin.backup))
  && execution(Client.update) {
    print("ERROR: access denied");
  }
}
```

Listing 2: Example usage of the `ccflow` construct

The initial object configuration of the example in its delMDSOC representation is shown in Fig. 12. The figure shows three composite objects representing the database's administrator, `admin`, and two instances of client users, `client_1` and `client_2`. The database itself and any class objects[5] are not shown here, as they do not play a significant role in this example.

The pointcut needed to capture all database updates across all clients during a backup operation is as follows, in an AspectJ-like syntax:

[5]Classes can be simulated in a prototype-based environment by letting objects representing instances delegate to an object that represents the class.

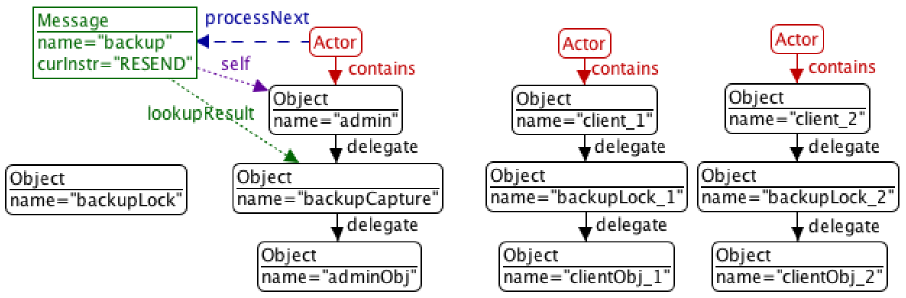


Figure 13: After deploying the `backupLock` objects

```
ccflow(execution(Admin.backup)) && execution(Client.update)
```

The `ccflow` pointcut construct is an extension of the regular `cflow` construct. It captures *all* join points across all actors in the duration of the specified control flow, whereas regular `cflow` can only capture the join points *within* a particular control flow.

To be able to capture the control flow of a backup operation, the object `backupCapture` has already been deployed in the `admin` composite object. The example scenario will be initiated once a `backup` message is sent to `admin`. This message will be intercepted by `backupCapture` before it can reach `adminObj`, which implements the backup operation itself. The implementation of the `backup` message in the `backupCapture` object can be expressed as follows:

```
SYNC(client_1.deployLock)     // Deploy backupLock at client_1
SYNC(client_2.deployLock)     // Deploy backupLock at client_2
RESEND                        // Do the backup operation
client_2.undeployLock
client_1.undeployLock
```

The first two instructions will deploy a clone of the `backupLock` object in each client's delegation chain. The `backupLock` objects will intercept any `update` message; its implementation of `update` contains our aspect's advice; it simply prints an error message stating that the database may currently not be updated. Also notice the use of the `SYNC` construct, first mentioned in Sec. 3.4.2. It indicates that the enclosed inter-actor message send should be synchronous. This ensures that our aspect is deployed *before* the backup operation is started. The program's current state after the two `backupLock` objects have been deployed is shown in Fig. 13. At this point, the actual backup operation can safely proceed. This is done with the `RESEND` statement; it resends the backup message starting at `backupCapture`'s delegate, `adminObj`. Once finished, we have left the control flow of the backup operation, so we can revert back to the original object configuration by undeploying the `backupLock` clones. All users are now free to modify the database once again.

A few remarks should be made regarding this example: Firstly, the solution presented here assumes that the `backup` operation does not contain any recursive calls. If a recursive backup call were made, it would be intercepted by the `backupCapture` object again, which is not desired. This can be solved by letting the `backupCapture` object undeploy itself before the backup operation is started and then redeploying itself once the operation has finished. A second remark about this solution is that our aspects, being the `backupLock` objects, are deployed per instance. This allows the `backupLocks` to keep their own separate state, which could be used, for example, to maintain a separate log of attempted database updates per client. However, this comes at the price of added complexity: The fact that clients can be added and removed at runtime should also be taken into account. While not shown in this example, this is handled by intercepting the instantiation of new clients, i.e., intercepting the `new` message that is sent to the client class object. For every new client that is created, a copy of the `backupLock` is deployed into the client's delegation chain. Additionally, the `backupCapture` object is notified of every client that is added, who will maintain a list of all available clients. Whenever the backup operation finishes, `backupCapture` will use its list of clients to undeploy all `backupLock` instances.

5. Related work

In the field of MDSOC languages, several different semantics are available, especially for aspect-oriented languages. Typically, such semantics are expressed as a structural operational semantics [14, 15, 16, 17]. However, there are a few instances in which graph rewriting is used: In [18], a graph-based operational semantics is provided for an aspect-oriented extension of Featherweight Java, where it is used for verification purposes. The idea of a machine model as a target for a multitude of MDSOC paradigms is not present however. This is apparent, for example, in the fact that an explicit *proceed* stack is modelled, in order to accommodate for the corresponding high-level language construct.

There are also a few instances of intermediate languages taking a high-level approach: The SUIF [19] infrastructure provides a high-level object-oriented intermediate representation and a toolkit for mapping programming languages onto this intermediate representation. This infrastructure is used as a vehicle to experiment with compiler optimization techniques for languages such as C and Fortran. While not the primary focus of our model, this suggests that delMDSOC may be a useful platform to experiment with MDSOC-specific optimization techniques.

More recent work includes the C Intermediate Language (CIL) [20], which provides a high-level representation of C programs. Similar to the delMDSOC model, CIL aims to rely on a minimal amount of concepts, making it easier to reason about C programs. Rather than using CIL for experimentation purposes, it is used for analyzing program properties such as memory safety and for performing source-to-source program transformations.

6. Conclusion and Future Work

This paper has introduced the delMDSOC virtual machine model as a platform to experiment with language constructs. The model's graph-based semantics in combination with the AGG tool allow the language developer to simulate programs making use of the model. This simulation is well-suited for experimentation in the sense that a program's entire state is visible; any aspect of it can be freely modified whilst the simulation is paused; the program's state can also be copied and stored, such that simulation can always resume from that point. However, a scalability problem does arise due to the fact that the entire state is visible. Therefore, our current approach is only applicable to small toy examples, which may not be sufficient for experimenting with more complex language constructs. Improved graph rewriting tool support should help significantly though: There is a need for better ways of navigating large graphs, improved layouting algorithms and better means to filter out the information that currently is irrelevant to the user.

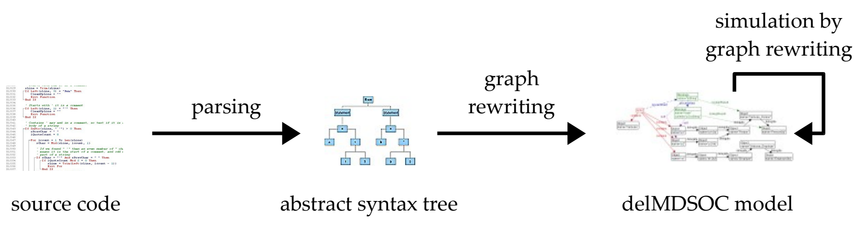


Figure 14: Constructing a new language with delMDSOC

The delMDSOC model itself is not yet considered to be in a finalized state. For example, the fact that the concurrency model is currently actor-based, is not a dogma. In fact, graph rewriting has been used before to model different approaches to concurrency, including process algebra [21] and Petri nets [22].

Regarding the experimentation platform itself, there also are several areas of future work that can be explored: Currently, the translation of programs into an AGG representation is a manual process. Ideally, this would be an automated process for each language. One approach would be to write a parser that uses the AGG API as a backend in order to generate an appropriate delMDSOC input graph. Another approach would be to have the parser only generate an AST in AGG format, and apply graph rewriting within AGG once more in order to transform the AST into an appropriate delMDSOC input graph. Fig. 14 illustrates the latter approach.

Another area of future work deals with the use of delMDSOC as a common framework in which MDSOC languages can be compared and integrated. The model may also be used for verification purposes, e.g., one can investigate the effects of the ordering of objects within a delegation chain or the condition under which it is safe to deploy or undeploy objects in remote actors.

References

[1] M. Haupt, H. 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, Vol. 4609 of Lecture Notes in Computer Science, Springer, 2007, pp. 501—524.

[2] G. Taentzer, AGG: a graph transformation environment for modeling and validation of software, in: Applications of Graph Transformations with Industrial Relevance, 2004, pp. 453, 446.

[3] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J. Loingtier, J. Irwin, Aspect-Oriented programming, in: M. Akşit, S. Matsuoka (Eds.), Proceedings European Conference on Object-Oriented Programming, Vol. 1241, Springer-Verlag, Berlin, Heidelberg, and New York, 1997, p. 220–242.

[4] R. Hirschfeld, P. Costanza, O. Nierstrasz, Context-Oriented programming, Journal of Object Technology, ETH Zürich 7 (2008) 125–151.

[5] S. Herrmann, Object teams: Improving modularity for crosscutting collaborations, in: NODe '02: Revised Papers from the International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World, Springer-Verlag, 2003, p. 248–264.

[6] C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, J. Tibble, Adding trace matching with free variables to AspectJ, in: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, ACM, San Diego, CA, USA, 2005, pp. 345–364.

[7] M. Appeltauer, R. Hirschfeld, H. Masuhara, M. Haupt, K. Kawauchi, Event-Specific software composition in Context-Oriented programming, in: Software Composition, 2010, pp. 50–65.

[8] A. Popovici, T. Gross, G. Alonso, Dynamic weaving for aspect-oriented programming, in: AOSD ’02: Proceedings of the 1st international conference on Aspect-oriented software development, ACM Press, New York, NY, USA, 2002, p. 141–147.
[9] M. Nishizawa, S. Chiba, M. Tatsubori, Remote pointcut: a language construct for distributed AOP, in: Proceedings of the 3rd international conference on Aspect-oriented software development, ACM, Lancaster, UK, 2004, pp. 7–15.
[10] G. Rozenberg (Ed.), Handbook of graph grammars and computing by graph transformation, Vol. 1-3, World Scientific Publishing Co., Inc., 1997.
[11] H. Schippers, D. Janssens, M. Haupt, R. Hirschfeld, Delegation-based semantics for modularizing crosscutting concerns, SIGPLAN Notices 43 (10) (2008) 525–542.
[12] M. S. Miller, E. D. Tribble, J. Shapiro, H. P. Laboratories, Concurrency among strangers: Programming in e as plan coordination, In Trustworthy Global Computing, International Symposium, TGC 2005 (2005) 195—229.
[13] J. H. C. Baker, C. Hewitt, The incremental garbage collection of processes, in: Proceedings of the 1977 symposium on Artificial intelligence and programming languages, ACM, 1977, pp. 55–59.
[14] D. Alhadidi, N. Belblidia, M. Debbabi, P. Bhattacharya, An AOP extended Lambda-Calculus, in: Proceedings of the Fifth IEEE International Conference on Software Engineering and Formal Methods, IEEE Computer Society, 2007, pp. 183–194.
[15] S. D. Djoko, R. Douence, P. Fradet, D. L. Botlan, CASB: common aspect semantics base, Tech. Rep. AOSD-Europe Deliverable D41, AOSD-Europe-INRIA-7, INRIA, France (2006).
[16] B. D. Fraine, E. Ernst, M. Südholt, Essential AOP: the a calculus, in: ECOOP, 2010, pp. 101–125.
[17] R. Jagadeesan, A. Jeffrey, J. Riely, A calculus of untyped Aspect-Oriented programs., in: L. Cardelli (Ed.), ECOOP 2003 - Object-Oriented Programming, 17th European Conference, 2003, pp. 54–73.
[18] T. Staijen, A. Rensink, Graph-based specification and simulation of featherweight java with around advice, in: Proceedings of the 2009 workshop on Foundations of aspect-oriented languages, ACM, Charlottesville, Virginia, USA, 2009, pp. 25–30.
[19] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. K. Tjiang, S. Liao, C. Tseng, M. W. Hall, M. S. Lam, J. L. Hennessy, SUIF: an infrastructure for research on parallelizing and optimizing compilers, SIGPLAN Not. 29 (12) (1994) 31–37.
[20] G. Necula, S. McPeak, S. Rahul, W. Weimer, CIL: intermediate language and tools for analysis and transformation of c programs, in: Compiler Construction, 2002, pp. 209–265.
[21] F. Gadducci, Graph rewriting for the pi-calculus, Mathematical Structures in Computer Science 17 (3) (2007) 407–437.
[22] P. Baldan, H. Ehrig, J. Padberg, G. Rozenberg, Workshop on petri nets and graph transformations, in: A. Corradini, H. Ehrig, U. Montanari, L. Ribeiro, G. Rozenberg (Eds.), Graph Transformations, Vol. 4178 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2006, pp. 467–469, 10.1007/11841883_35.