Implementing SOS with Active Objects: A Case Study of a Multicore Memory System*

Nikolaos Bezirgiannis[1], Frank de Boer[1], Einar Broch Johnsen[2], Ka I Pun[2,3], and S. Lizeth Tapia Tarifa[2]

[1] CWI, Amsterdam, the Netherlands
{n.bezirgiannis,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. This paper describes the development of a parallel simulator of a multicore memory system from a model formalized as a structural operational semantics (SOS). Our implementation uses the Abstract Behavioral Specification (ABS) language, an executable, active object modelling language with a formal semantics, targeting distributed systems. We develop general design patterns in ABS for implementing SOS, and describe their application to the SOS model of multicore memory systems. We show how these patterns allow a formal correctness proof that the implementation simulates the formal operational model and discuss further parallelization and fairness of the simulator.

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 for local transitions and their composition. 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 of abstraction greatly simplifies the verification of system properties, but not the simulation of system behavior as execution quickly becomes a reachability problem with a lot of backtracking. In this paper, we study how to implement a parallel simulator with a formal correctness proof from a SOS model, in terms of a case study of a multicore memory system. Such a correctness proof requires that the implementation language is also defined formally by an operational semantics.

A major challenge in software engineering is the exploitation of the computational power of multicore (and manycore) architectures. One important aspect of this challenge is the memory systems of these architectures. These memory systems generally use caches to avoid bottlenecks in data access from main memory, but caches introduce data duplication and require protocols to ensure coherence. Although data duplication

* 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/).

is usually not visible to the programmer, the way a program interacts with these copies largely affects performance by moving data around to maintain coherence. To develop, test and optimize software for multicore architectures, we need correct, executable models of the underlying memory systems. A SOS model of multicore memory systems with correctness proofs for cache coherency has been described in [2], together with a prototype implementation in the rewriting logic system Maude [3]. However, this fairly direct implementation of the SOS model is not well suited to simulate large systems.

This paper considers an implementation of the SOS model in ABS [4], a language tailored to the description of distributed systems based on active objects [5]. ABS is formally defined by an operational semantics and supports parallel execution on backends in Erlang, Haskell, and Java. The following features of ABS allow a high-level, coarse-grained view of the execution of different method invocations by different active objects: encapsulation of local state in active objects, communication using asynchronous method calls and futures, and cooperative scheduling of the method invocations of an active object. Our case study fully exploits these features and the resulting abstractions to correctly implement the complex process synchronization of the original SOS model.

The main contributions of this paper are as follows:

– We provide general design patterns in ABS for implementing structural operational semantics with active objects, and apply these patterns to the implementation in ABS of a structural operational semantics of multicore memory systems.
– We show how these patterns allow a formal correctness proof of this implementation by means of a simulation relation between the formal operational semantics of the ABS implementation and the operational model of multicore memory systems.
– We discuss how these ABS design patterns can be used to further parallelize the implementation while preserving correctness.
– Finally, we show how the ABS modeling concepts of symbolic time and virtual resources can be used to obtain a parallel implementation of the SOS model which abstractly ensures fairness between the progress of different parallel components, independently of the number of cores that are used in the simulation.

2 An Abstract Model of a Multicore Memory System

Design decisions for a program running on top of a multicore memory systems can be explored using simulators based on abstract models. Bijo et al. [2,6] developed a model which takes as input tasks (expressed as data access) to be executed, the corresponding data layout in main memory (indicating where data is allocated), and a parallel architecture consisting of cores with private multi-level caches and shared memory (see Fig. 1). Additionally, the model is configurable in the number of cores, the number and size of caches, and the associativity and replacement policy. Memory is organized in blocks which move between caches and main memory. For simplicity, the model assumes that the size of cache lines and memory blocks in main memory coincide, abstracts from the data content of memory blocks, and transfers memory blocks from the caches of one core to the caches of another core via main memory.

Tasks from the program are scheduled for execution from a shared task pool. Task execution on a core requires memory blocks to be transferred from main memory to the closest cache. Each cache has a pool of fetch/flush instructions to move blocks among caches and between caches and main memory. Consistency between multiple copies of a memory block is ensured using the standard cache coherence protocol MSI (e.g., [7]), with which a cache line is either modified, shared or invalid. A *modified* cache line has the most recent value of the memory block, therefore all other copies are *invalid* (including the one in main memory). A *shared* cache line indicates that all copies of the block are consistent. The protocol's messages are broadcast to the cores. The details of the broadcast (e.g., on a mesh or a ring) can be abstracted into an *abstract communication medium*. Following standard nomenclature, *Rd* messages request *read* access and *RdX* messages *read exclusive* access to a memory block. The latter invalidates other copies of the same block in other caches to provide write access.

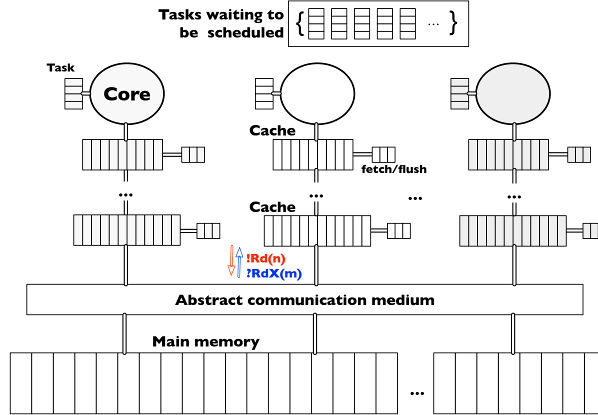


Fig. 1: Abstract model of a multicore memory system.

To access data from a block n, a core looks for n in its local caches. If n is not found in shared or modified state, a *read request* $!Rd(n)$ is broadcast to the other cores and to main memory. The cache can *fetch* the block when it is available in main memory. Eviction is required if the cache is full. Writing to block n requires n to be in shared or modified state in the local cache; if it is in shared state, an *invalidation request* $!RdX(n)$ is broadcast to obtain exclusive access. If a cache with block n in modified state receives a read request $?Rd(n)$, it *flushes* the block to main memory; if a cache with block n in shared state receives an invalidation request $?RdX(n)$, the cache line will be *invalidated*; the requests are discarded otherwise. Read and invalidation requests are broadcast instantaneously in the abstract model, reflecting that signalling on the communication medium is order of magnitude faster than moving data to or from main memory.

2.1 Formalization of the Multicore Memory System as an SOS Model

An operational meaning for the abstract model described above has be defined using structural operational semantics (SOS) [1] with labeled transitions to model broadcast in the abstract communication medium. The resulting formalization [2, 6] is shown to guarantee standard correctness properties for data consistency and cache coherence from the literature [8, 9], including the preservation of program order in each core, the absence of data races, and no access to stale data. We briefly outline the main aspects of the formal model. The runtime syntax is given in Fig. 2. A configuration cf consists of main memory M, cores $\overline{CR}$, caches $\overline{Ca}$, and tasks $\overline{dap}$ to be scheduled. (We syntactically

Syntactic categories.	***Definitions.***	
$cid \in CoreId$	$cf \in Config$	$::= M \circ \overline{dap} \circ \overline{Ca} \circ \overline{CR}$
$caid \in CacheId$	$CR \in Core$	$::= cid \bullet rst$
$n \in Address$	$Ca \in Cache$	$::= caid \bullet M \bullet dst$
$r \in Ref$	$st \in Status$	$::= \{mo, sh, inv\}$
	$dap \in AccessPtns$	$::= \varepsilon \mid dap;dap \mid \mathbf{read}(r) \mid \mathbf{write}(r) \mid \mathbf{commit}(r)$
		$\mid \mathbf{commit} \mid dap \sqcap dap \mid dap^{*} \mid \mathbf{skip} \mid \mathbf{spawn}(dap)$
	$rst \in RunLang$	$::= dap \mid rst;rst \mid \mathbf{readBl}(r) \mid \mathbf{writeBl}(r)$
	$dst \in DataLang$	$::= \varepsilon \mid \overline{dst} \mid \mathbf{fetch}(n) \mid \mathbf{flush}(n) \mid \mathbf{fetchBl}(n) \mid \mathbf{flush}$

Fig. 2: Syntax of runtime configurations, where over-bar denotes sets (e.g., $\overline{CR}$).

abuse set operations for multisets, including union $\cup$ and subtraction $\setminus$.) A core $cid \bullet rst$ with identifier *cid* executes runtime statements *rst*. A cache with identifier *caid* has a local cache memory M and data instructions *dst*. We assume that *caid* encodes the *cid* of the core to which the cache belongs and its level in the cache hierarchy. We denote by $Status \cup \{\bot\}$ the extension of the set of status tags with the undefined value $\bot$. Thus, a memory $M : Address \rightarrow Status \cup \{\bot\}$ maps addresses n to either a status tags *Status* or to $\bot$ if the memory block with address n is not found in M.

Data access patterns dap model tasks consisting of $\mathbf{read}(r)$ and $\mathbf{write}(r)$ operations to references r and control flow operations for sequential composition $dap_1;dap_2$, non-deterministic choice $dap_1 \sqcap dap_2$, repetition dap^{*}, task creation $\mathbf{spawn}(dap)$, and **commit** which flushes the entire cache after task execution. The empty access pattern is denoted ε. Cores execute *runtime statements rst*, which extend *dap* with $\mathbf{readBl}(r)$ and $\mathbf{writeBl}(r)$ to block execution while waiting for data. Caches execute *data instructions dst* to fetch and flush the memory block with address n, here $\mathbf{fetchBl}(n)$ blocks execution while waiting for data, and **flush** flushes the entire cache.

The *abstract communication medium* allows messages from one cache to be transmitted to the other caches and to main memory in a parallel instantaneous broadcast. Communication in the abstract communication medium is formalized in terms of label matching on transitions. The formal syntax for this label mechanism is as follows:

$$S ::= !Rd(n) \mid !RdX(n) \qquad\qquad R ::= ?Rd(n) \mid ?RdX(n)$$

Here, for any address n, a request of the form $!Rd(n)$ or $!RdX(n)$ is sent by one node and its dual of the form $dual(!Rd(n)) = ?Rd(n)$ or $dual(!RdX(n)) = ?RdX(n)$ is broadcast to the rest of nodes and main memory. The syntax of the model is further detailed in [2,6].

2.2 Local and Global SOS Rules

The semantics is divided into local and global rules. Local rules capture interaction inside a node containing a core and the hierarchy of caches. Global rules capture synchronization and coordination between different nodes and main memory. In an *initial* configuration cf_0, all blocks in main memory M have status *sh*, all cores are idle, all caches are empty, and the task pool in $\overline{dap}$ has a single task representing the main block

of a program. Let $cf \xrightarrow{*} cf'$ denote an execution starting from cf and reaching cf' by applying global transition rules, which in turn apply local transition rules for each core and its cache hierarchy. In the rules, let the auxiliary function $addr(r)$ return the address n of the block containing reference r, $cid(caid)$ the identity of the core associated with cache $caid$, $lid(caid)$ the cache level of $caid$, and $status(M,n)$ the status of block n in map M. Let the predicate $first(caid)$ hold when $caid$ is the first level and $last(caid)$ when $caid$ is the last level cache. Note that unlabelled transitions $\rightarrow$ can be executed asynchronously, while labelled transitions $\xrightarrow{S}$ require synchronization between all the nodes and main memory (see Figs. 3 and 4). We discuss some representative rules for local and global level of the SOS model. The full SOS formalization can be found in [6].

Local semantics. The first rules of Fig. 3 involve a core and its first level cache. In PRRD_1, reading reference r succeeds if the block containing r is available. Otherwise, in PRRD_2 a **fetch**(n) instruction is added to the data instructions dst of the first level cache and further execution of the core is blocked by **readBl**(r). Writing to r only succeeds if the associated memory block has mo status in the first level cache. If the cache line is shared, the core broadcasts a $!RdX(n)$ request to acquire exclusive access, where the broadcast appears as a label on the transition in PRWR_2. Otherwise, the block must be fetched from main memory in PRWR_3 and **writeBl**(r) blocks execution.

For the remaining rules of Fig. 3, LC-HIT_1 and LC-MISS_1 capture interactions between adjacent levels of caches, and LCC-MISS_1 local state change in a cache line. If cache $caid_i$ needs a block n that is sh or mo in the next level cache, the address where block n should be placed is decided by a function $select(M_i,n)$ which reflects the cache associativity and the replacement policy. If eviction is needed, block n in $caid_j$ will be swapped with the selected block in $caid_i$ in LC-HIT_1. LC-MISS_1 shows how **fetch**(n)-instructions propagate to lower cache levels: **fetch**(n) is replaced by **fetchBl**(n) in $caid_i$ and added to the data instructions in $caid_j$. If the block cannot be found in any local cache, we have a *cache miss*: Execution is blocked by **fetchBl**(n) and a read request $!Rd(n)$ is broadcast, represented by the label in LLC-MISS_1.

Global semantics. The global rules synchronize the cache hierarchies of different cores and main memory, and ensures coherence. Selected global rules are given in Fig. 4. Rule SYNCH_1 captures a global step with synchronization on a label S, which can be either $!Rd(n)$ or $!RdX(n)$. The request will be broadcast to other caches. To maintain data consistency, these caches must process the requests at the same time. The receiving label R is the *dual* of S. For synchronization, the transition is decomposed into a premise for main memory with label R and another premise for the caches with label S. Rule SYNCH_2 distributes the receiving label to caches $\overline{Ca}_2$, which do not belong to the cache hierarchy of the sender core CR_1. The predicate $belongs(\overline{Ca},\overline{CR})$ expresses that any cache in $\overline{Ca}$ belongs to exactly one core in $\overline{CR}$. Rule ASYNCH captures parallel transitions without label. These transitions can be local to individual nodes and caches, parallel memory accesses, or the parallel spawning and scheduling of new tasks.

3 The ABS Model of the Multicore Memory System

In this section we outline the translation of the formal model into an executable object-oriented model using the ABS modeling language. We first briefly introduce the lan-

(PRRD$_1$)

$$\frac{n = addr(r) \quad first(caid) = true \qquad cid(caid) = c \quad status(M,n) \in \{sh, mo\}}{(caid \bullet M \bullet \overline{dst}) \circ (c \bullet \texttt{read}(r); rst) \rightarrow (caid \bullet M \bullet \overline{dst}) \circ (c \bullet rst)}$$

(PRRD$_2$)

$$\frac{n = addr(r) \quad first(caid) = true \qquad cid(caid) = c \quad status(M,n) \in \{inv, \bot\}}{(caid \bullet M \bullet \overline{dst}) \circ (c \bullet \texttt{read}(r); rst) \rightarrow (caid \bullet M[n \mapsto \bot] \bullet \overline{dst} \cup \{\texttt{fetch}(n)\}) \circ (c \bullet \texttt{readBl}(r); rst)}$$

(PRWR$_2$)

$$\frac{n = addr(r) \quad first(caid) = true \qquad cid(caid) = c \quad status(M,n) = sh}{(caid \bullet M \bullet \overline{dst}) \circ (c \bullet \texttt{write}(r); rst) \xrightarrow{!RdX(n)} (caid \bullet M[n \mapsto mo] \bullet \overline{dst}) \circ (c \bullet rst)}$$

(PRWR$_3$)

$$\frac{n = addr(r) \quad first(caid) = true \qquad cid(caid) = c \quad status(M,n) \in \{inv, \bot\}}{(caid \bullet M \bullet \overline{dst}) \circ (c \bullet \texttt{write}(r); rst) \rightarrow (caid \bullet M[n \mapsto \bot] \bullet \overline{dst} \cup \{\texttt{fetch}(n)\}) \circ (c \bullet \texttt{writeBl}(r); rst)}$$

(LC-HIT$_1$)

$$\frac{status(M_i,n_i) = s_i \quad status(M_j,n) = s_j \quad s_j \in \{sh, mo\} \qquad lid(caid_j) = lid(caid_i) + 1 \quad cid(caid_i) = cid(caid_j) \quad select(M_i,n) = n_i}{(caid_i \bullet M_i \bullet \overline{dst}_i \cup \{\texttt{fetch}(n)\}) \circ (caid_j \bullet M_j \bullet \overline{dst}_j) \rightarrow (caid_i \bullet M_i[n_i \mapsto \bot, n \mapsto s_j] \bullet \overline{dst}_i) \circ (caid_j \bullet M_j[n \mapsto \bot, n_i \mapsto s_i] \bullet \overline{dst}_j)}$$

(LC-MISS$_1$)

$$\frac{lid(caid_j) = lid(caid_i) + 1 \quad cid(caid_i) = cid(caid_j) \quad status(M_j,n) \in \{inv, \bot\}}{(caid_i \bullet M_i \bullet \overline{dst}_i \cup \{\texttt{fetch}(n)\}) \circ (caid_j \bullet M_j \bullet \overline{dst}_j) \rightarrow (caid_i \bullet M_i \bullet \overline{dst}_i \cup \{\texttt{fetchBl}(n)\}) \circ (caid_j \bullet M_j[n \mapsto \bot] \bullet \overline{dst}_j \cup \{\texttt{fetch}(n)\})}$$

(LLC-MISS$_1$)

$$\frac{last(caid) = true \quad status(M,n) \in \{inv, \bot\}}{(caid \bullet M \bullet \overline{dst} \cup \{\texttt{fetch}(n)\}) \xrightarrow{!Rd(n)} (caid \bullet M[n \mapsto \bot] \bullet \overline{dst} \cup \{\texttt{fetchBl}(n)\})}$$

Fig. 3: Local transition rules.

(SYNCH$_1$)

$$\frac{S \neq \emptyset \quad R = dual(S) \qquad M \xrightarrow{R} M' \quad \overline{Ca} \circ \overline{CR} \xrightarrow{S} \overline{Ca'} \circ \overline{CR'}}{M \circ \overline{dap} \circ \overline{Ca} \circ \overline{CR} \rightarrow M' \circ \overline{dap} \circ \overline{Ca'} \circ \overline{CR'}}$$

(SYNCH$_2$)

$$\frac{\begin{array}{c} \overline{CR} = \{CR_1\} \uplus \overline{CR_2} \quad \overline{Ca} = \overline{Ca_1} \uplus \overline{Ca_2} \\ belongs(\overline{Ca_1}, \{CR_1\}) \quad belongs(\overline{Ca_2}, \overline{CR_2}) \quad R = dual(S) \\ \overline{Ca_1} \circ CR_1 \xrightarrow{S} \overline{Ca_1'} \circ CR_1' \quad \overline{Ca_2} \xrightarrow{R} \overline{Ca_2'} \\ \overline{CR'} = \{CR_1'\} \cup \overline{CR_2} \quad \overline{Ca'} = \overline{Ca_1'} \cup \overline{Ca_2'} \end{array}}{\overline{Ca} \circ \overline{CR} \xrightarrow{S} \overline{Ca'} \circ \overline{CR'}}$$

(ASYNCH)

$$\frac{\begin{array}{c} \overline{CR} = \overline{CR_1} \uplus \overline{CR_2} \uplus \overline{CR_3} \quad \overline{Ca} = \overline{Ca_1} \uplus \overline{Ca_2} \uplus \overline{Ca_3} \uplus \overline{Ca_4} \quad belongs(\overline{Ca_3}, \overline{CR_3}) \\ M \circ \overline{Ca_1} \rightarrow M' \circ \overline{Ca_1'} \quad \overline{Ca_2} \rightarrow \overline{Ca_2'} \quad \overline{dap} \circ \overline{CR_2} \rightarrow \overline{dap'} \circ \overline{CR_2'} \quad \overline{Ca_3} \circ \overline{CR_3} \rightarrow \overline{Ca_3'} \circ \overline{CR_3'} \\ \overline{CR'} = \overline{CR_1} \cup \overline{CR_2'} \cup \overline{CR_3'} \quad \overline{Ca'} = \overline{Ca_1'} \cup \overline{Ca_2'} \cup \overline{Ca_3'} \cup \overline{Ca_4} \end{array}}{M \circ \overline{dap} \circ \overline{Ca} \circ \overline{CR} \rightarrow M' \circ \overline{dap'} \circ \overline{Ca'} \circ \overline{CR'}}$$

Fig. 4: Global transition rules.

guage and later explain the structural and behavioural correspondence between these two models, with a focus on the main challenges.

3.1 The ABS Language

ABS is a modeling 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 [10] into active objects which decouple communication and synchronization using asynchronous method calls, futures and cooperative scheduling [5] . 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, so 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).

We explain the basic mechanism of asynchronous method calls and cooperative scheduling in ABS by the simple code example of a class Bus. 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. 5 illustrates the use of the **await** statement on a Boolean condition to model a binary semaphore, which is used to enforce exclusive access to a communication medium implemented as a "bus". Thus, the statement **await** bus!lock_bus() will suspend the calling method invocation (and release control in the caller object) and will be resumed when the generated invocation of the method lock_bus of the "bus" itself has been resumed when the local condition unlocked (of the "bus") has become true.

```
class Bus {
  Bool unlocked = True;
  Unit lock_bus{await unlocked; unlocked = False;}
  Unit release_bus{unlocked = True;} }
```

Fig. 5: Bus lock implementation in ABS using await on Booleans.

3.2 The Structural View

The runtime syntax of the SOS is represented by ABS classes, as outlined in Fig. 6. We briefly overview the translation. In ABS, object identifiers guarantee unique names and object references are used to capture how cores and caches are related. These references are encoded in a one-to-one correspondence with the naming scheme of the SOS.

A core $cid \bullet rst$ is translated into a class Core with a a field currentTask representing the current task rst. Each core holds a reference to the first level cache. A cache memory $caid \bullet M \bullet dst$ is translated into a class Cache with an interface ICache and a class parameter nextLevel. In a cache, nextLevel holds a reference to the next level cache. If this reference is Nothing, it is last level cache (in the SOS, a predicate *last* is used to identify the last level). The field cacheMemory models the cache's memory M in SOS. The process pool of each cache object in ABS represents the data instruction set dst.

An ABS configuration consists of a number of cores with their corresponding cache hierarchies, the main memory, a scheduler with tasks waiting to be scheduled, and the ABS classes Bus and Barrier, which model the abstract communication medium and the

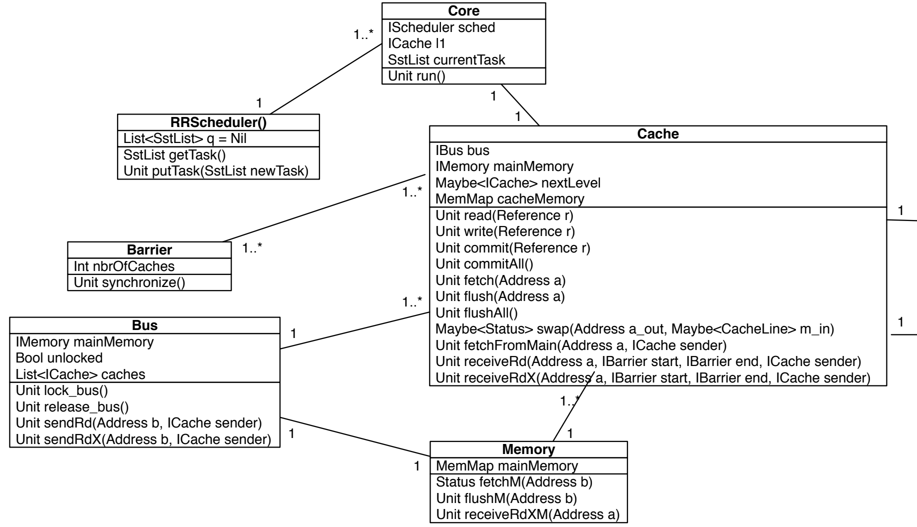


Fig. 6: Class diagram of the ABS model.

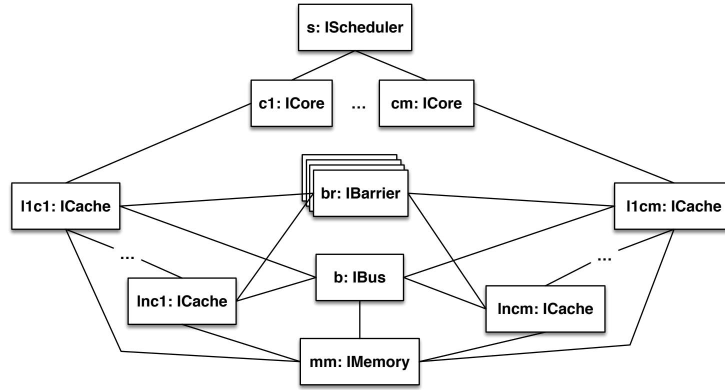


Fig. 7: Object diagram of an initial configuration.

global synchronization with labels $!Rd(n)$ and $!RdX(n)$ in the SOS. The object diagram in Fig. 7 shows an initial configuration corresponding to the one depicted in Fig. 1.

3.3 The Behavioral View

We discuss in this section the design patterns in ABS that implement the synchronization inherent in the SOS model. We observe here that the combination of asynchronous method calls and cooperative scheduling is crucial because of the *multitasking* inherent in the SOS model, which requires that objects need to be able to process other requests; e.g., caches need to flush memory blocks while waiting for a fetch to succeed.

```
Just(nextCache) => {
  Maybe<Status> s = Nothing;
  Maybe<CacheLine> selected = Nothing;
  while (s == Nothing) {
    retValue = await nextCache!fetch(n);
    selected = select(cacheMemory, maxSize, n);
    s = nextCache.swap(n,selected,name);
  }
  case selected {
    Nothing => skip;
    Just(Pair(n1,_)) => cacheMemory = removeKey(cacheMemory,n1);
  }
  cacheMemory = put(cacheMemory, n, fromJust(s));
```

Fig. 9: Extract of ABS method fetch. When this code is reached, the requested cache line n has status invalid or it is not in the cache. The function select chooses a cache line to be swapped with n. If there is still free space in the cache, select returns Nothing. If n has either shared or modified status in the next level cache, the method swap removes the cache line with address n, inserts the selected cache line and returns the current status of n; otherwise, swap simply returns Nothing.

Local synchronization in the SOS model between two structural entities (e.g., two caches in rule LC-HIT$_1$ of Fig. 3), is implemented by the following synchronization pattern in ABS (see Fig. 8). Given two objects o_1 and o_2, let o_1 execute method m_1, which checks the local conditions of o_1 (highlighted as region **A** in Fig. 8). If these local conditions hold, method m_2 on o_2 is called asynchronously. Method m_2 completes when the local conditions of o_2 hold (highlighted as region **B** in Fig. 8). However, when m_2 has returned and object o_1 again schedules method m_1, the conditions on object o_2 need no longer hold. Therefore, o_1 next calls the method m_3 *synchronously* to check these conditions again. If these condition still hold, method m_3 returns successfully (in general, having updated o_2), and we can proceed to do the local changes in o_1 (highlighted as region **C** in Fig. 8). Otherwise, the process needs to be repeated until we succeed. Note that method m_3 should not contain release points; because this method is called synchronously from a different object, a release point will in general have the potential of introducing deadlocks in the caller object.

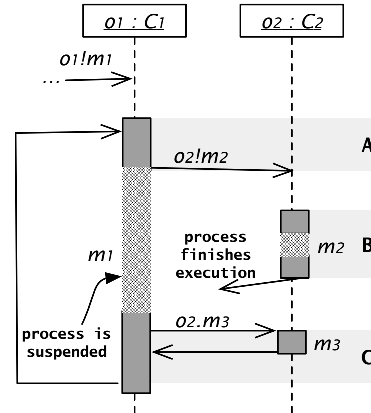
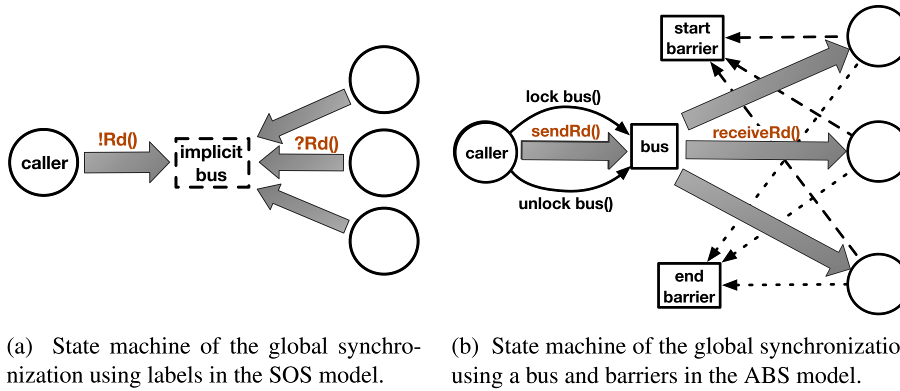


Fig. 8: Local synchronization between two ABS objects.

To illustrate the above protocol, consider the code snippet in Fig. 9, which corresponds to part of several rules in the SOS (in particular, rule LC-HIT$_1$). Here, the current object **this** corresponds to $caid_i$ in the SOS, running method fetch, and the referenced object in nextCache corresponds to $caid_j$. When fetch from nextCache returns, all the



(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. 10: Synchronization in SOS vs ABS. In the SOS model (Fig. a), circles represent nodes in the memory system and shaded arrows labelled transitions. Note that the bus is *implicit* in the SOS model, as synchronization is captured by label matching. In the ABS model (Fig. b), circles represent the same nodes as in the SOS model, shaded arrows method invocations, solid arrows mutual access to the bus object and dotted arrows barrier synchronizations.

required conditions in nextCache are *True*. However, since the call is asynchronous, (some of) the conditions may no longer hold when execution continues in **this**. This is addressed by checking the return value of method swap: If swap returns an address, it means the conditions still hold and the necessary updates are performed both locally and in nextCache; otherwise (when swap returns Nothing) fetch will be called again.

Global synchronization in the SOS (see Fig. 10a) is modelled by matching labelled transitions. To simulate this instantaneous communication in ABS, we introduced the classes Bus and Barrier. The synchronization protocol is activated by asynchronous calls to the respective methods sendRd and sendRdX of the bus. The bus subsequently asynchronously calls the corresponding methods receiveRd and receiveRdX of the caches. Two barriers start and end are used by the caches to synchronize the start, as well as the completion, of the local executions of methods receiveRd and receiveRdX.

However, observe that objects in ABS are input enabled: it is always possible to call a method on an object. In our model, this scheme may give rise to inconsistent states: the local status of a memory location which triggers an asynchronous call of one of the methods sendRd and sendRdX of the bus may be invalidated by other bus synchronizations. Therefore, we add a lock to the bus (see Figs. 5 and 6), which is used to ensure exclusive access to the *message pool* of the bus when one of the methods read, write, and fetch are executed. The lock is released in case bus synchronization is not needed. The overall scheme is depicted in Fig. 10b. The exclusive access to the message pool of the bus guarantees that the message pool of the bus contains at most one call to one of the methods sendRd and sendRdX. Consequently, the triggering condition of the call cannot be invalidated before the call has been executed. This *strict* locking strategy, however, decreases concurrency in the distributed system, but reduces the complexity of the proof of equivalence between the SOS and the distributed implementation. We discuss how to further enhance the parallelization in Sec. 5.

4 Correctness

In this section we discuss the correctness of the ABS model by means of a simulation relation between the transition system describing the semantics of the ABS model ot the multicore memory system and the transition system described by the SOS model.

The semantics of an ABS model can be described by a transition relation between global configurations. A global configuration is a (finite) set of object configurations. An object configuration is a tuple of the form $\langle oid, \sigma, p, Q \rangle$, where *oid* denotes the unique identity of the object, σ assigns values to the instance variables (fields) of the object, p denotes the currently executing process, and Q denotes a set of (suspended) processes. A process is a closure (τ, S) consisting of an assignment τ of values to the local variables of the statement S.

We refer to [4] for the details of the structural operational semantics for deriving transitions $G \to G'$ between global configurations in ABS. Since in ABS concurrent objects only interact via asynchronous method calls and processes are scheduled non-deterministically (which provides an abstraction from the order in which the processes are generated by method calls), the ABS semantics satisfies the following global confluence property that allows to commute consecutive computations steps of *independent* processes which belong to *different* objects. Two processes are independent if neither one is generated by the other by an asynchronous call.

Lemma 1 (Global confluence). *For any two transitions $G \to G_1$ and $G \to G_2$ that describe execution steps of independent processes of different objects, there exists a global configuration G' such that $G_1 \to G'$ and $G_2 \to G'$*

An object configuration is *stable* if the statement S to be executed has terminated or starts either with a **get** operation on a future or with an **await** statement on a Boolean condition or a future. A global ABS configuration is *stable* if all its object configurations are stable. Observe that our ABS model does not give rise to local divergent computations without passing through stable configurations; i.e., every local computation eventually enters a stable configuration. Together with the global confluence property in Lemma 1, this allows to restrict the semantics of the ABS model in the simulation relation to stable global configurations; i.e., transitions $G \Rightarrow G'$ between stable global configurations G and G' which result from a (non-empty) sequence of local execution steps of a *single* process from one stable configuration to a next one.

Because of the global synchronization with the bus in ABS described above, we may also represent without loss of generality the synchronization on the bus by a *single* global transition $G \Rightarrow G'$ which involves a completed execution of the method sendRd(...) (or sendRdX(...)) by the bus. This is justified because the global confluence allows for a scheduling policy such that the execution of the processes that are generated by these methods, i.e., the calls of the methods receiveRd(...) (or receiveRd(...)) are not interleaved with any other processes.

The simulation relation. The structural correspondence between a global configuration of the ABS model and a configuration of the SOS model is described in Sec. 3.2. For each method we have constructed a table which, among others, associates with some,

so-called *observable*, occurrences of **await** statements (appearing in the method body) a corresponding **dst** instruction. In general, the execution of the remaining (occurrences of) **await** statements, for which there does not exist a corresponding **dst** instruction, involves some asynchronous messaging *preparing* for the corresponding synchronous exchange of information in the SOS model. In some cases, the execution of these unobservable statements (e.g., the read and write methods) also does not correspond to a change of the SOS configuration. Let α map every stable global configuration G of the ABS model to a structurally equivalent configuration $\alpha(G)$ of the SOS model, which additionally maps every observable process (either queued or active) to the associated **dst** instruction (a process is observable if its corresponding statement is observable).

We arrive at the following theorem which expresses that the ABS model is a correct implementation of the abstract model.

Theorem 1. *Let G be a stable global configuration of the ABS model. If $G \Rightarrow G'$ then $\alpha(G) \rightarrow^* \alpha(G')$, where $\rightarrow^*$ denotes the reflexive, transitive closure of $\rightarrow$.*

Proof. The proof proceeds by a case analysis of the given transition $G \Rightarrow G'$, which, as discussed above, involves the local execution of some basic sequential code by a single object. For example, for the case of a completed execution of a method sendRd(...) (or sendRdX(...)) by the bus, a simple inspection of the sequential code of the methods that have been executed, e.g., sendRd(...) and receiveRd(...), suffices to establish the existence of a corresponding transition $\alpha(G) \rightarrow \alpha(G')$.

The remaining cases are captured by tables (as mentioned above) which provide for each method the following information. The statements in the **Location** column of each table represent for the respective method all possible processes generated by a call, i.e., a call to the method itself, and the processes which correspond to the **await** statements appearing in its body. In each row the **Next release point** statement indicates the next **await** statement or **return** statement that can be reached (statically). The **dst** instruction in each row specifies the instruction which corresponds to the **Location** statement in the simulation. Finally, **Enable condition** in each row specifies the enabling conditions (expressed in the abstract model) of the rule applications (of the abstract model) specified in **Rules**. In general these rule applications involve the sequential application of one or more rules. For unobservable statements, for which there is no corresponding **dst** instruction, the latter two columns are left unspecified.

The case analysis then consists of checking statically for each row the *local* structural correspondence between the resulting ABS process (the **Next release point**) and the resulting SOS configuration described by the specified rule applications.

5 Parallelism and Fairness of the ABS Model

This section discusses how to relax the eager locking policy of the bus implementation, without generating inconsistent states. Instead of locking the bus unconditionally when executing the read, write, and fetch methods in the ABS model, and releasing the lock when no bus synchronization is required, we only lock the bus when the triggering conditions of the bus synchronization may be invalidated. For example, an *optimistic* write implementation (see Fig. 11) tries to acquire the lock of the bus, and only after the

```
Int write(Ref r) {
 case lookup(cacheMemory, addr(r)) {
   Just(Sh) => {
     await bus!lock_bus();
     // after waking up do RACE DETECTION
     if (lookup(cacheMemory,addr(r)) == Just(Sh)) { // NO RACE
       await bus!sendRdX(addr(r),this);
       await bus!release_bus();
       cacheMemory = put(cacheMemory,addr(r),Mo);
     }
     else { // RACE CONDITION
       await bus!release_bus();
       await this!write(r); // RETRY
     } } ... } }
```

Fig. 11: Alternative, optimistic implementation of the write method to detect a bus race-condition and, in that case, retry the operation.

acquisition checks if a race-condition has happened and invalidated the shared status of the address n; in this case, the write method will *backtrack* and retry (by calling itself); otherwise the write operation can safely be performed.

The strict and relaxed variations of the global synchronization bear strong resemblance respectively to conservative [11, 12] and optimistic [13] algorithms in parallel and distributed discrete-event simulation (PDES) [14]. As with PDES, there is no clear winner between the strict (conservative) and relaxed (optimistic) versions of our cache simulator; certain computer programs (input-models) will be simulated faster using one version or the other, depending on the inter-dependency of the parallel components (for us, the caches). For the contrived experiment, we implemented a penalty system in the ABS model. A cache penalty is the cost (delay) incurred by failing to read or write to a particular level of cache — set here to $(L_1, L_2, L_3) =_{cost} (1, 10, 100)$ [15]. We compared the two versions for a scenario with full inter-dependency (simultaneous write instructions on the same memory block) and a scenario with minimal inter-dependency (write instructions on separate memory blocks) between 16 simulated cores. In these experiments the strict version was slightly faster up to 2% for the first case and losing out by up to 12% in the second case. The experiments were executed using the ABS-Erlang backend [16] and Erlang version 21, running on quad-socket 8-cores 16-hyperthreads Xeon®L7555, which yielded in total 64 hardware threads.

Fairness. A concern that often arises in parallel execution is fairness: the degree of variability when distributing the computing resources among different parallel components — here, the simulated cores. Fairness of parallel execution can affect the simulation's accuracy in approximating the intended (or idealized) manycore hardware. To ensure fairness of the simulation, we make use of *deployment components* [17] in ABS.

A *Deployment Component* (DC) is an ABS execution location that is created with a number of virtual resources (e.g., execution speed, memory use, network bandwidth), which are shared among its deployed objects. Any annotated statement [Cost: x] S decrements by x the resources of its DC and then completes, or it will stall its computation if there are currently not enough resources remaining; the statement S may

	Strict with DC	Relaxed with DC	Strict	Relaxed
$\sum_{penalty}$	43068	43290	39183	24956

Table 1: Total cache penalties between strict/relaxed, with/without DC configurations.

continue on the next passage of the global symbolic time where all the resources of the DCs have been renewed, and will eventually complete when its Cost has reached zero.

We make use of this resource modeling of ABS to assign equal (fair) resources of virtual execution speed to the simulated cores of the system. Each Core object is deployed onto a separate DC with fixed Speed(1) resources. The processing of each instruction has the same cost [Cost: 1] — a generalization, since common processor architectures execute different instructions in different speeds (cycles per instruction); e.g., `JUMP` is faster than `LOAD`. The result is that all Cores can execute maximum one instruction in every time interval of the global symbolic clock, and thus no Core can get too far ahead with processing its own instructions — a problem that manifests upon the parallel simulation of N number of cores using a physical machine of M cores, where N is vastly greater than M. To test this, we performed a write-congested experiment with a configuration of 20 simulated cores and 3 cache levels, comparing the strict and relaxed variations, with and without the use of deployment components. The results (shown in Table 1) were measured on a quad-core system running ABS-Erlang, counting the total cache penalties of all the cores. With respect to the strict variation, the results with and without DC have similar penalties; this can be attributed to the lock-step nature of strict bus synchronization, where no cache (and thus core) can unfairly stride forward. In the relaxed variation, however, where synchronization is less strict, we see that without the fairness imposed by DC, the penalties are almost halved, which means some cores are allowed to do multiple (successful) write operations while other cores are still waiting on the "backlog" to be simulated. This gives rise to less penalties, because of less runtime interleavings of the simulated cores and thus less competition between them.

6 Related Work

There is in general a significant gap between a formal model and its implementation [18]. 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 [19], rewriting logic [20, 21], and $\mathbb{K}$ [22] reduce this gap, and have been used to develop executable formal models of complex languages like C [23] and Java [24]. The relationship between SOS and rewriting logic semantics has been studied [25] without proposing a general solution for label matching. Bijo et al. implemented their SOS multicore memory model [26] in the rewriting logic system Maude [3] using an orchestrator for label matching, but do not provide a correctness proof wrt. the SOS. Different semantic styles can be modeled 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 [27]. Compared to these works on semantics, we implemented an SOS model in a distributed active object setting, and proved the correctness of this implementation.

Correctness-preserving compilation is related to correctness proofs for implementations, and ensures that the low-level representation of a program preserves the properties of the high-level model. Examples of this line of work include type-preserving translations into typed assembly languages [28] and formally verified compilers [29, 30], which proves the semantic preservation of a compiler from C to assembler code, but leaves shared-variable concurrency for future work. In contrast to this work which studies compilation from one language to another, our work focuses on a specific model and its implementation and specifically targets parallel systems.

Simulation tools for cache coherence protocols can evaluate performance and efficiency on different architectures (e.g., gems [31] and gem5 [32]). These tools perform evaluations of, e.g., the cache hit/miss ratio and response time, by running benchmark programs written as low-level read and write instructions to memory. Advanced simulators such as Graphite [33] and Sniper [34] run programs on distributed clusters to simulate executions on multicore architectures with thousands of cores. Unlike our work, these simulators are not based on a formal semantics and correctness proofs. Our work complements these simulators by supporting the executable exploration of design choices from a programmer perspective rather from hardware design. Compared to worst-case response time analysis for concurrent programs on multicore architectures [35], our focus is on the underlying data movement rather than the response time.

7 Conclusion

We have introduced in this paper a methodology for implementing SOS models in the active object language ABS, and applied this methodology to the implementation of a SOS model of an abstraction of multicore memory systems, resulting in a parallel simulator for these systems. A challenge for this implementation is to correctly implement the synchronization patterns of the SOS rules, which may cross encapsulation barriers in the active objects, and in particular label synchronization on parallel transitions steps. We prove the correctness of this particular implementation, exploiting that the ABS model allows for a high-level coarse-grained semantics. We investigated the further parallelization and fairness of the ABS model.

The results obtained in this paper provide a promising basis for further development of the ABS model for simulating the execution of (object-oriented) programs on multicore architectures. A first such development concerns an extension of the abstract memory model with data. In particular, having the addresses of the memory locations themselves as data allows to model and simulate different data layouts of the dynamically generated object structures.

References

1. Plotkin, G.D.: A structural approach to operational semantics. Journal of Logic and Algebraic Programming **60-61** (2004) 17–139

2. Bijo, S., Johnsen, E.B., Pun, K.I, Tapia Tarifa, S.L.: A formal model of parallel execution on multicore architectures with multilevel caches. In: Proc. 14th Intl. Conf. on Formal Aspects of Component Software (FACS 2017). Volume 10487 of Lecture Notes in Computer Science., Springer (2017) 58–77
3. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.L., eds.: All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic. Volume 4350 of Lecture Notes in Computer Science. Springer (2007)
4. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: A core language for abstract behavioral specification. In Aichernig, B., de Boer, F.S., Bonsangue, M.M., eds.: Proc. 9th International Symposium on Formal Methods for Components and Objects (FMCO 2010). Volume 6957 of Lecture Notes in Computer Science., Springer (2011) 142–164
5. Boer, F.D., Serbanescu, V., Hähnle, R., Henrio, L., Rochas, J., Din, C.C., Johnsen, E.B., Sirjani, M., Khamespanah, E., Fernandez-Reyes, K., Yang, A.M.: A survey of active object languages. ACM Comput. Surv. **50**(5) (October 2017) 76:1–76:39
6. Bijo, S., Johnsen, E.B., Pun, K.I, Tapia Tarifa, S.L.: A formal model of parallel execution in multicore architectures with multilevel caches (long version). Research report, Department of Informatics, University of Oslo (2018) Under revision for journal publication. URL `http://violet.at.ifi.uio.no/papers/mc-rr.pdf`.
7. Solihin, Y.: Fundamentals of Parallel Multicore Architecture. 1st edn. Chapman & Hall/CRC (2015)
8. Culler, D.E., Gupta, A., Singh, J.P.: Parallel Computer Architecture: A Hardware/Software Approach. 1st edn. Morgan Kaufmann Publishers Inc. (1997)
9. Sorin, D.J., Hill, M.D., Wood, D.A.: A Primer on Memory Consistency and Cache Coherence. 1st edn. Morgan & Claypool Publishers (2011)
10. 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'73, San Francisco, CA, USA, Morgan Kaufmann Publishers Inc. (1973) 235–245
11. Bryant, R.E.: Simulation of Packet Communication Architecture Computer Systems. Technical Report MIT/LCS/TR-188, MIT, Lab for Computer Science (November 1977)
12. Chandy, K.M., Misra, J.: Distributed Simulation: A Case Study in Design and Verification of Distributed Programs. IEEE Transactions on Software Engineering **SE-5**(5) (September 1979) 440–452
13. Jefferson, D.R.: Virtual Time. ACM Trans. Program. Lang. Syst. **7**(3) (July 1985) 404–425
14. Fujimoto, R.M.: Parallel and distributed simulation systems. Wiley (2000)
15. Schmidl, D., Vesterkjær, A., Müller, M.S.: Evaluating OpenMP performance on thousands of cores on the numascale architecture. In: Parallel Computing: On the Road to Exascale, Proceedings of the International Conference on Parallel Computing (ParCo 2015). Volume 27 of Advances in Parallel Computing., IOS Press (2016) 83–92
16. Wong, P.Y.H., Albert, E., Muschevici, R., Proença, J., Schäfer, J., Schlatte, R.: The ABS tool suite: modelling, executing and analysing distributed adaptable object-oriented systems. STTT **14**(5) (2012) 567–588
17. Johnsen, E.B., Schlatte, R., Tapia Tarifa, S.L.: Integrating deployment architectures and resource consumption in timed object-oriented models. Journal of Logical and Algebraic Methods in Programming **84**(1) (2015) 67–91
18. 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: It's All About Coordination - Essays to Celebrate the Lifelong Scientific Achievements of Farhad Arbab. Volume 10865 of Lecture Notes in Computer Science., Springer (2018) 107–121

19. Felleisen, M., Findler, R.B., Flatt, M.: Semantics Engineering with PLT Redex. The MIT Press (2009)
20. Meseguer, J., Rosu, G.: The rewriting logic semantics project: A progress report. Inf. Comput. **231** (2013) 38–69
21. Meseguer, J., Rosu, G.: The rewriting logic semantics project. Theor. Comput. Sci. **373**(3) (2007) 213–237
22. Rosu, G.: $\mathbb{K}$: A semantic framework for programming languages and formal analysis tools. In: Dependable Software Systems Engineering. IOS Press (2017) 186–206
23. Ellison, C., Rosu, G.: An executable formal semantics of C with applications. In Field, J., Hicks, M., eds.: Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2012), ACM (2012) 533–544
24. Bogdanas, D., Rosu, G.: K-java: A complete semantics of java. In Rajamani, S.K., Walker, D., eds.: Proc. 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2015), ACM (2015) 445–456
25. Serbanuta, T., Rosu, G., Meseguer, J.: A rewriting logic approach to operational semantics. Inf. Comput. **207**(2) (2009) 305–340
26. Bijo, S., Johnsen, E.B., Pun, K.I, Tapia Tarifa, S.L.: A Maude framework for cache coherent multicore architectures. In: Proceedings of the 11th International Workshop on Rewriting Logic and Its Applications (WRLA). Volume 9942 of Lecture Notes in Computer Science., Springer (2016) 47–63
27. Eckhardt, J., Mühlbauer, T., Meseguer, J., Wirsing, M.: Semantics, distributed implementation, and formal analysis of KLAIM models in Maude. Sci. Comput. Program. **99** (2015) 24–74
28. Morrisett, J.G., Walker, D., Crary, K., Glew, N.: From system F to typed assembly language. ACM Trans. Program. Lang. Syst. **21**(3) (1999) 527–568
29. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM **52**(7) (2009) 107–115
30. Leroy, X.: A formally verified compiler back-end. J. Autom. Reasoning **43**(4) (2009) 363–446
31. Martin, M.M.K., Sorin, D.J., Beckmann, B.M., Marty, M.R., Xu, M., Alameldeen, A.R., Moore, K.E., Hill, M.D., Wood, D.A.: Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. SIGARCH Computer Architecture News **33**(4) (2005) 92–99
32. Binkert, N., Beckmann, B., Black, G., Reinhardt, S.K., Saidi, A., Basu, A., Hestness, J., Hower, D.R., Krishna, T., Sardashti, S., Sen, R., Sewell, K., Shoaib, M., Vaish, N., Hill, M.D., Wood, D.A.: The gem5 simulator. SIGARCH Computer Architecture News **39**(2) (2011) 1–7
33. Miller, J.E., Kasture, H., Kurian, G., Gruenwald, C., Beckmann, N., Celio, C., Eastep, J., Agarwal, A.: Graphite: A distributed parallel simulator for multicores. In: Proceedings of the 16th International Symposium on High-Performance Computer Architecture (HPCA), IEEE Computer Society (2010) 1–12
34. Carlson, T.E., Heirman, W., Eeckhout, L.: Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In: Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis (SC), ACM (2011) 52:1–52:12
35. Li, Y., Suhendra, V., Liang, Y., Mitra, T., Roychoudhury, A.: Timing analysis of concurrent programs running on shared cache multi-cores. In: Proceedings of the 30th IEEE Real-Time Systems Symposium (RTSS), IEEE Computer Society (2009) 57–67