A Graph-based Operational Semantics for a Machine Model with Actor-based Concurrency

Tim Molderez, Hans Schippers, Dirk Janssens

University of Antwerp, Belgium

Abstract

This paper introduces an extension to an existing delegation-based machine model that offers support for multi-dimensional separation of concerns (MDSOC). More specifically, we add concurrency primitives based on the well-known actor model of computation and specify an operational semantics for this extended machine model using graph rewrite rules. Apart from being inherently suitable for dealing with parallelism, graph rewriting has the advantage of decent tool support, allowing for visual simulation. We validate our work by demonstrating a practical example in which the concurrency primitives are combined with existing MDSOC features.

Keywords: MDSOC, concurrency, actor model, graph rewriting, dynamic deployment, delegation

1. Introduction

The delegation-based *delMDSOC* machine model of [1] has good support for modularizing crosscutting concerns and hence for generally supporting programming languages enabling multi-dimensional separation of concerns (MDSOC). Its implementation, the *delMDSOC kernel*[1] [2], achieves most of the features required to implement MDSOC programming languages. An important feature still missing is explicit concurrency support. In [3] an informal description was given towards an extension of the *delMDSOC* model based on the well-known actor model of concurrency [4].

This paper takes this idea one step further and introduces an operational semantics of such an actor-based extension. Although a structural operational semantics (SOS) already exists for *delMDSOC* [1], we now opt for an approach based on graph rewrite rules [5]. One reason for this choice is that the application of a graph rewrite rule typically only affects a small part of the complete graph. Hence, provided they affect disjoint parts of the graph, several rules may be applied at the same time without interference, rendering the mechanism especially useful in the context of concurrency. For example, in a concrete implementation, a scheduler will not be constrained by a particular ordering of these rules, allowing them to be interleaved or even assigned to different physical processors.

Another significant advantage of a graph-based approach is tool support, which allows the effect of the rewrite rules to be visualized and simulated, increasing confidence in the correctness of the specification.

The remainder of this paper is structured as follows: Sec. 2 first provides a brief overview of the original *delMDSOC* machine model. Next, Sec. 3 introduces the actor-based concurrency extension of the machine model, including its operational semantics. This extension will then be demonstrated by means of an example application in Sec. 4. Related work is presented in Sec. 5, while Sec. 6 concludes this paper and discusses potential directions for future work.

Email addresses: tim.molderez@ua.ac.be (Tim Molderez), hans.schippers@ua.ac.be (Hans Schippers), dirk.janssens@ua.ac.be (Dirk Janssens)

[1] http://www.hpi.uni-potsdam.de/hirschfeld/projects/delmdsoc/

2. A Machine Model for MDSOC

In this section we briefly describe the *delMDSOC* machine model originally introduced in [1]. Based on the notions of objects, messages and delegation, its original application domain is to serve as a compilation target for high-level languages dealing with the modularization of *crosscutting concerns*. Such concerns cannot be modularized in a program's dominant decomposition, and hence appear scattered throughout several unrelated modules. Examples of MDSOC paradigms trying to address this issue are *aspect-oriented programming* (AOP) [6] and *context-oriented programming* (COP) [7].

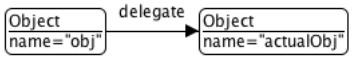

Figure 1: An object represented as a combination of a proxy and the actual object.

The model assumes a prototype-based object-oriented environment in which each object is visible to others via a *proxy* 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 Fig. 1. The model allows dynamic modification of this 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. An Actor-based Concurrent Machine Model

As explained in [3], the extension to our machine model involves the addition of explicit concurrency support, based on the well-known actor model of computation [4]. The essential entities in this model are *actors*, which do not share any mutable state and may only communicate with each other via *asynchronous message sending*, hence preventing many racing conditions and deadlocks prevalent in shared-state multithreaded models. Unlike the pure actor model, we make a distinction between actors and regular objects: Actors act as *containers* of regular objects. An object can send an asynchronous message to an object belonging to a different actor, resulting in this message being added to the back of the receiving 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 execution. When this happens, the message at the front of the mail queue is removed and pushed onto the *process stack*. Although the actor may at all times receive additional messages in its mail queue, it won't process them until it has dealt with the message that currently resides on the process stack. Executing this message 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 belonging to one and the same actor occurs synchronously, whereas communication involving two different actors occurs asynchronously.

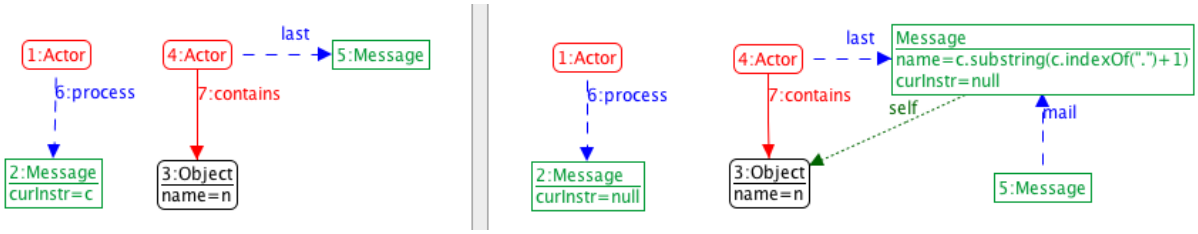

Figure 2: Graph rewrite rule for sending an (asynchronous) message to a different actor.

This behavior is described by an operational semantics in the form of a set of (single pushout) graph rewrite rules. Due to space restrictions it is impossible to present all of them in this paper, but an example rule, as it appears in the AGG tool [8], is displayed in Fig. 2. The rule represents the situation where the body of the current message on the process stack contains an instruction c which involves a message send to an object in a different actor with a non-empty mail queue. (The case with an empty queue is currently handled by a seperate rule.) The rule consists of a *left-hand side*

(LHS) and a *right-hand side* (RHS), separated by a gray bar. The LHS represents a pattern that should be matched in the current host graph. The RHS specifies the modifications that should be applied to the found match. Nodes that only appear on the LHS are removed; nodes that only appear on the RHS are added; nodes with the same number label in the LHS and the RHS simply remain in the host graph. On top of this, it is possible to modify attributes of some nodes in the RHS based on LHS information.

Two different actors (labeled 1 and 4) are present in the LHS of Fig. 2 : One with a message (labeled 2) currently residing on the process stack, and one containing the object (labeled 3) which is targeted by the current instruction `c` in the body of said message. It is important to understand that, when this situation occurs, the lookup process of message 2 has already finished, its corresponding message body has been found, and instruction `c` of this body needs processing and is of the form `n.msg`, with `n` the name of the target object and `msg` the name of the message which should be sent to it. The information about `c`'s format is not visible in Fig. 2, as it is specified by means of additional textual constraints. However, note that, in the LHS, the name of object 3 is indeed required to be equal to `n`.

The RHS shows how the current instruction is effectively processed: A new message is appended to the mail queue of actor 4, representing the asynchronous message send, with the `self` edge pointing to its target object. The current instruction in this message is also set to the desired message name. All other nodes remain untouched, except for the fact that the current instruction is now considered processed and hence the value of the `curInstr` attribute of message 2 is set to `null`.

Eventually, the new message will be associated with a message body when actor 4 decides to process it, all of which is described by different graph rewrite rules. Theoretically, all of these rules may be applied to the host graph at the same time (concurrently), transforming each match they find of their LHS into an instance of their RHS. In particular, one rule may be applied concurrently to multiple disjoint parts of the host graph; consequently, communication between actors can also be concurrent. It is important to understand that this does not necessarily mean there is no logical ordering of certain actions. For example, the rule performing message lookup should only be applied after the rule that removes a message from the mail queue in order to process it. This is achieved by ensuring that the former rule's LHS only matches when a message is already on the process stack. In practice, however, this can become a tedious process, hence it makes sense to impose a partial ordering on some rules, a mechanism which is built into AGG.

4. A Peer-to-peer Chat Application

This section discusses an example scenario which demonstrates the combination of the concurrency and MDSOC features in our model. The scenario involves a peer-to-peer chat application with three different peers: Alice, Bob and Charlie. Two chat sessions are currently active: Alice is having a conversation with Bob, whilst Bob is also having a separate conversation with Charlie. At some point in time, Alice wants to share some confidential information with Bob and decides to encrypt her chat session, resulting in two encryption proxy objects which are deployed at both ends of the communication channel. Of course, this encryption operation shouldn't affect the conversation between Bob and Charlie, which remains unencrypted.

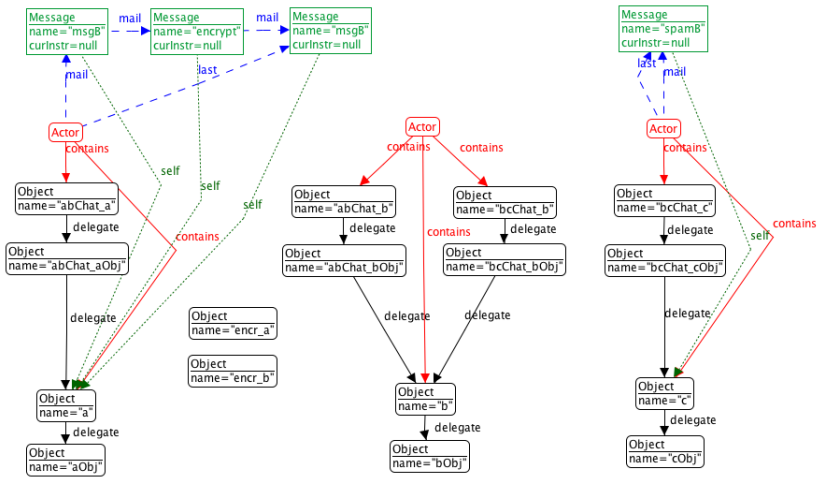

Figure 3: Initial object configuration.

Fig. 3 shows the initial object configuration of the whole scenario (Alice's, Bob's and Charlie's names are shortened to `a`, `b` and `c`). Three disjoint subgraphs, one for each peer, can be distinguished. They each consist of: an *actor* object (, indicating that they operate concurrently), a *client* (composite) object near the bottom capable of understanding chat messages and a composite object near the middle of the figure representing a *channel end*. Since Bob is active in two chat sessions, his actor contains two channel ends. Note that delegation not only takes place inside composite objects (i. e., between a proxy and its delegate), but also between a channel end and its corresponding client object. This is merely for practical reasons as it reduces the number of required message sends. Finally, there are two proxies `encr_a` and `encr_b`, which will at some point be inserted in the delegation chain of the channel end objects representing the channel between Alice and Bob.

The UML sequence diagram in Fig. 4 depicts how the conversation between Alice and Bob takes place over time. The exact scenario between Alice and Bob is as follows: Alice first sends a chat message to Bob, then enables encryption for both ends of their communication channel and finally sends the same chat message once more, this time encrypted. Sending a chat message takes place in several steps: First, `a` sends `msgB` to her own end of the communication channel, represented by `abChat_a`. Next, `abChat_a` sends an asynchronous message `msg` to the other end of the communication channel, `abChat_b`, which resides in a different actor. This message is then delegated to `b`, where it is understood and the corresponding message body is executed, resulting in the augmentation of a counter attribute `IMs`, which keeps track of the number of unencrypted chat messages.

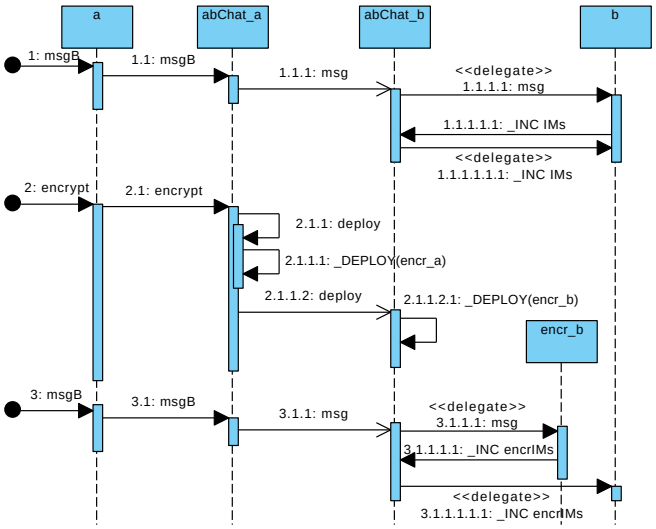

Figure 4: UML sequence diagram for the interaction between Alice and Bob.

Encrypting the channel is performed by deploying two proxies, `encr_a` and `encr_b`; one at each end of the communication channel. When the chat message is resent over the now encrypted channel, it will no longer reach `b`. Instead, it is intercepted in `encr_b`, where it is understood and hence a different message body is executed, resulting in the augmentation of a different counter `encrIMs`, one that keeps track of the number of encrypted chat messages.

5. Related Work

In [9], a graph-based operational semantics is provided for an aspect-oriented extension of Featherweight Java. Given the fact that the process involves a serialization step towards a set of low-level instructions, the graph rewrite rules, at first sight, are at a similar level of abstraction as the ones in this paper. However, the idea of a machine model as a target for a multitude of MDSOC paradigms isn't present. This is apparent, for example, in the fact that an explicit *proceed* stack is modeled, in order to accommodate for the corresponding high-level language construct. Moreover, as the original FJ semantics involves a sequentialization process, there is no support for concurrency.

There also are several high-level aspect-oriented languages that support pointcuts dealing with concurrency. For example, the *percflow* pointcut implies the creation of different aspect instances for each thread that passes through a particular control flow. The CaesarJ [10] language can dynamically activate an aspect for a particular block of code and limit the aspect's applicability to the threads that are busy executing that block of code. In [11], the *threadlocal* pointcut was introduced, which restricts the application of advice to a user-specified set of threads.

Finally, Rasche *et al.* [12] propose self-adaptive multithreaded applications as a suitable use case to demonstrate the value of aspect-oriented languages with support for both concurrency and dynamic deployment.

6. Conclusion and Future Work

This paper introduced an operational semantics of our extended delegation-based MDSOC machine model, based on graph rewrite rules. The extension to the model involves support for concurrency based on the actor model of computation. More specifically, we added an explicit notion of actors as containers of objects. All communication between objects belonging to different actors occurs asynchronously.

We also presented an example peer-to-peer chat application in which concurrency and MDSOC features are combined, illustrating the feasibility of our approach.

A number of issues are to be addressed in future work. Currently, a number of basic features, such as message parameters and cascading are still missing and should be added to our semantics. Moreover, we did not yet implement all of the ideas presented in [3]. For example, it may prove useful to introduce lightweight concurrency support within one actor, in the form of coroutines. This would allow for synchronous message sends between actors, which would involve pausing the sending coroutine, a mechanism which may also be used for the implementation of high-level mechanisms such as futures. Sending asynchronous messages within one actor is another feature which may be added.

At this point, we assume a global namespace in which every object knows all other objects by name. This is obviously unrealistic as well as undesirable, especially in the context of distributed applications. A mechanism needs to be devised in which every object (or actor) has an updatable set of acquaintances.

Another issue deals with object migration: It may be useful, if only for scheduling purposes, for objects to migrate from one actor to another. For example, if we assume that every actor corresponds to a different physical processor, it may be desirable to minimize inter-actor communication for performance reasons. As patterns of dense communication may change dynamically, objects would need to switch actors accordingly.

Eventually, we aim to use our semantics for the implementation of an actual MDSOC virtual machine, which may serve as a compilation target for a whole range of high-level MDSOC languages.

Acknowledgements

The authors are grateful to Tom Van Cutsem for fruitful discussions and comments related to introducing concurrency support in our machine model.

[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] H. Schippers, M. Haupt, R. Hirschfeld, An implementation substrate for languages composing modularized crosscutting concerns, in: Proceedings of the 2009 ACM symposium on Applied Computing, ACM, Honolulu, Hawaii, 2009, pp. 1944–1951.

[3] H. Schippers, T. V. Cutsem, S. Marr, M. Haupt, R. Hirschfeld, Towards an actor-based concurrent machine model, in: ICOOOLPS'09 workshop at ECOOP'09, 2009.

[4] G. Agha, Actors: a model of concurrent computation in distributed systems, MIT Press, 1986.

[5] G. Rozenberg (Ed.), Handbook of graph grammars and computing by graph transformation, Vol. 1-3, World Scientific Publishing Co., Inc., 1997.

[6] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. Loingtier, J. Irwin, Aspect-Oriented programming, in: M. Aksit, S. Matsuoka (Eds.), ECOOP '97: Object-Oriented Programming, Vol. 1241 of Lecture Notes in Computer Science, Springer, 1997, p. 220–242.

[7] R. Hirschfeld, P. Costanza, O. Nierstrasz, Context-oriented programming, Journal of Object Technology (JOT) 7 (3) (2008) 125–151.

[8] 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.

[9] 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.

[10] I. Aracic, V. Gasiunas, M. Mezini, K. Ostermann, An overview of CaesarJ, Transactions on AOSD LNCS 3880.

[11] T. Molderez, Supporting thread-local aspects in a virtual machine for modularized crosscutting concerns, Master's Thesis, University of Antwerp (2009).

[12] A. Rasche, W. Schult, A. Polze, Self-adaptive multithreaded applications: a case for dynamic aspect weaving, in: Proceedings of the 4th workshop on Reflective and adaptive middleware systems, ACM, Grenoble, France, 2005.