From SOS to Asynchronously Communicating Actors

Frank de Boer[1], Einar Broch Johnsen[2](✉), Ka I Pun[2,3], and Silvia Lizeth Tapia Tarifa[2]

[1] CWI, Amsterdam, The Netherlands
f.s.de.boer@cwi.nl
[2] Department of Informatics, University of Oslo, Oslo, Norway
{einarj,violet,sltarifa}@ifi.uio.no
[3] Western Norway University of Applied Sciences, Bergen, Norway

Abstract. Structural Operational Semantics (SOS) provides a general format to describe a model as a transition system with very powerful synchronization mechanisms. Actor systems are distributed, asynchronously communicating units of computation with encapsulated state, with much weaker means of synchronizing between actors. In this paper, we discuss an implementation of a SOS model using actors in the object-oriented actor language ABS and how to argue that global properties about the model are inherited from the SOS level to the actor implementation. The work stems from a case study modelling the memory system of a cache-coherent multicore architecture.

1 Introduction

Structural operational semantics (SOS) [1], introduced by Plotkin in 1981, describes system behavior as transition relations in a syntax-oriented, compositional way, using inference rules to capture local transitions and how these compose into transitions at the global level. Process synchronization in SOS rules is expressed abstractly using, e.g., assertions over system states and reachability conditions over transition relations as premises, and label synchronization for parallel transitions. This high level abstraction greatly simplifies the verification of system properties. In particular, reasoning about SOS semantics has been used to prove meta-properties for all instances of a model such as type preservation properties for the execution of programs in a programming language (e.g., [2]). In contrast, a direct implementation of an SOS model for the simulation of system behavior is less common, as execution quickly becomes a reachability problem with a lot of backtracking. Often, the implementation of an SOS model can be quite far from the transition rules of the model itself, and, as a result,

Supported by *SIRIUS: Centre for Scalable Data Access* (www.sirius-labs.no) and *ADAPt: Exploiting Abstract Data-Access Patterns for Better Data Locality in Parallel Processing* (www.mn.uio.no/ifi/english/research/projects/adapt/).

© Springer Nature Switzerland AG 2020
J. Camara and M. Steffen (Eds.): SEFM 2019 Workshops, LNCS 12226, pp. 269–275, 2020.
https://doi.org/10.1007/978-3-030-57506-9_20

we do not always know if the properties laboriously proven for the SOS model indeed also hold of its implementation.

We are interested in decentralized implementations of SOS models, to obtain efficient yet faithful realizations of these models, without unnecessary global synchronization and backtracking yet preserving the safety properties of the SOS model. For our implementations, we work with active object languages [3], which combine the scalable, asynchronous nature of actor languages with the code structuring mechanisms of object orientation. In particular, we target ABS [4] because it supports *cooperative scheduling*, which allows a simple yet expressive form of synchronization, and because it has a *formally defined semantics*, which allows us to study the preservation of safety properties in a formal setting.

This paper is an extended abstract of an invited talk given at FOCLASA 2019. Further details of the ideas discussed in this paper may be found in [5,6] and the source of the original SOS model which triggered our interest in this line of investigation may be found in [7].

2 Background

2.1 SOS

Structural operational semantics (SOS) [1] define the meaning of programs by (labelled) transition systems and simple operations on data. Programs are defined syntactically by a grammar and execute in a (local) context. Let us assume that these contexts resemble objects, such that programs (or sequences of actions) execute on a local state and exchange messages or synchronize with each other in the transition rules. If P and Q are such programs in local contexts, let $P||Q$ denote the configuration which consists of P and Q executing in parallel. The transition rules then have formats such as

$$\frac{\textit{condition on } P}{P \to P'}\ (\text{Local}) \qquad \frac{\textit{condition on } P}{P \to P'||Q}\ (\text{AsyncSend}) \qquad \frac{\textit{condition on } P \textit{ and } Q}{P||Q \to P'}\ (\text{AsyncReceive})$$

$$\frac{\textit{condition on } P \textit{ and } Q}{P||Q \to P'||Q'}\ (\text{Handshake}) \qquad \frac{P \to P'}{P||Q \to P'||Q}\ (\text{Context}) \qquad \frac{P \xrightarrow{l} P' \quad Q \xrightarrow{\bar{l}} Q'}{P||Q \to P'||Q'}\ (\text{labelsync})$$

Compared to decentralized systems such as actors, the premises of the rules AsyncReceive and Handshake contain applicability conditions on *both* P and Q and LabelSync introduces synchronization over events l (where $\bar{l}$ denotes the dual of l). These forms of synchronization are difficult to express in the asynchronous setting. Conditions further include *reachability* expressions, captured here by transitions in the premises of the rules Context and LabelSync.

2.2 ABS

ABS is a modelling language for designing, verifying, and executing concurrent software [4]. The language combines the syntax and object-oriented style of Java with the Actor model of concurrency [8] into active objects which decouple communication and synchronization using asynchronous method calls, futures and cooperative scheduling [3]. Although only one thread of control can execute in an active object at any time, cooperative scheduling allows different threads to interleave at explicitly declared points in the code. Access to an object's fields is encapsulated, thus, any non-local (outside of the object) read or write to fields must happen explicitly via asynchronous method calls so as to mitigate race-conditions or the need for mutual exclusion (locks).

```
class Lock {
  Bool unlocked = True;
  Unit take_lock{await unlocked; unlocked = False;}
  Unit release_lock{unlocked = True;} }
```

Fig. 1. Lock implementation in ABS using await on Booleans.

We explain the basic mechanism of asynchronous method calls and cooperative scheduling in ABS by the simple code example of a class `Lock`. First, the execution of a statement `res = await o!m(args)` consists of storing a message `m(args)` corresponding to the asynchronous call to the message pool of the callee object o. This **await** statement *releases the control* of the caller until the return value of that method has been received. Releasing the control means that the caller can execute other messages from its own message pool in the meantime. ABS supports the shorthand `o.m(args)` to make an asynchronous call `f=o!m(args)` followed by the operation `f.get` which *blocks* the caller object (does not release control) until the future `f` has received the return value from the call. As a special case the statement `this.m(args)` models a self-call, which corresponds to a standard subroutine call and avoids this blocking mechanism. The code in Fig. 1 illustrates the use of the **await** statement on a Boolean condition to model a binary semaphore, which can be used to enforce exclusive access to a communication medium such as a channel. Thus, the statement `await channel!take_lock()` will suspend the calling method invocation (and release control in the caller object) and can first resume when the generated invocation of the method `take_lock` returns, which can only happen when the local condition `unlocked` (of the channel) has become true.

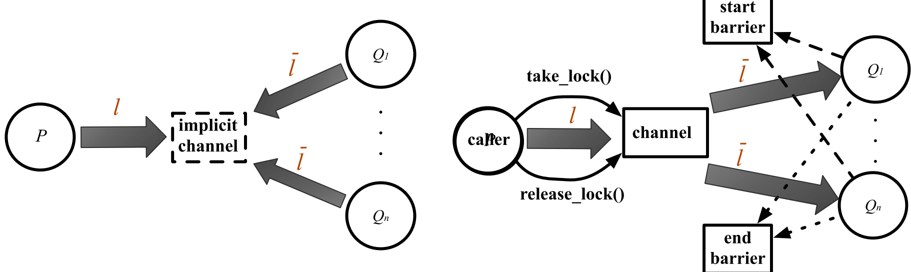
3 Example of a SOS Synchronization Pattern

$$\text{(LocalSend)}\ \frac{\textit{conditions on } P}{P \xrightarrow{l} P'} \qquad \text{(LocalReceive)}\ \frac{\textit{conditions on } Q}{Q \xrightarrow{\bar{l}} Q'}$$

$$\text{(GlobalSync)}\ \frac{P \xrightarrow{l} P' \quad Q_i \xrightarrow{\bar{l}} Q'_i \textit{ for } 0 < i \leqslant n}{P||Q_1||\dots||Q_n \rightarrow P'||Q'_1||\dots||Q'_n}$$

Fig. 2. Multiparty synchronization in SOS.

We illustrate the problem of implementing SOS rules by considering *multiparty label synchronization*, inspired by the multicore memory model of Bijo *et al.* [7], where bus



(a) State machine of the global synchronization using labels in the SOS model.

(b) State machine of the global synchronization using a bus and barriers in the ABS model.

Fig. 3. Label synchronization in SOS vs barrier synchronization in ABS. In the SOS model (a), circles represent synchronized entities and shaded arrows labelled transitions. Note that the synchronization channel is *implicit* in the SOS model, as synchronization is captured by label matching. In the ABS model (b), circles represent the same nodes as in the SOS model, shaded arrows method invocations, solid arrows mutual access to the synchronization channel and dotted arrows barrier synchronizations.

synchronization is a label matching problem such that an invalidation request for a cache line succeeds when the cache line has been invalidated in all other caches. Somewhat simplified, this problem corresponds to the SOS rules in Fig. 2, in which n objects synchronize on a broadcast from P to Q_i (where $0 < i \leqslant n$) and both sender and receivers have local synchronization conditions denoted *conditions1* and *conditions2*, respectively.

The synchronization problem corresponding to these SOS rules can be illustrated by the state machine in Fig. 3a. However, in the input-enabled ABS system, we need to ensure that only one object can send on the synchronization channel at any time, using a *lock* such as the one in Fig. 1. Then, a physical synchronization channel forwards the synchronization event to all receiving objects. To receive the synchronization event, all readers need to make a transition simultaneously, Hence, the implementation needs to introduce a *start barrier*. The bus can only return the success to the sender of the communication event once all receivers have completed their transition. This corresponds to an *end barrier* synchronizing on the success of the transitions of all receivers, after which the send-method can return and the synchronization channel can be unlocked. The corresponding synchronization code in ABS is illustrated in Fig. 3b.

The correctness of the decentralized active object implementation of the SOS model can then be addressed by a simulation relation between the ABS code and the transitions of the SOS model. This approach is based on the notion of *stable points* in the execution of ABS programs [5], at which an object requires external input to make progress (either an event or a scheduling decision). The semantics of ABS then allows us to prove that executions are *globally confluent* at the granularity of stable points [5,6]. Consequently, it is sufficient to reason about one object

at a time between stable points in the program execution. These stable points are syntactically defined on the ABS code, and the abstraction relation between the ABS code and the SOS model need only to hold at the stable points. Thus, we can reason about the transitions between stable points in the ABS code and the corresponding transitions in the SOS model. Furthermore, if the scheduling at stable points is deterministic in the ABS model, two transitions can be merged, further reducing the number of cases that need to be considered [5].

4 Related Work

There is in general a significant gap between a formal model and its implementation [9]. SOS [1] succinctly formalizes operational models and are well-suited for proofs, but direct implementations of SOS quickly lead to very inefficient implementations. Executable semantic frameworks such as Redex [10], rewriting logic [11], and $\mathbb{K}$ [12] reduce this gap, and offer executable formal models of complex languages like C and Java. The relationship between SOS and rewriting logic semantics has been studied [13] without proposing a general solution for label matching. Bijo et al. implemented their SOS multicore memory model [14] in Maude [15] using an orchestrator for label matching, but do not provide a correctness proof wrt. the SOS model. Different semantic styles can be modelled and related inside one framework; for example, the correctness of distributed implementations of KLAIM systems in terms of simulation relations have been studied in rewriting logic [16]. Compared to these works on semantics, our focus here is on implementing an SOS model in a distributed active object setting in a way which allows formal proofs of correctness for this implementation.

Correctness-preserving compilation is related to correctness proofs for implementations, and ensures that low-level representations of a program preserve the properties of the high-level model. Examples here include type-preserving translations into typed assembly languages [17] and formally verified compilers [18]; the latter proves the semantic preservation of a compiler from C to assembler code, but leaves shared-variable concurrency for future work. In contrast to work which studies compilation from one language to another, our work focuses on a specific model and its implementation and specifically targets parallel systems.

5 Conclusion

We have outlined a methodology for the decentralized implementation of SOS models, targeting the active object language ABS. A challenge for this methodology is to correctly implement the synchronization patterns of the SOS rules, which may cross encapsulation borders in the active objects, and in particular label synchronization on parallel transitions steps. To address this problem, we exploit that ABS allows for a globally confluent coarse-grained semantics.

References

1. Plotkin, G.D.: A structural approach to operational semantics. J. Log. Algebr. Program. **60–61**, 17–139 (2004)
2. Igarashi, A., Pierce, B.C., Wadler, P.: Featherweight Java: a minimal core calculus for Java and GJ. ACM Trans. Program. Lang. Syst. **23**(3), 396–450 (2001)
3. Boer, F.D., et al.: A survey of active object languages. ACM Comput. Surv. **50**(5), 76:1–76:39 (2017)
4. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: a core language for abstract behavioral specification. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) FMCO 2010. LNCS, vol. 6957, pp. 142–164. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25271-6_8
5. Bezirgiannis, N., de Boer, F., Johnsen, E.B., Pun, K.I., Tapia Tarifa, S.L.: Implementing SOS with active objects: a case study of a multicore memory system. In: Hähnle, R., van der Aalst, W. (eds.) FASE 2019. LNCS, vol. 11424, pp. 332–350. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-16722-6_20
6. Tveito, L., Johnsen, E.B., Schlatte, R.: Global reproducibility through local control for distributed active objects. In: Wehrheim, H., Cabot, J. (eds.) FASE 2020. LNCS, vol. 12076, pp. 140–160. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45234-6_7
7. Bijo, S., Johnsen, E.B., Pun, K.I., Tapia Tarifa, S.L.: A formal model of data access for multicore architectures with multilevel caches. Sci. Comput. Program. **179**, 24–53 (2019)
8. Hewitt, C., Bishop, P., Steiger, R.: A universal modular ACTOR formalism for artificial intelligence. In: Proceedings of the 3rd International Joint Conference on Artificial Intelligence. IJCAI 1973, pp. 235–245. Morgan Kaufmann Publishers Inc. (1973)
9. Schlatte, R., Johnsen, E.B., Mauro, J., Tapia Tarifa, S.L., Yu, I.C.: Release the beasts: when formal methods meet real world data. In: de Boer, F., Bonsangue, M., Rutten, J. (eds.) It's All About Coordination. LNCS, vol. 10865, pp. 107–121. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-90089-6_8
10. Felleisen, M., Findler, R.B., Flatt, M.: Semantics Engineering with PLT Redex. The MIT Press, Cambridge (2009)
11. Meseguer, J., Rosu, G.: The rewriting logic semantics project. Theor. Comput. Sci. **373**(3), 213–237 (2007)
12. Rosu, G.: $\mathbb{K}$: a semantic framework for programming languages and formal analysis tools. In: Dependable Software Systems Engineering, pp. 186–206. IOS Press (2017)
13. Serbanuta, T., Rosu, G., Meseguer, J.: A rewriting logic approach to operational semantics. Inf. Comput. **207**(2), 305–340 (2009)
14. Bijo, S., Johnsen, E.B., Pun, K.I., Tapia Tarifa, S.L.: A Maude framework for cache coherent multicore architectures. In: Lucanu, D. (ed.) WRLA 2016. LNCS, vol. 9942, pp. 47–63. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-44802-2_3
15. Clavel, M., et al. (eds.): All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic. LNCS, vol. 4350. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71999-1
16. Eckhardt, J., Mühlbauer, T., Meseguer, J., Wirsing, M.: Semantics, distributed implementation, and formal analysis of KLAIM models in Maude. Sci. Comput. Program. **99**, 24–74 (2015)

17. Morrisett, J.G., Walker, D., Crary, K., Glew, N.: From system F to typed assembly language. ACM Trans. Program. Lang. Syst. **21**(3), 527–568 (1999)
18. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM **52**(7), 107–115 (2009)