Proving Correctness of Parallel Implementations of Transition System Models

FRANK S. DE BOER, CWI, the Netherlands
EINAR BROCH JOHNSEN, University of Oslo, Norway
VIOLET KA I PUN, Western Norway University of Applied Sciences, Norway
SILVIA LIZETH TAPIA TARIFA, University of Oslo, Norway

This paper addresses the long-standing problem of program correctness for programs that describe systems of parallel executing processes. We propose a new method for proving correctness of parallel implementations of high-level models expressed as transition systems. The implementation language underlying the method is based on the concurrency model of actors and active objects. The method defines program correctness in terms of a simulation relation between the transition system which specifies the program semantics of the parallel program and the transition system that is described by the correctness specification. The simulation relation itself abstracts from the fine-grained interleaving of parallel processes by exploiting a global confluence property of the concurrency model of the implementation language considered in this paper. As a proof of concept, we apply our method to the correctness of a parallel simulator of multicore memory systems.

CCS Concepts: • **Theory of computation → Semantics and reasoning**; **Program specifications**; **Program verification**.

Additional Key Words and Phrases: correctness, transition system models, active objects, simulation, multicore memory systems

ACM Reference Format:
Frank S. de Boer, Einar Broch Johnsen, Violet Ka I Pun, and Silvia Lizeth Tapia Tarifa. 2024. Proving Correctness of Parallel Implementations of Transition System Models. *ACM Trans. Program. Lang. Syst.* XX, Y, Article 1 (April 2024), 51 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

A long-standing challenge in Computer Science is the formal specification and verification of programs, notably that of parallel programs supporting complex communication and synchronization mechanisms. We can distinguish between logic- and semantics-based methods for establishing program correctness. Methods that are based on logic use assertions to express behavioral properties and generate proof conditions for their validation, which are usually discharged by interactive theorem proving. These methods are applicable to infinite-state systems and to actual programs used in practice (see for example [24] for the verification of a corrected version of the TimSort sorting program of the Java Collections Framework). One of the main challenges for the use of logic-based approaches stems from the complexity of the specification of invariant properties and the interactive use of a theorem prover. On the other hand, semantics-based methods use transition

Authors' addresses: Frank S. de Boer, CWI, Amsterdam, the Netherlands, F.S.de.Boer@cwi.nl; Einar Broch Johnsen, University of Oslo, Oslo, Norway, einarj@ifi.uio.no; Violet Ka I Pun, Western Norway University of Applied Sciences, Bergen, Norway, Violet.Ka.I.Pun@hvl.no; Silvia Lizeth Tapia Tarifa, University of Oslo, Oslo, Norway, sltarifa@ifi.uio.no.

ACM 0164-0925/2024/4-ART1
https://doi.org/10.1145/nnnnnnn.nnnnnnn

systems to model system behavior. Transitions may be small-step or big-step, finite or infinite state, and support a range of verification methods including inductive reasoning and automated model checking (in the case of finite state models).

The main contribution of this paper is a new semantics-based verification method for parallel programs, exploiting the concurrency model of Actors [2] and active objects [23]. The method is based on specifying an abstraction of the overall behavior of a parallel program in terms of a *transition system model* (TS model, for short). The TS model describes behavior by means of the local transformation of states by application of symbolic transformation rules. Examples of formalisms that can be used to specify such TS models include the *Chemical Abstract Machine* [7], the rewriting system *Maude* [19], $\mathbb{K}$ [51] and *Structural Operational Semantics* [50]. Verifying that a parallel program satisfies such a correctness specification then involves establishing a *simulation relation* between the transition system describing the semantics of the parallel program and the system described by the TS model (which may be infinite state). Although these systems are at different levels of abstraction and use different mechanisms for communication and synchronization, our proof method allows the simulation relation to be established by means of *syntax-directed local reasoning*. In this paper, the proposed proof method is justified in terms of the semantic properties of our targeted implementation language, which is an actor language with cooperative scheduling.

Our method supports a general two-step approach to proving the correctness of parallel programs:

(1) Verify global behavioral properties using a high-level formal model which abstracts from the complexity of the concurrency model of the target language to support inductive proofs of global properties.
(2) Justify the correctness of the parallel implementation in the target language with respect to the high-level model in terms of a simulation relation.

For the first step, a TS model allows for the formal description of overall system behavior in a syntax-oriented, compositional way, using inference rules for local transitions and their composition. Process synchronization can be expressed abstractly using, e.g., conditions on 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. Whereas TS models are well-suited for formalizing language semantics and for reasoning about language meta-theory, they are also well-suited to describe specific systems in order to reason about, e.g., reachability or state invariance.

For the second step, we need an implementation in an Actor language with a formal semantics (e.g., formalized by a TS model), such that a simulation relation can be formally established. The concurrency model of Actors then enables proving correctness in terms of syntax-directed sequential reasoning. In this paper, we have opted for the *active object* language ABS [30, 52] (ABS stands for *Abstract Behavioral Specification*). The semantics of ABS is formally defined by a TS model [32] and implemented by backends[1] in Erlang, Haskell, and Java, all of which support parallel execution. It has been developed and applied in the context of various EU projects, e.g., in the EU FP7 projects HATS[2] (*Highly Adaptable and Trustworthy Software using Formal Models*) and ENVISAGE[3] (*Engineering Virtualized Services*). In these projects, ABS has been extended and successfully applied to the formal modeling and analysis of software product families [21] and software services deployed on the Cloud [34]. The ABS tool suite [3, 5, 8, 25, 26, 35, 37, 38, 53]

[1]https://abs-models.org/
[2]https://cordis.europa.eu/project/id/231620
[3]https://cordis.europa.eu/project/id/610582

has been further applied to case studies, targeting, e.g., cloud-based frameworks [4, 33, 42, 43, 59], railway operations [36] and computational biology.[4]

The parallel execution of active objects (for a survey of active object languages, see [23]) is a direct consequence of decoupling method execution from method invocation by means of *asynchronous* method invocations. ABS further integrates a strict *encapsulation* of the local state of an active object with explicit language constructs for the *cooperative scheduling* of its method executions. Since ABS is tailored to the description of distributed systems, it abstracts from the order in which method invocations are generated.

In the definition of the simulation relation, cooperative scheduling allows the interleaving of methods in an active object to match the granularity of the transition rules of the corresponding TS model. Moreover, the parallel execution of active objects in ABS satisfies a *global confluence* property which allows to express *locally* the proof conditions of the simulation relation in a syntax-directed manner, abstracting from the fine-grained interleaving of the method executions.

As a proof of concept we introduce our method by application to a parallel simulator of multicore memory systems. 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 is usually not visible to the programmers, 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 TS model of multicore memory systems with correctness proofs for cache coherency was described in [13, 14], together with a prototype implementation in the rewriting logic system Maude [19]. However, this fairly direct implementation of the TS model is not well suited to simulate large systems. Therefore this paper introduces a parallel implementation, based on the active object model of ABS; we apply our method to prove its correctness.

This paper extends [9] which describes a first version of the use case. The extension consists of formalizing the novel idea of annotating ABS programs with the rule names of the TS model and the use of a global confluence property of the ABS semantics in the formal semantics (and verification) of these annotations. Because of the absence of this high-level specification of the simulation relation between the ABS program and the TS model, the ABS implementation in [9] has been developed largely independent of the TS model, which considerably complicated the correctness proof. In contrast, the application of our new methodology led to a major refactoring of the ABS implementation described in [9], reflecting a correctness-by-design development methodology.

In summary, the main contributions of this paper are

- a novel semantics-based method for proving the correctness of parallel systems, based on the Actor model of concurrency,
- a justification of the method in terms of the semantic properties of the Actor concurrency model, and
- a proof-of-concept case study illustrating the application of the method to a parallel simulator of a multicore memory system.

Plan for the paper. The next section introduces the main concepts of the ABS language and Section 3 the use of transition rules as annotations of ABS programs. Section 4 explains the TS model of multicore memory systems and Section 5 considers its ABS implementation and the associated correctness proof. We discuss related work in Section 6, and draw some general conclusions and discuss future work in Section 7. Appendices A, B and C further detail the global confluence property, the correctness proof and the multicore TS model of the case study, respectively.

[4]https://www.compugene.tu-darmstadt.de

Statement	**Meaning**
new C	Creation of an instance of class C
switch (e){e$_1$=> s$_1 \cdots$ e$_n$=> s$_n$}	Pattern matching
await b	Suspension on a Boolean condition
await e!m($e_1, \ldots, e_n$)	Suspension on termination of a asynchronous call
e!m($e_1, \ldots, e_n$)	Non-blocking asynchronous call
e.m($e_1, \ldots, e_n$)	Blocking synchronous call
this.m($e_1, \ldots, e_n$)	Inlined (recursive) self-call

Table 1. Basic ABS statements used in this paper. Here, b is a Boolean expression, e and e$_i$ denote expressions.

2 ABS: ACTORS WITH COOPERATIVE CONCURRENCY

ABS is a modeling language for designing, verifying, and executing concurrent software [30, 52]. The core language combines the syntax and object-oriented style of Java with the Actor model of concurrency [31], resulting in active objects which decouple communication and synchronization using asynchronous method calls and cooperative scheduling [23]. In ABS communication (sending a method call) and execution (scheduling an incoming method call) are decoupled via asynchronous method calls that generate processes (which execute the called methods) within the called (active) object and do not impose any synchronization between caller and callee. Thus, the caller can continue execution until the result of a method call is needed, and the callee can schedule method calls from multiple callers as needed. When synchronization between two objects is needed (e.g., a process needs the result of a method call), it is realised by means of (implicit) futures and cooperative scheduling. In ABS, objects have state (fields and object parameters) which is shared between all its processes. However, only one process may execute at a time in any object (i.e., the objects have a built-in mutex). Similarly, processes also have their own state (local variable and parameters to the method calls). A process executing in one object can allow another process to be scheduled in the same object by means of explicit suspension points. The active process suspends itself when waiting for the result of another method call or waiting for a Boolean condition over the actor state, using an **await**-statement. Rescheduling the process at the suspension point may then depend on the resolution of a future or on a Boolean condition becoming true. This mechanism of cooperative scheduling allows the interleaving of different processes to be captured very precisely in ABS.

The imperative layer of synchronization and communication is complemented by a functional layer, used for computations over the internal data of objects. The functional layer combines parametric algebraic datatypes (ADTs) and a simple functional language with case distinction and pattern matching. ABS includes a library with predefined datatypes and operations (e.g., Int, Bool), including a datatype Maybe that is used to store optional values,[5] and parametric datatypes and associated functions (e.g., lists,[6] sets and maps). All other types and functions are user-defined.

In the following, the basic ABS-related statements used in this paper (and shown in Table 1) are explained in terms of some general synchronization patterns (we omit well-known statements such assignment, **skip**, **if**, **foreach**, **return**, etc.).[7]

[5]The Maybe datatype is used when the functional code that is assigned to a variable may not return a value in all cases. If no value is returned, the default value is Nothing, otherwise the value is wrapped inside the Just construct, e.g., Just(True). Further details can be found in the ABS documentation https://abs-models.org/manual/#sec:builtin-types.

[6]The parametric data type List<A> is defined as Nil | Cons(A head, List<A> tail), where A is a datatype.

[7]For the full set of statements of ABS, we refer to the ABS documentation at https://abs-models.org/manual/#-statements. In ABS, method calls are not guaranteed to terminate since statements in the methods might include **while**-loops and recursive calls.

2.1 Synchronization Patterns

In this section, we discuss encodings in ABS of a basic locking mechanism, atomic operations, and a broadcast mechanism for global synchronization (using barriers).

Locks. The basic mechanism of asynchronous method calls and cooperative scheduling in ABS can be explained by the simple code example of a class Lock (Figure 1). It uses an **await**-statement on a Boolean condition to model a binary semaphore, which enforces exclusive access to a common resource "lock", modeled as an instance of the class Lock (instances are dynamically created by executing the expression **new** Lock).

The execution of the take_lock method will be suspended by the **await** unlocked statement. This statement *releases the control*, allowing the scheduling of other (enabled) processes within the Lock object. When the local condition unlocked inside the Lock object has become true, the generated take_lock processes within the Lock object will compete for execution. The scheduled process will then terminate and return by setting unlocked to False.

In general, the *suspension points* defined by **await**-statements define the granularity of interleaving of the processes of an object. The ABS statement **await** lock!take_lock() will only suspend the process that issued the call (and release control in the caller object) until take_lock has returned. In contrast, a *synchronous* call lock.take_lock() will generate a process for the execution of the take_lock() method by the lock object and block (all the processes of) the caller object until the method returns.

```
class Lock {
  Bool unlocked = True;

  Unit take_lock {
    await unlocked;
    unlocked = False; }

  Unit release_lock {
    unlocked = True; }
}
{// using the lock
  ... lock = new Lock();
  await lock!take_lock;
  ... // critical resource
  lock.release_lock();
}
```

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

Atomic operations. The interleaving model of concurrency of ABS allows for a simple and high-level implementation of atomic operations. For example, Figure 2 shows a general ABS implementation of test-and-set instructions [6], where the concurrency model guarantees that the local */*test(input)*/* and */*set*/* statements, assuming that they do not involve suspension points, are not interleaved and thus can be thought of as executed in a single atomic operation. An instance of this atomic operation can be observed in the method remove_inv in Figure 11.

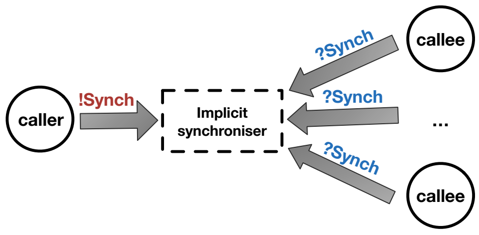
In ABS test instructions can be implemented using the **switch**-statement, which evaluates an expression that matches the resulting value against a pattern e_i[8] in the different branches. This statement has mainly been used to pattern match the ADTs used in the ABS program discussed in this paper. In the simplest case, this pattern can be replaced by an **if**-statement.

```
Bool TestandSet (/*input*/) {
  Bool fail = False;
  switch /*test(input)*/ {
    True => /*set*/;
    False => fail = True;
  }
  return fail;
}
```

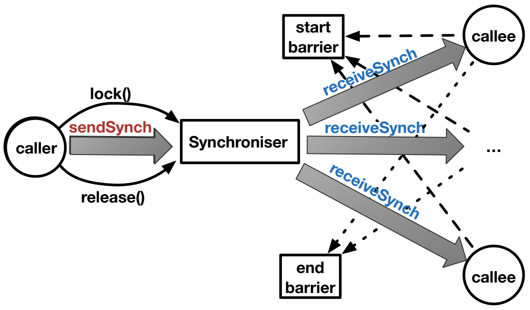
Fig. 2. Test-and-set pattern in ABS.

Broadcast synchronization. Figure 3a shows how broadcast synchronization in a labelled TS model can be enforced by simply matching labels (where exclamation marks and question mark denote sending and receiving a signal, respectively. An example is detailed in Section 4), thus abstracting from the implementation details of the implicit multi-party synchronizer. On the other hand, in programming languages for distributed systems like

[8]A pattern e_i in the **switch**-statement can be an expression that includes a datatype constructor e.g., the list constructor Cons(h,t), in which case the local variables h and t are bound by the pattern matching and available in the statement s_i associated to e_i. Underscore _ matches any pattern.



a) Broadcast synchronization in a labelled TS model.



b) Broadcast synchronisation using an explicit synchronizer and barriers in the ABS model.

Fig. 3. Broadcast synchronisation patterns in TS model and ABS.

ABS the multi-party synchronization needs to be programmed explicitly; Figure 3b illustrates the architecture of the ABS implementation shown in Figure 4.

The class Broadcast serves as a template (or design pattern) for the implementation of a broadcast mechanism between objects that are specified by the interface IBroadcast. The broadcastSync method encapsulates a synchronisation protocol between Broadcast instances which uses the additional classes Synchronizer and Barrier. This protocol consists of a synchronous call to the method sendSync of an instance of the class Synchronizer (denoted by sync) which in turn asynchronously calls the method receiveSync of the objects stored in the set network of Broadcast instances, excluding the caller object executing the broadcastSync method. We abstract from whether the sync object is passed as a parameter of the broadcastSync method or is part of the local state of any Broadcast instance. The local computation specified by the receiveSync method by the objects in receivers is synchronized by calls of the method synchronize of the new instances start and end of class Barrier. That is, the execution of this method by the start and end barriers synchronize the start and the termination of the execution of the method receiveSync by the objects in receivers and the termination of the sendSync method itself. This is achieved by a "countdown" of the number of objects in receivers that have called the synchronize method plus one, in case of the end barrier. The synchronize method of the start barrier is called asynchronously (Line 16) and introduces a release point in order to avoid a deadlock that may arise when an object that has not yet called the synchronize method of the start barrier is blocked on a synchronous method call to an object that has already invoked (synchronously) the synchronize method of the start barrier. On the other hand, the corresponding call to the end barrier is synchronous to ensure that all the objects in receivers have completed their local computations. The additional synchronisation of the synchronizer object on the end barrier ensures that also the caller of the sendSync method is blocked until all the local computations specified by the receiveSync method have been completed.

Objects in ABS are input enabled, so it is always possible to call a method on an object. In our implementation, this scheme could give rise to inconsistent states if several objects start the protocol in parallel. To ensure exclusive access to the synchronizer at the start of the protocol, we add a lock to the synchronizer protocol, such that the caller must take the lock before calling sendSync and release the lock upon completion of the call. The resulting exclusive access to the synchronizer guarantees that its message pool contains at most one call to the method sendSync.

```
interface IBroadcast {
   Bool broadcastSync(...);
   Unit receiveSync (IBarrier start, IBarrier end, ...)
}

class Broadcast implements IBroadcast, ... {
   Bool broadcastSync(...) {
      Bool signal=False;
      await sync!lock();
      if /*test*/ { sync.sendSync(this,...); /*set*/; signal=True; }
      sync.release();
      return signal
   }

   Unit receiveSync(IBarrier start, IBarrier end, ...) {
      await start!synchronize();
      /*some local computation*/;
      end.synchronize();
   }
   ...
}

class Synchronizer (Set<IBroadcast> network) implements ISynchronizer {
   Bool unlocked = True;
   Unit lock() { await unlocked; unlocked = False; }
   Unit release() { unlocked = True; }
   Unit sendSync(IBroadcast caller,...) {
      Set<IBroadcast> receivers = remove(network,caller);
      Int nrrecs= size(receivers);
      IBarrier start = new Barrier(nrrecs);
      IBarrier end = new Barrier(nrrecs+1);
      foreach (receiver in receivers) { receiver!receiveSync(start,end,...); }
      end.synchronize();
   }
}

class Barrier(Int participants) implements IBarrier {
   Unit synchronize() { participants = participants − 1; await (participants == 0); }
}
```

Fig. 4. Global synchronisation pattern in ABS.

2.2 Semantics

ABS is a formally defined language [32]; in fact, its (operational) semantics is defined by a TS model which allows us to reason formally about the execution of ABS programs. 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 (the object's "queue"). 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 [32] for the details of the TS model for deriving transitions $G \rightarrow G'$ between global configurations in ABS.

Although only one thread of control can execute in an active object at any time (taken by the active process), cooperative scheduling allows different processes to interleave by releasing the thread of control (allowing another suspended process to become active) at explicitly declared points in the code, i.e., the **await**-statements. When the currently executing process is suspended by an **await**-statement, another (enabled) process is scheduled. Access to an object's fields is protected: any non-local (outside of the object) read or write to fields happens via method calls, mitigating race-conditions or the need for extensive use of explicit mutual exclusion mechanisms (locks).

Since active objects only interact via 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 (see also [9, 60]) that allows commuting consecutive local computations steps of processes which belong to *different* objects.

Theorem 1 (Global confluence). *For any two transitions $G_1 \rightarrow G_2$ and $G_1 \rightarrow G_3$ that describe execution steps of processes of different objects, there exists a global configuration G_4 such that $G_2 \rightarrow G_4$ and $G_3 \rightarrow G_4$.*

It is worthwhile to observe that this global confluence property follows from the following basic principles underlying actor-based languages:

- encapsulation of the local state,
- monotonicity of the local transitions which are not affected by adding messages and
- the basic algebraic laws of adding and deleting elements from a multiset (of messages).

In fact, global confluence can be proven in an abstract setting which captures the general semantics of actor languages, see Appendix A. Theorem 1 then follows by embedding the semantics of ABS into this abstract setting. The details of this embedding for the communication and synchronization mechanisms of ABS (method calls, **await**-statements and futures) are discussed in the appendix.

An important consequence of Theorem 1, which underlies the main results of this paper, is that we can restrict the global interleaving between processes by reordering the execution steps in an ABS computation. In particular, we can restrict the interleaving semantics of the ABS model taking into account general semantic properties of synchronous communication, and the implementation of locks and broadcast synchronization in ABS, as explained next.

Since a synchronous call to a method of *another* object in ABS, blocks all processes of the caller (object), the global confluence property allows further restricting the interleaving of the ABS processes so that the caller process is resumed *immediately* after the synchronous method invocation has terminated.

It is worthwhile to note that in general we *cannot* assume that a method that is called synchronously in ABS is also scheduled *immediately for execution* because this would discard the possible execution of other processes by the callee.

The global confluence property also allows for abstracting from the internal computation steps of the above ABS implementation of the global (broadcast) synchronization pattern, because it allows scheduling the processes generated by the *broadcast* method such that the execution of this method is not interleaved with any other processes.

We can formalize the above in terms of the following notion of *stable* object configurations. An object configuration is stable if the statement to be executed denotes the termination of an *asynchronously* called method (we let *idle* denote the terminated process), or it starts with a (blocking) synchronous call or an **await**-statement.

Definition 1 (Stable configurations). *Let S and S' be statements and τ a local variable assignment. An object configuration $\langle oid, \sigma, p, Q\rangle$ is* stable *if p denotes the terminated process idle or p denotes a process (τ, S'), where S' denotes one of the statements*

- $e.m(\overline{e}); S$,
- **await** $b; S$ *or*
- **await** $e!m(\overline{e}); S$.

A global ABS configuration is stable *if all its object configurations are stable.*

Since synchronous self-calls are executed by inlining, they do not represent interleaving points. In the sequel $G \Rightarrow G'$ denotes the transition relation which describes execution starting from a global stable configuration G to a next global stable configuration G' (without passing intermediate global stable configurations). We distinguish the following three cases:

(1) The transition $G \Rightarrow G'$ describes the *local* execution of a method by a single object.
(2) The transition $G \Rightarrow G'$ describes the *rendez-vous* between the caller and callee of a synchronous method call in terms of the terminating execution of the called method, *followed* by the resumption of the suspended call.
(3) The transition $G \Rightarrow G'$ describes the effect of executing the broadcast method, which thus describes the *global* synchronization of different objects.

This coarse-grained interleaving semantics of ABS forms the basis for the general methodology to prove correctness of ABS implementations of TS models, described next.

3 THE GENERAL METHODOLOGY

3.1 Annotating ABS Methods with Rules from the TS Model

For an introduction to TS models, see, e.g., [49]. The general methodology for developing ABS implementations of abstract TS models exploits the coarse-grained interleaving described in Section 2 (denoted by the transition relation $\Rightarrow$). This course-grained interleaving allows us to focus on the design of *local, sequential* code that implements the individual transition rules. This is reflected by the use of transition rules as a *specification formalism* for ABS code. A *conditional transition rule $b : R$* consists of a local Boolean condition b in ABS and the name R of a transition rule. We use sequences $b_1 : R_1; \ldots; b_n : R_n$ of conditional transition rules to annotate *stable points* in ABS method definitions. A stable point in a method definition denotes either its body or a statement of its body that starts with an external synchronous call or an **await**-statement. The idea is that each b_i is evaluated as a condition which identifies a *path* leading from the annotated stable point to a next one or to termination. The execution of this path should correspond to the application of the associated transition rule R_i. This correspondence involves a simulation relation, described below.

A sequence $b_1 : R_1; \ldots; b_n : R_n$ of conditional transition rules is evaluated from left to right, that is, the first transition rule from the left, the Boolean condition of which evaluates to true, is applicable. The case that all Boolean conditions are false means that there does not exist a transition rule for *any* path from the annotated stable point to a next one or to termination (in the simulation relation, all these paths would correspond to a "silent" transition). As a special case, we stipulate that for *any* path leading from a stable point *which has no associated annotation* to a next stable point (or to termination), there does *not* exist a corresponding transition rule. The use of annotations in the ABS code of the multicore memory system is shown in Section 5.2.1.

3.2 Correctness of the Implementation

The correctness of the ABS implementation with respect to a given TS model can be established by means of a simulation relation between the transition system describing the semantics of the

ABS implementation and the transition system describing the TS model. The annotation of ABS code with (conditional) rules from the TS model provides a high-level description of the simulation relation, describing which rule(s) correspond to the execution of the ABS code from one stable point to a next one (or to termination). Underlying this high-level description, we define a simulation relation between ABS configurations and the runtime states of the TS model. This simulation relation is defined as an abstraction function α which maps every stable global ABS configuration G to a behaviorally equivalent configuration $\alpha(G)$ of the TS model. The abstraction function for the ABS code of the multicore memory system is shown in Section 5.2.2.

We restrict the simulation relation to *reachable* ABS configurations. A configuration G of an ABS program is reachable if $G_0 \Rightarrow^* G$, for some *initial* configuration G_0. In an initial configuration of the ABS multicore program, all process queues are empty and the only active processes are those about to execute the run methods of the cores. This restriction allows us to use some general properties of the ABS semantics; e.g., upon return of a synchronous call, the local state of the calling object has not changed.

We can now express that an ABS program is a correct implementation of a TS model by proving that the following theorem holds:

Definition 2 (Correctness). *Given an ABS program and a TS model, let α be an abstraction function from configurations of the ABS program to configurations of the TS model. The ABS program is a correct implementation of the TS model, if for any reachable configuration G and transition $G \Rightarrow G'$ of the ABS program we have that $\alpha(G) = \alpha(G')$ or $\alpha(G) \rightarrow \alpha(G')$.*

Because of the general confluence property of the ABS semantics, it suffices to verify the annotations of methods in terms of the abstraction function α to prove that α is a simulation relation. The general idea is that for each transition $G \Rightarrow G'$ that results from the execution from one stable point to a next one (or to termination), we have to show that $\alpha(G')$ results from $\alpha(G)$ by applying the enabled rule from the TS model associated with the initial stable point. In case no rule from the TS model is enabled, we have a "silent" step, that is, $\alpha(G) = \alpha(G')$.

4 A TS MODEL FOR MULTICORE MEMORY SYSTEMS

Design decisions for programs running on top of a multicore memory system can be explored using simulators (e.g., [15, 18, 44, 47]). Bijo et al. developed a TS model for multicore memory systems [13, 14]. Taking this TS model as a starting point, we will study how a parallel simulator can be developed in ABS which implements the TS model and use this development to discuss the details of our proof methodology for program correctness. We first introduce the main concepts of multicore memory systems and then look at their formalization in terms of a TS model.

4.1 A Short Overview of Multicore Memory Systems

A multicore memory system consists of cores that contain *tasks* to be executed, the *data layout* in main memory (indicating where data is allocated), and a system *architecture* consisting of cores with private multi-level caches and shared memory (see Figure 5). Such a system is parametric in the number of cores, the number and size of caches, and the associativity and replacement policy. Data is organized in blocks that move between the caches and the main memory. For simplicity, we abstract from the data content of the memory blocks, assume that the size of cache lines and memory blocks in main memory coincide and that a local reference to a memory block is represented directly by the corresponding memory address, and transfer memory blocks from the caches of one core to the caches of another core via the main memory. As a consequence, the tasks executed in the cores are represented as data access patterns, abstracting from their computational content.

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

Fig. 6. Syntax of runtime configurations, where over-bar denotes sets (e.g., $\overline{CR}$).

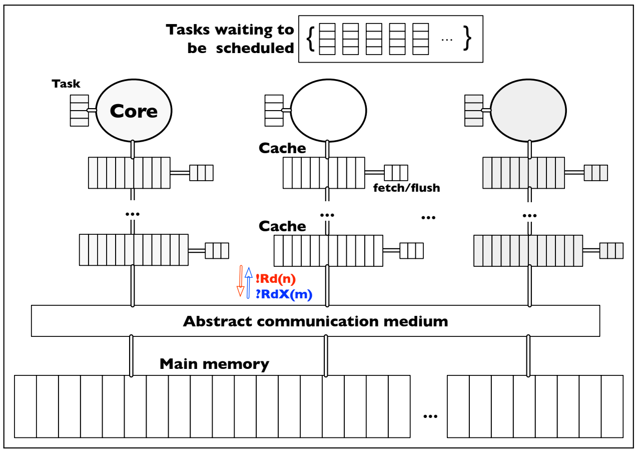


Fig. 5. Abstract model of a multicore memory system.

Task execution on a core requires memory blocks to be transferred from the main memory to the closest cache. Each cache has a pool of instructions to move memory blocks among caches and between caches and main memory. Memory blocks may exist in multiple copies in the memory system. Consistency between different copies of a memory block is ensured using the standard cache coherence protocol MSI (e.g., [57]), with which a cache line can be 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 copies of the block in other caches, to provide write access.

We summarize the operational aspects of cache coherency with the MSI protocol. To access data from a memory 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 needed if the cache is full, removing some memory block to free space. 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 $?RdX(n)$, it *invalidates* the cache line; the requests are discarded otherwise. Read and invalidation requests are broadcast instantaneously in the abstract model, reflecting that signalling on the communication medium is orders of magnitude faster than moving data to or from main memory.

4.2 A TS Model of Multicore Memory Systems

The multicore TS model describes the interactions between a core, caches, and the main memory in the multicore memory system. It further includes labeled transitions to model instantaneous broadcast. The multicore TS model is parametric in the number of cores and caches. The multicore TS model [13, 14] is shown to guarantee correctness properties for data consistency and cache coherence (see, e.g., [20, 58]), including the preservation of program order in each core, the absence of data races, and that stale data is never accessed.

In this paper, we present a simplified version of the multicore TS model which, in its original and more complex form, was introduced in [13, 14] and implemented as a correct distributed system in [9]. This simplified version allows us to focus on the main challenges of a correct distributed implementation. The runtime syntax is given in Figure 6. A configuration cf is a tuple consisting of a main memory M, cores $\overline{CR}$, caches $\overline{Ca}$ (we abstract from the task queue, which contains all tasks waiting to be scheduled). A core $(cid \bullet rst)$ with identifier cid executes *runtime statements* rst. A cache $(caid \bullet M \bullet dst)$ with identifier $caid$ has a local cache memory M and data instructions dst. We assume that the cache identifier $caid$ encodes the cid of the core to which the cache belongs and its level in the cache hierarchy. We use $Status_\perp$ to denote the extension of the set $\{mo, sh, inv\}$ of status tags with the undefined value $\perp$. Thus, a memory $M : Address \rightarrow Status_\perp$ maps addresses n to either a status tag st or to $\perp$ if the memory block with address n is not found in M.

Data access patterns dap model tasks consisting of finite sequences of **read**(n) and **write**(n) operations to address n (that is, we abstract from control flow operations for sequential composition, non-deterministic choice, repetition, and task creation). The empty access pattern is denoted ε. Cores execute runtime statements rst, which extend dap with **readBl**(n) and **writeBl**(n) to block execution while waiting for data. Caches execute *data instructions* from a multiset dst to fetch or flush a memory block with address n; here, **fetch**(n) fetches a block with address n, **fetchBl**(n) blocks execution while waiting for data, **fetchW**(n, n') waits for a memory block n' to be flushed before fetching n (this is needed when the cache is full), and **flush**(n) flushes a memory block.

The connection between the main memory and the caches of the different cores is modelled by an *abstract communication medium* which 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 captured in the TS model by label matching on transitions. For any address n, an output of the form $!Rd(n)$ or $!RdX(n)$ is broadcast and matched by its dual of the form $?Rd(n)$ or $?RdX(n)$. The syntax of the model is further detailed in [13, 14].

In the next section, we will introduce the rules that describe the interaction between core and cache incrementally when discussing their ABS implementation. (For a complete overview of the transition rules, we refer to Appendix C.) The following auxiliary functions are used in the transition rules:

- $next(caid)$ for a cache identifier $caid$ (respectively, $next(cid)$ for a core identifier cid) gives the next-level cache, if it exists, and otherwise returns $\perp$, representing 'undefined';
- $status(M, n)$ returns the status of block n in memory M or $\perp$ if the block is not found in M; and
- $select(M, n)$ determines the address where a block n should be placed in the cache memory M, based on a cache associativity (e.g., random, set associativity or direct map) and a replacement policy (e.g., random or LRU—Least Recently Used).

The function $next$ is assumed to be injective.

5 THE ABS MODEL OF THE MULTICORE MEMORY SYSTEM

This section describes the implementation of the multicore TS model by a model in ABS.[9] We explain the structural and behavioural correspondence between these two models.

5.1 The Structural Correspondence

The runtime syntax of the multicore TS model is represented in ABS by classes, user-defined datatypes and type synonyms, outlined in Figures 8–9. An ABS configuration consists of class instances to reflect the cores with their corresponding cache hierarchies and the main memory. 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 multicore TS model (and reflecting the implementation of the function *next*).

A core $(cid \bullet rst)$ in the multicore TS model corresponds to an instance of the class Core in ABS, where a field currentTask of type RstList (as defined in Figure 9) represents the current list of runtime statements. Each instance of the class Core further holds a reference to the first-level cache. An important design decision we made is to represent the runtime statements *rst* (of a core in the multicore TS model) as an ADT (see Figure 9). A core in ABS then drives the simulation by processing these runtime statements which in general require information about the first-level cache. Alternatively, a core in ABS could delegate the processing of each runtime statement by calling corresponding methods of the first-level cache. However, this latter approach complicates the required callbacks.

A cache $(caid \bullet M \bullet dst)$ in the multicore TS model corresponds to an instance of class Cache with a class parameter nextLevel which holds a reference to the next-level cache and a field cacheMemory which models the cache's memory M (of type MemMap, Figure 9). The multiset *dst* of a cache's data instructions (see Figure 6) is represented by corresponding *processes* in the message pool of the cache object in ABS. If the value of nextLevel is Nothing, then the object represents the last-level cache (in the multicore TS model, the function *next* returns $\perp$ in the case of the last-level cache.

In addition, the ABS implementation of the global synchronization with labels $!Rd(n)$ and $!RdX(n)$ used in the multicore TS model is based on the global synchronization pattern as described in Figure 4. However, instead of distinguishing between these two labels by means of an additional parameter, we introduce two corresponding broadcast interfaces:

```
interface IBroadcast {
  Bool broadcast(...);
  Unit receiveRd (IBarrier start, IBarrier end, ...)}

interface IBroadcastX {
  Bool broadcastX(...);
  Unit receiveRdX (IBarrier start, IBarrier end, ...)}
```

The class Cache then provides an implementation of both interfaces following the template of the class Broadcast in Figure 4. The ABS class Bus, on the other hand, follows the template of the Synchronizer class with the two versions sendRd and sendRdX of the method sendSync.

The object diagram in Figure 7 shows an initial configuration corresponding to the multicore memory system depicted in Figure 5.

[9]The ABS model for the multicore memory system can be found at https://abs-models.org/documentation/examples/multicore_memory/

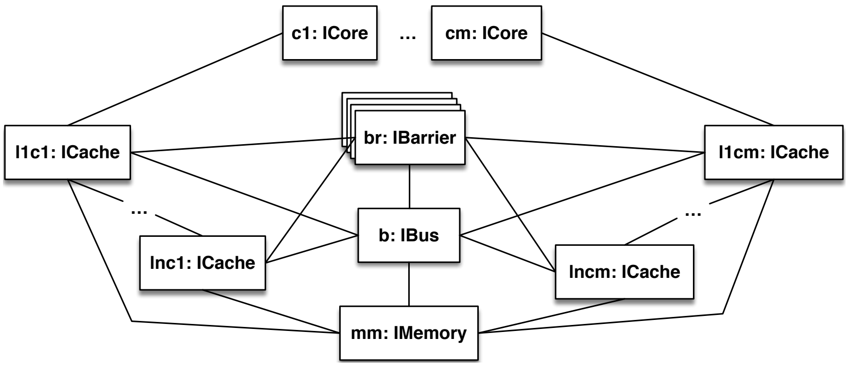


Fig. 7. Object diagram of an initial configuration.

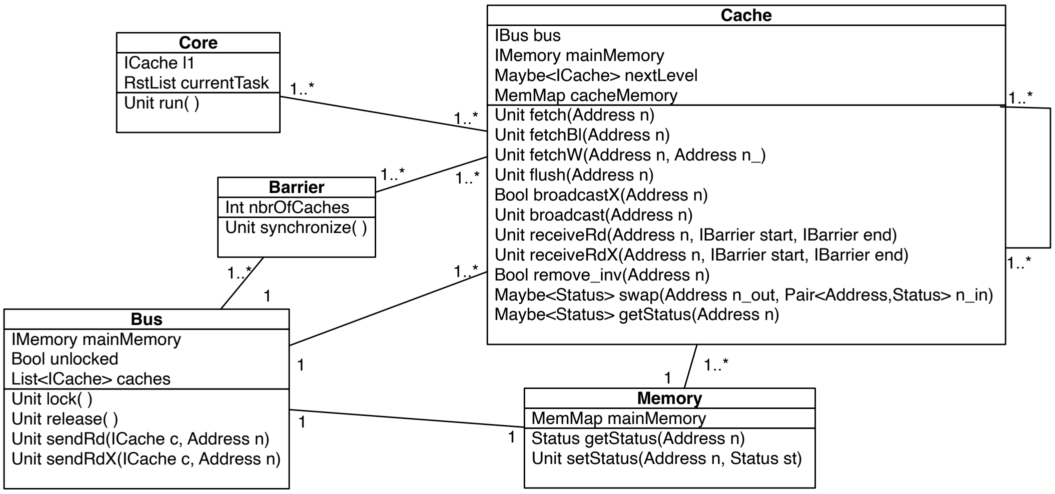

Core
ICache l1
RstList currentTask
Unit run( )
Cache
IBus bus
IMemory mainMemory
Maybe<ICache> nextLevel
MemMap cacheMemory
Unit fetch(Address n)
Unit fetchBl(Address n)
Unit fetchW(Address n, Address n_)
Unit flush(Address n)
Bool broadcastX(Address n)
Unit broadcast(Address n)
Unit receiveRd(Address n, IBarrier start, IBarrier end)
Unit receiveRdX(Address n, IBarrier start, IBarrier end)
Bool remove_inv(Address n)
Maybe<Status> swap(Address n_out, Pair<Address,Status> n_in)
Maybe<Status> getStatus(Address n)
Barrier
Int nbrOfCaches
Unit synchronize( )
Bus
IMemory mainMemory
Bool unlocked
List<ICache> caches
Unit lock( )
Unit release( )
Unit sendRd(ICache c, Address n)
Unit sendRdX(ICache c, Address n)
Memory
MemMap mainMemory
Status getStatus(Address n)
Unit setStatus(Address n, Status st)
1..*
1..*
1..*
1..*
1..*
1..*
1..*
1
1..*
1
1..*
1
1
1
1

Fig. 8. Class diagram of the ABS model.

```
data Rst = Read(Address) | ReadBl(Address) | Write(Address) | WriteBl(Address);
data Status = Sh | Mo | In;
type RstList = List<Rst>;
type Address = Int;
type MemMap = Map<Address,Status>;
```

Fig. 9. Abstract data types of the model of the multicore memory system.

5.2 The Behavioural Correspondence

We next discuss the ABS implementation of the transition rules of the multicore TS model, and the ABS synchronization patterns described in Section 2. We observe that the combination of *asynchronous method calls* and *cooperative scheduling* in ABS is crucial because of the *interleaving* inherent to the multicore TS model, which requires that objects are able to process other requests while executing a method in a controlled way; e.g., caches need to flush memory blocks while waiting for a fetch to succeed.

5.2.1 *The Annotated ABS Multicore Implementation.*

The classes Core and Cache pose the main implementation challenges. Here we explain the implementation of the run method (Figure 10) of the class Core (which is its only method) informally, in terms of its annotations (see Section 3.1). In Section 5.2.2, we introduce a formal semantics of these annotations as a high-level description of a simulation relation, which we use to prove the correctness of the ABS implementation.

The run method in the class Core may generate synchronous calls to the auxiliary methods in the class Cache, given in Figure 11. The method remove_inv instantiates the test-and-set pattern of Figure 2. The method broadcastX is an instance of the global synchronization pattern described in Section 2, Figure 4. The method sendRdX, of the global synchronizer bus, asynchronously calls the method receiveRdX (see Figure 12) of all caches (except for the calling cache), using the barrier synchronization described in Section 2.

Since the stable point at the beginning of the run method has no associated annotation, by definition (see Section 3.1), for *any* path from the beginning of the method to a next stable point (or to termination) there is *no* corresponding transition rule (of the multicore TS model). For example, there is no transition rule corresponding to the case that the run method terminates when currentTask==Nil (note that because of the structural correspondence, the corresponding core has no runtime statements *rst* to execute). Similarly, there are no transition rules corresponding to the execution of the code from the beginning of the method to the synchronous calls to the auxiliary methods remove_inv (Figure 10, Line 7) and getStatus (Figure 10, Lines 14, 18 and 31) of the first-level cache which, besides the pattern matching, only consists of the call itself.

The condition of the annotation removed==True : PrRd$_2$ (Figure 10, Line 7) associated with the synchronous call to the remove_inv method describes the path which leads from its execution and returns via the **then**-branch of the subsequent **if**-statement to the termination of the run method (after it has called itself again asynchronously). According to the annotation, the execution of this path corresponds to the PrRd$_2$ transition rule:

$$\frac{\textit{next}(\textit{cid}) = \textit{caid} \quad \textit{status}(M, n) \in \{\textit{inv}, \bot\}}{\begin{array}{c}(\textit{cid} \bullet \ \mathbf{read}(n); \textit{rst}\ ),\ (\textit{caid} \bullet \ M \bullet \textit{dst}\ ) \longrightarrow \\ (\textit{cid} \bullet \ \mathbf{readBl}(n); \textit{rst}\ ),\ (\textit{caid} \bullet \ M[n \mapsto \bot] \bullet \textit{dst} + \mathbf{fetch}(n)\ )\end{array}}\ (\textsc{PrRd}_2)$$

This rule handles the case when a core intends to read a memory block with address n, which is not found in the first-level cache. The core will then be blocked (by adding a ReadBl(n) to the currentTask, using the list constructor Cons) while waiting for the memory block to be fetched, either from the lower-level caches or main memory. The condition as returned by the remove_inv method signals that the status of the address of the first-level cache is undefined or invalid.

On the other hand, the condition removed==False describes the path which leads from its execution and return via the **else**-branch (Figure 10, Line 11), which also leads to the termination of this invocation of the run method. According to the annotation, the execution of this path corresponds to the PrRd$_1$ transition rule:

$$\frac{\textit{next}(\textit{cid}) = \textit{caid} \quad \textit{status}(M, n) \in \{\textit{sh}, \textit{mo}\}}{(\textit{cid} \bullet \ \mathbf{read}(n); \textit{rst}\ ),\ (\textit{caid} \bullet M \bullet \textit{dst}) \longrightarrow (\textit{cid} \bullet \ \textit{rst}\ ),\ (\textit{caid} \bullet M \bullet \textit{dst})}\ (\textsc{PrRd}_1)$$

```
 1  Unit run() {
 2    if (currentTask!=Nil) {
 3      switch (currentTask) {
 4        Cons(rst, rest) =>
 5          switch (rst) {
 6            Read(n) => {
 7              Bool removed = l1.remove_inv(n); // removed==True: PrRd2; removed==False: PrRd1
 8              if (removed) {
 9                l1!fetch(n);
10                currentTask = Cons(ReadBl(n),rest); }
11              else {currentTask = rest; }
12            }
13            ReadBl(n) => {
14              Maybe<Status> status = l1.getStatus(n); // status!=Nothing: PrRd3
15              if (status != Nothing) { currentTask = Cons(Read(n),rest); }
16            }
17            Write(n) => {
18              Maybe<Status> status = l1.getStatus(n); // status==Just(Mo): PrWr1
19              switch (status) {
20                Just(Mo) => { currentTask = rest; }
21                Just(Sh) => {
22                  Bool res = l1.broadcastX(n); // res==True: SynchX
23                  if (res) { currentTask = rest;  }
24                }
25                _ => {
26                  Bool removed = l1.remove_inv(n); // removed==True: PrWr3
27                  if (removed) { l1!fetch(n); currentTask = Cons(WriteBl(n),rest); }
28                } }
29            }
30            WriteBl(n) => {
31              Maybe<Status> status = l1.getStatus(n); // status!=Nothing: PrWr4
32              if (status != Nothing) { currentTask = Cons(Write(n),rest); }
33            }
34          }
35      } }
36    this ! run();
37  }
```

Fig. 10. Annotated run method of class Core.

This rule covers the case when the memory block to be read by a core is found in its first-level cache. Note that the condition as returned by the remove_inv method implies that the status of the address of the first-level cache is either shared or modified.

Next, we consider the annotation status!=Nothing : PrRd_3 of the synchronous call to the getStatus method (Figure 10, Line 14).[10] Its condition describes the execution path which leads from the execution and return of the called getStatus method to termination of the run method via the **then**-branch of the subsequent **if**-statement (Line 15). According to the annotation, the

[10]Observe that the local variable status is of type Maybe, which means that the return value is Nothing or a value of type Status wrapped around the construct Just.

```
Maybe<Status> getStatus(Address n) { return lookup(cacheMemory,n); }

Bool remove_inv(Address n) {
   Bool answer = False;
   switch (lookup(cacheMemory,n)) {
      Nothing => { answer = True; }
      Just(In) =>{ cacheMemory = removeKey(cacheMemory,n); answer = True; }
      _ => skip;
   }
   return answer;
}

Bool broadcastX(Address n) {
   Bool res = False;
   await bus!lock(); //(lookup(cacheMemory,n) ==Just(Sh)): PrWr2
   if (lookup(cacheMemory,n) ==Just(Sh)) {
      mainMemory.setStatus(n,In);
      bus.sendRdX(this, n);
      cacheMemory = put(cacheMemory,n,Mo);
      res = True;
   }
   bus.release();
   return res;
}
```

Fig. 11. Methods getStatus, remove_inv, and broadcastX of class Cache.

```
Unit receiveRdX(Address n, IBarrier start, IBarrier end) {
   // lookup(cacheMemory,n))==Just(Sh): Invalidate-One-Line;
   // lookup(cacheMemory,n))!=Just(Sh): Ignore-Invalidate-One-Line
   await start!synchronize();
   switch (lookup(cacheMemory,n)) {
      Just(Sh) => { cacheMemory = put(cacheMemory,n,In); }
      _ => skip;
   }
   end.synchronize();
}
```

Fig. 12. Annotated receiveRdX method of class Cache.

execution of this path corresponds to the PrRd$_3$ transition rule:

$$\frac{\textsc{(PrRd}_3\textsc{)} \quad\quad \mathit{next}(\mathit{cid}) = \mathit{caid} \quad n \in \mathit{dom}(M)}{(\mathit{cid} \bullet \ \mathbf{readBl}(n); \mathit{rst}\ ),\ (\mathit{caid} \bullet M \bullet \mathit{dst}) \longrightarrow (\mathit{cid} \bullet \ \mathbf{read}(n); \mathit{rst}\ ),\ (\mathit{caid} \bullet M \bullet \mathit{dst})}$$

This rule unblocks the core from waiting when *n* (i.e., the block to be read) is found in the first-level cache. On the other hand, there does not exist a transition rule which corresponds to the execution path described by the condition status==Nothing. This path leads from the execution of the called getStatus method directly to the termination of the run method without an update of the (local) state, e.g., currentTask is not updated. In other words, the evaluation of the statement readBl(n) in ABS involves *busy waiting* until the status returned by the first-level cache is defined. Alternatively, this could be implemented by calling a method of the first-level cache synchronously, which simply executes the statement **await** lookup(cacheMemory,n)!=Nothing.

The annotation of the synchronous call to method getStatus (Figure 10, Line 31) involves the transition rule

$$\frac{\textsc{(PrWr}_4\textsc{)}\qquad next(cid) = caid \quad n \in dom(M)}{(cid \bullet \boxed{\mathbf{writeBl}(n); rst}),\ (caid \bullet M \bullet dst) \rightarrow (cid \bullet \boxed{\mathbf{write}(n); rst}),\ (caid \bullet M \bullet dst)}$$

This annotation is explained in a similar manner as the annotation of the synchronous call to the getStatus method on Line 14. This rule unblocks the core from waiting when n (i.e., the block to be written) is found in the first-level cache.

We now consider the annotation status==Just(Mo) : $\textsc{PrWr}_1$ of the synchronous call to the method getStatus (Figure 10, Line 18). Its condition describes the execution path which leads from the execution of the called getStatus method and subsequent execution of the **switch**-statement to termination of the run method. According to the annotation, the execution of this path corresponds to the $\textsc{PrWr}_1$ transition rule:

$$\frac{\textsc{(PrWr}_1\textsc{)}\qquad next(cid) = caid \quad status(M, n) = mo}{(cid \bullet \boxed{\mathbf{write}(n); rst}),\ (caid \bullet M \bullet dst) \rightarrow (cid \bullet \boxed{rst}),\ (caid \bullet M \bullet dst)}$$

This rule allows a core to write to memory block n if the block is found in a modified state in the first-level cache. On the other hand, in case the condition does not hold, according to the annotation no transition rules correspond to the execution paths which lead from the execution of the called getStatus method to the next stable points, i.e., the synchronous calls to the methods broadcastX and remove_inv (Lines 22 and 26, respectively).

The condition of the annotation res==True : SynchX of the synchronous call to the broadcastX method (Figure 10, Line 22) of the first-level cache describes the path which leads from the execution of the broadcastX method, followed by the execution of the subsequent **if**-statement to termination of the run method (after an update of currentTask and calling the run method again asynchronously). According to the annotation, this path corresponds to the global synchronization rule

$$\frac{\textsc{(SynchX)}\qquad CR \notin \overline{CR_1} \quad CR, \overline{Ca}, M \xrightarrow{!RdX(n)} CR', \overline{Ca'}, M'}{\langle \overline{CR_1} \cup \{CR\},\ \overline{Ca},\ M\rangle \rightarrow \langle \overline{CR_1} \cup \{CR'\},\ \overline{Ca'},\ M'\rangle}$$

where the second premise is generated by successive applications of the rule

$$\frac{\textsc{(Synch-DistX)}\qquad Ca_1 \notin \overline{Ca} \quad CR, \overline{Ca}, M \xrightarrow{!RdX(n)} CR', \overline{Ca'}, M' \quad Ca_1 \xrightarrow{?RdX(n)} Ca_2}{CR, \overline{Ca} \cup \{Ca_1\}, M \xrightarrow{!RdX(n)} CR', \overline{Ca'} \cup \{Ca_2\}, M'}$$

This latter rule itself is triggered by the following rules

$$\frac{next(cid) = caid \quad status(M', n) = sh}{\begin{array}{c}(cid \bullet \boxed{\mathbf{write}(n); rst}),\ (caid \bullet \boxed{M'} \bullet dst),\ \boxed{M} \\ \xrightarrow{!RdX(n)} (cid \bullet \boxed{rst}),\ (caid \bullet \boxed{M'[n \mapsto mo]} \bullet dst),\ \boxed{M[n \mapsto inv]}\end{array}}\ (\textsc{PrWr}_2)$$

$$\frac{status(M, n) = sh}{caid \bullet \boxed{M} \bullet dst \xrightarrow{?RdX(n)} caid \bullet \boxed{M[n \mapsto inv]} \bullet dst}\ (\textsc{Invalidate-One-Line})$$

$$\frac{status(M, n) \in \{inv, \bot\}}{caid \bullet M \bullet dst \xrightarrow{?RdX(n)} caid \bullet M \bullet dst}\ (\textsc{Ignore-Invalidate-One-Line})$$

Together, these rules capture the broadcast mechanism for invalidation in the multicore memory system. Rule PrWr$_2$ corresponds to the case where a core writes to a memory block n that is marked as shared in its first-level cache, which requires broadcasting an invalidation message, $!RdX(n)$, to all the other caches. This is achieved by triggering the global synchronization rules SynchX and Synch-DistX. While the former identifies the core CR that broadcasts the invalidation message, the latter recursively propagates the message, $?RdX(n)$, to the other caches. Depending on the local status of memory block n in the recipient cache, the recipient cache will either invalidate the local copy of the block (Invalidate-One-Line), or ignore the message (Ignore-Invalidate-One-Line).

To explain this application of the SynchX rule, we have a closer look at the definition of the broadcastX method. Its body involves an instance of the global synchronization pattern (Figure 4). As discussed in Section 2, because of the global confluence property, we may assume that its execution is *atomic*; i.e., its execution is not interleaved with any process that it has not generated. The synchronous call to the sendRdX method of the bus generates asynchronous calls to the receiveRdX method (Figure 12) of all caches except the one that initiated the global bus synchronization. Following the general global synchronization pattern (Figure 4), these method calls are synchronized by a start and an end barrier. The two conditions of the annotation at the beginning of the receiveRdX method describe the two possible execution paths and their corresponding transition rules Invalidate-One-Line and Ignore-Invalidate-One-Line.

If the condition res==True does not hold, according to the annotation no transition rule corresponds to the execution of broadcastX. In this case the bus synchronization, as invoked by the broadcastX method (Figure 11), failed because the status of the address of the first-level cache is not shared anymore (as required by the PrWr$_2$ rule). Consequently, the processing of the statement write(n) fails and it will be processed again by the asynchronous self-call to the run method.

We conclude the informal explanation of the annotated run method with the annotation removed==True : PrWr$_3$ of the synchronous call to the method remove_inv (Figure 10, Line 26). Its condition describes the path that corresponds to the transition rule:

$$\frac{next(cid) = caid \quad status(M, n) \in \{inv, \bot\}}{\begin{array}{c}(cid \bullet \boxed{\mathbf{write}(n); rst}),\ (caid \bullet \boxed{M \bullet dst}) \rightarrow \\ (cid \bullet \boxed{\mathbf{writeBl}(n); rst}),\ (caid \bullet \boxed{M[n \mapsto \bot] \bullet dst + \mathbf{fetch}(n)})\end{array}}\ (\textsc{PrWr}_3)$$

This rule handles the case when a core tries to write to a memory block with address n, which is either invalid or not found in the first-level cache. The core will then be blocked while the memory block is fetched from the lower-level caches or from the main memory. On the other hand, according to the annotation, no transition rule corresponds to the execution path that is described by the negation of the condition (removed==False). As above, the run method terminates without having successfully processed the write(n) task, which will be evaluated again by the next asynchronous invocation of the run method. Note that this covers the case when the status returned by getStatus (Line 18) has changed; i.e., the status of the memory block is no longer undefined or invalid.

5.2.2 *Correctness of the Annotated Multicore Implementation.*

We now consider the formalization of the correctness argument for the ABS implementation of the multicore system. First, we detail the construction of the abstraction function α, which maps an ABS runtime configuration to the corresponding configuration of the TS model (see Figure 6). Recall that objects in the ABS semantics take the form $\langle oid, \sigma, p, Q\rangle$, where oid is the object identifier, σ assigns values to the object's field, p refers to the active process (either a closure (τ, s) or the *idle* process), and Q the process queue (see Section 2.2). With no loss of generality, we use $cid \in CoreId$ and $caid \in CacheId$ to symbolically reference cores and caches in ABS runtime configurations, and identify the ABS data structures defined in Figure 9 with the corresponding TS data strucures and their syntax as defined in Figure 6. To reduce notational overhead, we represent the ABS constructs, including the names of fields and local variables in italics to match the syntax of the TS model, e.g., WriteBl(n) is parsed to **writeBl**(n), Sh to *sh*, etc.

We explain the representation of the classes Core, Cache and Memory in the runtime configurations of the ABS semantics. The fields declared in these classes are summarised in Figure 13. An instance of the class Core is then represented by a runtime object $\langle cid, \sigma, p, Q\rangle$, where cid is the identifier of the core, and σ assigns to the field *l1* the object identifier *caid* for the core's first-level cache and *currentTask* to the task's runtime statements *rst*, p is the active process and Q the process queue. Note that for this class, the active process p will either be an activation of run or the *idle* process, and the process queue Q will only consist of the next call to run just before the current call to run terminates (since the interface of the class does not provide methods). An instance of class Cache is represented by a runtime object $\langle caid, \sigma, p, Q\rangle$, where *caid* is the object identifier and σ binds *bus* to the reference for the (single) instance of class Bus, *mainMemory* to the reference for the (single) instance of class Memory which represents the main memory, *nextLevel* to an identifier *caid′* for another instance of class Cache, and *cacheMemory* to a data structure M of type MemMap. The unique instance of class Memory is represented by a runtime object $\langle main, \sigma, p, Q\rangle$ where *main* is the object identifier and σ binds *mainMemory* to a data structure M of type MemMap.

Class	**Fields**
Core	*l1*, *currentTask*
Cache	*nextLevel*, *mainMemory*, *bus*, *cacheMemory*
Memory	*mainMemory*

Fig. 13. Domain of the field-assignment σ for the main ABS classes.

The abstraction function α is now defined for instances of Core, Cache and Memory as follows:

$$\begin{array}{lll}
\alpha(\langle cid, \sigma, p, Q\rangle) & = cid \bullet \sigma(currentTask) & Core \\
\alpha(\langle caid, \sigma, p, Q\rangle) & = caid \bullet \sigma(cacheMemory) \bullet \alpha(Q) + \alpha(p) & Cache \\
\alpha(\langle main, \sigma, p, Q\rangle) & = \sigma(mainMemory) & Main\ memory
\end{array}$$

Note that for objects of classes Main and Core, α abstracts from the process of the objects, while for objects of class Cache the processes of the queue Q that are generated by (asynchronous) calls to the methods fetch, fetchB, fetchW, and flush, are abstracted into the corresponding *dst* instructions in the TS model, capturing the method name and actual parameter of the processes; e.g., a process, generated by an asynchronous call to fetch(n), denoted by fetch(n), is abstracted into the corresponding **fetch**(n) instruction. This way, α abstracts from the actual process executing the call. Further, α abstracts from all the other processes. Instances of the ABS classes Barrier and Bus do not have a direct representation in the TS model.

For an ABS runtime configuration, which consists of a set of objects, the abstraction function returns the union of the abstraction of each of the objects. Given a runtime configuration G of the ABS multicore program such that every Core instance is associated with a singly-linked list of Cache instances (note that such a linked list is represented in the TS model by the auxiliary function *next*), we denote by $\alpha(G)$ the result of the above translation to its cores and caches, and its single main memory.

We now consider the execution of the ABS multicore program. Recall from Section 2.2 that the transition relation $\Rightarrow$ is between stable configurations in ABS. The following theorem states that the ABS multicore program is a correct implementation of the multicore TS model:

Theorem 2. *Let G be a reachable stable global configuration of the ABS multicore program. If $G \Rightarrow G'$ then $\alpha(G) = \alpha(G')$ or $\alpha(G) \rightarrow \alpha(G')$.*

For the proof, we need to reason about *arbitrary* ABS runtime configurations of the multicore program, e.g., abstracting from the number of cores and caches, we reason in terms of the *symbolic execution* of the ABS multicore program. We show by *static analysis* of the ABS program that the TS model provides a high-level description of the symbolic execution of the ABS multicore model, under the abstraction function α. This static analysis relies on the fact that only a finite number of paths are possible from one stable point to another in the ABS multicore model. In fact, there are no **while**-statements or synchronous self-calls between two stable points in the ABS multicore program. Further, the Boolean conditions associated with the rule annotations can be used to statically identify the corresponding paths.

The overall idea of the symbolic execution of ABS programs is based on the distinction between *local program variables* (and *fields*) and *global logical variables*, which do not appear in programs but are used to describe symbolically the values of the program variables. Logical expressions (to be distinguished from the programming expressions) are constructed from these logical variables, using the built-in and user-defined data structures. We give ABS runtime objects $\langle oid, \sigma, p, Q\rangle$ a *symbolic interpretation*, where *oid* is a logical variable (representing the symbolic value of **this**), σ binds fields to logical expressions, p is a symbolic process and Q is a logical expression denoting the process queue. A process is described symbolically by the pair (τ, S), where τ binds the local variables to logical expressions and S is a symbolic representation of a statement. The abstraction function α is then extended to the symbolic representation of the instances of the classes Core, Cache and Memory. Note that $\alpha(\langle oid, \sigma, p, Q\rangle)$, where $\langle oid, \sigma, p, Q\rangle$ denotes a symbolic representation of such an instance, is itself a symbolic representation of a concrete instance of a core, cache or memory object in the TS multicore model.

Let $C, D, \ldots$ denote symbolic states that consist of symbolic instances of the classes Core, Cache and Memory. As above, we denote by $\alpha(C)$ the result of the abstraction α to its (symbolic) cores, caches, and its single (symbolic) main memory in the TS multicore model. A *symbolic configuration* $C \mid \phi$ additionally specifies a *path condition* by a logical expression ϕ (which, as explained above, does not contain program variables). For a symbolic execution path from one stable point of the ABS multicore program to a next one, we show that the associated rule of the TS multicore model can be obtained as $\alpha(C) \rightarrow \alpha(D)$ (modulo renaming of the logical variables), where $C \mid \textbf{true} \Rightarrow D \mid \phi$ denotes a symbolic execution of this path, starting from the symbolic state C and resulting in symbolic state D. The path condition ϕ generated by the symbolic execution corresponds to the enabling conditions of the corresponding rule in the TS model.

We briefly explain the symbolic execution of the relevant rules of the ABS semantics. Assignment to an instance variable (a field) is described symbolically by the transition

$$\langle oid, \sigma, (\tau, x := e; S), Q\rangle, C \mid \phi \rightarrow \langle oid, \sigma[x := \theta(e)], (\tau, S), Q\rangle, C \mid \phi$$

where $\theta = \sigma \cup \tau$ (i.e., θ denotes the union of the substitutions σ and τ), $\theta(e)$ denotes the result of replacing every program variable x in e by $\theta(x)$ and $\sigma[x := \theta(e)]$ denotes the corresponding update of σ. We omit the similar rule for an assignment to a local variable (which will instead update τ).

The symbolic execution of Boolean conditions extends the path condition to capture an assumption about validity of the condition. In ABS, Boolean conditions can occur in **if**-, **await**- and **switch**-statements. The symbolic execution of an **if**-statement which assumes that the Boolean condition holds, is captured by

$$\langle oid, \sigma, (\tau, \mathbf{if}\ b\ \{S_0\}\{S_1\}; S), Q\rangle, C \mid \phi \rightarrow \langle oid, \sigma, (\tau, S_0; S), Q\rangle, C \mid \phi \wedge \theta(b)$$

where $\theta = \sigma \cup \tau$. Assuming that the Boolean condition does not hold, the symbolic execution of the **else**-branch is similar. The symbolic execution of an **await**-statement is captured by

$$\langle oid, \sigma, (\tau, \mathbf{await}\ b; S), Q\rangle, C \mid \phi \rightarrow \langle oid, \sigma, (\tau, S), Q\rangle, C \mid \phi \wedge \theta(b)$$

where $\theta = \sigma \cup \tau$. The symbolic execution of a **switch**-statement is captured by

$$\langle oid, \sigma, (\tau, \mathbf{switch}\ e\{\ldots e_i \Rightarrow S_i \ldots\}; S), Q\rangle, C \mid \phi \rightarrow \langle oid, \sigma, (\tau', S_i; S), Q\rangle, C \mid \phi \wedge \theta(e = e_i)$$

where τ' extends τ by binding the fresh (local) variables appearing in e_i to the corresponding subterms of e, and $\theta = \sigma \cup \tau'$.

The symbolic execution of method calls extends the path condition to capture the assumption about the identity of the callee. The symbolic execution of an *asynchronous call* to a method m with body S_0 and formal parameters $x_1, \ldots, x_n$, is captured by

$$\langle oid, \sigma, (\tau, e_0!m(\bar{e}); S), Q\rangle,\ \langle oid', \sigma', p, Q'\rangle, C \mid \phi \rightarrow \langle oid, \sigma, (\tau, S), Q\rangle,\ \langle oid', \sigma', p, Q''\rangle, C \mid \phi'$$

where $\theta = \sigma \cup \tau$ and ϕ' denotes the updated path condition $\phi \wedge \theta(e_0) = oid'$. Here, Q'' is obtained from Q' by adding the symbolic process (τ_0, S_0), where S_0 denotes the body of the method m and τ_0 binds every formal parameter x_i (for $i = 1, \ldots, n$) to the logical expression $\theta(e_i)$ (where $\bar{e} = e_1, \ldots, e_n$).

The symbolic execution of a *synchronous call* is obtained by method inlining. Let $\rightarrow^*$ denote the transitive closure of $\rightarrow$; as before, *idle* denotes the terminated process (see in Section 2.2). The symbolic execution of a method m with body S_0 and formal parameters $x_1, \ldots, x_n$ is captured by

$$\frac{\langle oid', \sigma', (\tau_0, S_0), Q'\rangle, C \mid \phi \rightarrow^* \langle oid', \sigma'', (\tau', \mathbf{return}\ e), Q''\rangle, C' \mid \phi'}{\begin{array}{c}\langle oid, \sigma, (\tau, x := e_0.m(\bar{e}); S), Q\rangle,\ \langle oid', \sigma', idle, Q'\rangle, C \mid \phi \\ \rightarrow \langle oid, \sigma[x := \theta'(e)], (\tau, S), Q\rangle,\ \langle oid', \sigma'', idle, Q''\rangle, C' \mid \phi' \wedge \theta(e_0) = oid'\end{array}}$$

assuming that x is a field (the case of a local variable is treated similarly). Here, $\theta' = \sigma'' \cup \tau'$ and, for every formal parameter x_i (for $i = 1, \ldots, n$), τ_0 binds x_i to the logical expression $\theta(e_i)$ (where $\bar{e} = e_1, \ldots, e_n$). As before, $\theta = \sigma \cup \tau$.

The correspondence between the concrete and symbolic semantics of ABS can now be formally expressed, following [22]. Let $C \mid \mathbf{true} \Rightarrow D \mid \phi$ denote the symbolic execution from one stable point to a next one, and let γ assign values to the logical variables. For any logical expression e we denote by $\gamma(e)$ its value with respect to γ, defined inductively in the standard manner. For any symbolic representation $\langle oid, \sigma, p, Q\rangle$ of an ABS object, we define $\gamma(\langle oid, \sigma, p, Q\rangle) = \langle \gamma(oid), \sigma', p', Q'\rangle$, where $\sigma'(x) = \gamma(\sigma(x))$ and $\{p'\} \cup Q'$ is obtained from $\{p\} \cup Q$ by replacing $(\tau, S) \in \{p\} \cup Q$ by (τ', S), where, as above, $\tau'(x) = \gamma(\tau(x))$. Further, for any symbolic state C we denote by $\gamma(C)$ the point-wise extension of γ.

Theorem 3. *Let G and G' be stable configurations of the ABS multicore program. For every transition $G \Rightarrow G'$, there exists an assignment γ and a symbolic transition $C \mid \mathbf{true} \Rightarrow D \mid \phi$ such that $\alpha(\gamma(C)) = \alpha(G)$, $\alpha(\gamma(D)) = \alpha(G')$, and $\gamma(\phi) = \mathbf{true}$.*

Let us now consider the *general broadcast patterns* implemented by the ABS methods broadcast and broadcastX. Assuming the correctness of the ABS implementation scheme of these patterns (which can be established by the standard proof techniques as, for example, described in [26]), for the symbolic execution we break down these patterns into, on the one hand, the symbolic execution of the methods broadcast and broadcastX, and, on the other hand, the symbolic execution of the corresponding receiveRd and receiveRdX methods, *abstracting* from the synchronization on the bus, the synchronous calls of the methods sendRd and sendRdX, and the barrier synchronization. Note that abstracting from the synchronization on the bus and the synchronous calls of the methods sendRd and sendRdX, the methods broadcast, broadcastX, receiveRd, and receiveRdX all reduce to simple TestandSet methods (which are called synchronously as explained above). For example, abstracting from the synchronization on the bus and the synchronous call of the method sendRd, the broadcast method simply reduces to a **skip**-statement. On the other hand, the synchronous call to the setStatus method of the main memory corresponds directly to a (remote) field assignment (which requires to symbolically execute the abstracted broadcastX method by the first-level cache in the context of the symbolic representation of the main memory). It remains to show that, abstracting from the barrier synchronization constructs, the symbolic execution of, e.g., the broadcastX method matches the triggering rule PrWr$_2$, and the symbolic execution of the receiverRdX method matches the rules Invalidate-One-Line and Ignore-Invalidate-One-Line of the TS model.

Proof of Theorem 2. The proof consists of symbolically executing all paths from one stable point to the next of the asynchronously called ABS methods run, fetch, fetchB, flush, and flushW. All other methods are called synchronously, and thus inlined in the execution of these methods. By the definition of the ABS multicore program, it suffices to restrict the symbolic execution of

- a local computation of a single instance of the classes Core and Cache;
- the execution of a TestandSet method of a Cache instance called synchronously by an instance of the classes Core or Cache, followed by a local computation of the caller;
- the execution of a method of the main memory called synchronously by an instance of the class Cache, followed by a local computation of the caller;
- a synchronous call of the methods broadcast, broadcastX.

To showcase the proof method, we here consider three characteristic cases (for the remaining cases of the proof, we refer to Appendix B). Each case is described by the relevant code segment in terms of a reference to figure and line numbers, and the corresponding rule of the TS model. In some cases, the execution from one stable point to the next in ABS results in the same state of the TS model, we refer to the execution in these cases as a "silent step".

Case: Figure 10, Lines 1–7, silent step. Consider the path starting from the initial stable point of the run method of a Core object and leading to the synchronous call to the remove_inv method. Let $\langle cid, \sigma, (\tau, S), Q\rangle$ be a symbolic instance of class Core, where $\sigma(\mathit{currentTask}) = \ell$ for some fresh logical variable ℓ, and Q is a logical variable representing the initial process queue. The process (τ, S) results from the activation of the run method, where S denotes the body of the run method. We specify here only the initial symbolic value of the relevant field *currentTask*, note that for example the local variables *rst* and *rest* are initialized by the **switch**-statement. It follows from the above symbolic transitions of the **if**- and **switch**-statements that

$$\langle cid, \sigma, (\tau, S), Q\rangle \mid \mathbf{true} \Rightarrow \langle cid, \sigma, ((\tau[rst \mapsto \mathit{first}(\ell), rest \mapsto \mathit{tail}(\ell)], S'), Q\rangle \mid \phi$$

where S' denotes the remaining statement to be executed and ϕ denotes the path condition ℓ!=Nil $\wedge$ $\ell = \mathit{first}(\ell); \mathit{tail}(\ell) \wedge \mathit{first}(\ell) = \mathbf{read}(n)$, where the first conjunct is generated by the **if**-statement, the second conjunct is generated by the first **switch**-statement on the field currentTask, and the last conjunct by the **switch**-statement on the local variable *rst*. Applying the abstraction function α to both configurations of the above symbolic transition gives $cid \bullet \ell$, thus this transition corresponds to a "silent step" in the TS model (note that no rule is associated with the initial stable point of run).

Case: Figure 10, Lines 7–37, PrRd$_2$. The annotation removed==True :PrRd$_2$ is associated with the synchronous call to method remove_inv on Line 7. Consider the initial symbolic configuration

$$\langle cid, \sigma, (\tau, S), Q\rangle, \langle caid, \sigma', idle, Q'\rangle \mid \mathbf{true}$$

where (τ, S) is the process about to synchronously call the remove_inv method. Note that the initial symbolic state $\langle cid, \sigma, (\tau, S), Q\rangle$ of the core coincides with the symbolic state resulting from the symbolic execution of the path discussed above. Therefore, we may assume that $\tau(rest) = \ell$ and $\sigma(currentTask) = \mathbf{read}(n); \ell$, for some logical variable ℓ. Further, let $\sigma'(cacheMemory) = M$, for some fresh logical variable M (note that we can abstract from the local variable rst since it is not used anymore). Further, note that for any symbolic representation of a Core instance, the singly linked list structure is modeled by $\sigma(l1) = next(cid)$. The condition removed==True allows us to statically identify the path which consists of first applying the transition rule for the symbolic execution of the synchronous call. This gives rise to the symbolic configuration

$$\begin{aligned}&\langle cid, \sigma, (\tau[removed \mapsto \mathrm{True}], S'), Q\rangle,\\ &\langle caid, \sigma'[cacheMemory \mapsto M[n \mapsto \bot]], idle, Q'\rangle\\ &\mid next(cid) = caid \wedge status(M, n) \in \{inv, \bot\}\end{aligned}$$

where S' denotes the remaining statement to be executed. Applying the symbolic execution rules for assignments, **if**-statements and asynchronous calls to S', we obtain the symbolic configuration

$$\begin{aligned}&\langle cid, \sigma[currentTask \mapsto \mathbf{readBl}(n); \ell], idle, Q\text{+run}\rangle,\\ &\langle caid, \sigma'[cacheMemory \mapsto M[n \mapsto \bot]], idle, Q'\text{+fetch}(n)\rangle\\ &\mid next(cid) = caid \wedge status(M, n) \in \{inv, \bot\}\end{aligned}$$

For notational convenience, we represent by fetch(n) the process resulting from the call of the fetch method with argument n (similarly, run represents above the corresponding run-process), which is added to the set of pending processes Q' (respectively Q) via the + operator. Applying α to the above initial and final symbolic configurations, we obtain

$$(cid \bullet \mathbf{read}(n); \ell),\ (caid \bullet M \bullet Q')$$

and

$$(cid \bullet \mathbf{readBl}(n); \ell),\ (caid \bullet M[n \mapsto \bot] \bullet Q'\text{+}\mathbf{fetch}(n))$$

which correspond to the configurations of the transition rule PrRd$_2$ (modulo renaming of the logical variables). The path condition corresponds to the premises of the rule.

Case: Figure 10, Lines 22–37, SynchX. We abstract from the bus and barrier synchronization as explained above. This case reduces to the following symbolic executions. First, we have the symbolic execution of the *abstracted* broadcastX method, following the path uniquely identified by the condition res==True, which starts with the initial symbolic configuration

$$\langle cid, (\tau, S), Q\rangle, \langle caid, \sigma', idle, Q'\rangle, \langle main, \sigma'', idle, Q''\rangle \mid \mathbf{true}$$

where (τ, S) is the process about to call the (abstracted) broadcastX method. The initial symbolic configuration of the core instance coincides with the symbolic configuration which results from the symbolic execution of the path leading to the call of the broadcastX method on Line 22, so we may assume that $\tau(rest) = \ell$ and $\sigma(currentTask) = \mathbf{write}(n); \ell$. Further, let $\sigma'(cacheMemory) = M$ and $\sigma''(mainMemory) = M'$, for fresh logical variables M and M'. The symbolic execution of S

following the path determined by the condition res==True then leads to

$$\begin{array}{l}\langle cid, \sigma[currentTask \mapsto \ell], idle, Q\text{+run}\rangle,\\ \langle caid, \sigma'[cachMemory \mapsto M[n \mapsto mo]], idle, Q'\rangle,\\ \langle main, \sigma''[mainMemory \mapsto M'[n \mapsto inv]], idle, Q''\rangle\\ \mid next(cid) = caid \wedge status(M, n) = sh\end{array}$$

omitting the trivial conjunct $main = main$ in the path condition, generated by the synchronous call to the method setStatus of the class Memory (which we simply model symbolically by a remote field update), assuming that $\sigma'(mainMemory) = main$. It is straightforward to check that applying α (modulo renaming of the logical variables) to the these initial and final symbolic configurations, we obtain the rule PrWr$_2$, where the premises coincide with the path condition.

Next, let $\langle caid, \sigma, (\tau, S), Q\rangle \mid \textbf{true}$ be a symbolic configuration, where S denotes the (abstracted) body of the receiveRdX method. the symbolic execution of the **switch**-statement in S leads to either

$$\langle caid, \sigma[cachMemory \mapsto M[n \mapsto inv]], idle, Q\rangle \mid status(M, n) = sh$$

or

$$\langle caid, \sigma, idle, Q\rangle \mid status(M, n) \neq sh$$

Applying the abstraction function α in the first case gives the Invalidate-One-Line rule, and in the second case the Ignore-Invalidate-One-Line rule.

To conclude this case, we observe that the symbolic execution of the path identified by the condition res!=True does not affect the above initial symbolic configuration, and corresponds to a silent step in the TS multicore model.

6 RELATED WORK

There is in general a significant gap between a TS model and its implementation in a (high-level) parallel programming language [54]. TS models (e.g., SOS [50]) succinctly formalize operational models and are well-suited for proofs, but direct implementations of such models quickly lead to very inefficient implementations. Executable semantic frameworks such as Redex [29], rewriting logic [45, 46], and $\mathbb{K}$ [51] reduce this gap, and have been used to develop executable formal models of complex languages like C [28] and Java [17]. The relationship between TS models and rewriting logic semantics has been studied [55] without proposing a general solution for synchronization by label matching. Bijo et al. implemented their multicore memory model [12] in the rewriting logic system Maude [19] using an orchestrator for label matching, but do not provide a correctness proof wrt. the TS model. 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 developed a general methodology for proving the correctness of parallel implementations of TS models in the active object language ABS. Our methodology features a new integration of these two formalisms which consists of a formal scheme for annotating ABS programs with transition rules. These annotations provide a high-level specification of the proof obligations for establishing the simulation relation between a TS model and its ABS implementation.

Our approach enables the syntax-directed verification of infinite-state ABS implementations by means of inductive reasoning. In contrast, actor models in Rebeca [56] can be verified by model-checking techniques, which involve exhaustive state-space exploration. These techniques are restricted to finite-state systems, with statically bounded queues, loops, and data types. To tackle the state-space explosion problem, the model-checking of closed actor systems can make use of a compositional verification technique that relies on a user-defined system decomposition into suitable component abstractions. These component abstractions, which allow internal messages to be ignored, are proven sound by means of a weak simulation relation. This use of weak simulation differs from our work, in which the correctness of an ABS implementation itself is defined in terms of a simulation relation between the program's semantics and a TS model describing its overall behavior. Apart from the high-level TS model, no further abstractions are required.

Correctness-preserving compilation and refinement 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 the B-method [1], which is based on refinement between abstract state machines, type-preserving translations into typed assembly languages [48], and formally verified compilers [39, 40], which proves the semantic preservation of a compiler from C to assembler code, but leaves shared-variable concurrency for future work. In contrast to these works our work specifically targets the correctness of parallel systems.

Simulation tools for cache coherence protocols can evaluate performance and efficiency on different architectures (e.g., gems [44] and gem5 [15]). 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 [47] and Sniper [18] 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 [41], our focus is on the underlying data movement rather than the response time.

7 CONCLUSION

We have introduced in this paper a methodology for proving the correctness of parallel implementations of high-level transition system specifications in the active object language ABS. The proof method consists of establishing a simulation relation between the transition system describing the semantics of the ABS program and the transition system described by the high-level specification. The proof method exploits a general global confluence property of the ABS semantics which allows to abstract from the interleaving of parallel processes and focus on the analysis of sequential code in the simulation proof. We introduced a new *symbolic* transition system for the ABS language to formalize this analysis.

As a proof of concept we applied our methodology to the ABS implementation of a transition system specification of a multicore memory system. Here we additionally exploited that the analysis of the sequential code of this implementation reduces to the symbolic execution of a *finite* number of (finite) control-flow paths. This holds for a wide class of ABS programs without affecting the computational power, because the general requirement that there exist only finite executions from one stable point to another, which enables a static, automated analysis, still allows for non-terminating behavior. In fact the multicore memory system of our case study allows non-terminating behavior as two cores might compete for the same memory address, leading to a ping-pong effect of fetching and flushing the memory block from the respective caches. In future work, we plan to implement the symbolic transition system that can generate the derivations needed for correctness arguments such as for Theorem 2 in this paper.

The ABS implementation of the high-level specification of a multicore memory system allows for the parallel simulation of such systems. In this paper we focussed on its correctness. An orthogonal concern however that often arises in parallel execution of, for example, discrete-event simulation models, 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 a simulation's to in approximating the intended (or idealized) manycore hardware. For example, as stated in [10, 11], Maude treats the whole configuration as a single term with a fixed normal form where pattern matching for rule application goes from left to right; thus, the leftmost core will always be selected as long as there exists an applicable rule for the core. Such unfair selection of cores may lead to skewed simulations. To remedy this situation and to allow exploring different execution paths, additional equations and rewrite rules are defined in Maude to randomly select a core in the configuration. In general, to obtain a faithful parallel simulation, low-level control of the underlying computing resources is required. In contrast, to ensure fairness of the simulation of the ABS multicore program, we can exploit the high-level maximal progress timed semantics [16] of ABS to ensure that cores execute at the same speed. Resource models in ABS introduce *deployment components* (DS) [34] as locations for execution that provide virtual resources (e.g., execution capacity, memory availability, network bandwidth), which are shared among the objects at this location. Any annotated statement [Cost: x] S decrements by x the available resources of its DC. Computation will stall if there are currently not enough resources available; the statement S may 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 use these resource models to assign equal (fair) resources of virtual execution speed to the simulated cores of the system. The Core objects are deployed onto separate DCs, all with the same execution capacity. The processing of each instruction has the same cost (e.g., [Cost: 1]) — a generalization, since common processor architectures execute different instructions in different speeds (cycles per instruction); e.g., `JUMP` is faster than `LOAD`. As a result, all Cores can execute up to the same number of instructions 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 itself upon the parallel simulation of N cores using a physical machine with M cores, where N is vastly greater than M.

We plan further development of this extension of the ABS multicore model with deployment components 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] Jean-Raymond Abrial. 2010. *Modeling in Event-B - System and Software Engineering.* Cambridge University Press. https://doi.org/10.1017/CBO9781139195881

[2] Gul A. Agha. 1990. *ACTORS - a model of concurrent computation in distributed systems.* MIT Press.

[3] Elvira Albert, Puri Arenas, Antonio Flores-Montoya, Samir Genaim, Miguel Gómez-Zamalloa, Enrique Martin-Martin, German Puebla, and Guillermo Román-Díez. 2014. SACO: Static Analyzer for Concurrent Objects. In *Proc. 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2014) (Lecture Notes in Computer Science, Vol. 8413)*, Erika Ábrahám and Klaus Havelund (Eds.). Springer, 562–567. https://doi.org/10.1007/978-3-642-54862-8_46

[4] Elvira Albert, Frank S. de Boer, Reiner Hähnle, Einar Broch Johnsen, Rudolf Schlatte, Silvia Lizeth Tapia Tarifa, and Peter Y. H. Wong. 2014. Formal modeling and analysis of resource management for cloud architectures: an industrial case study using Real-Time ABS. *Service Oriented Computing and Applications* 8, 4 (2014), 323–339. https://doi.org/10.1007/S11761-013-0148-0

[5] Elvira Albert, Miguel Gómez-Zamalloa, and Miguel Isabel. 2016. SYCO: a systematic testing tool for concurrent objects. In *Proc. 25th International Conference on Compiler Construction (CC 2016)*, Ayal Zaks and Manuel V. Hermenegildo (Eds.). ACM, 269–270. https://doi.org/10.1145/2892208.2892236

[6] Gregory R. Andrews. 2000. *Foundations of Multithreaded, Parallel, and Distributed Programming.* Addison Wesley.

[7] Gérard Berry and Gérard Boudol. 1992. The Chemical Abstract Machine. *Theoretical Computer Science* 96, 1 (1992), 217–248. https://doi.org/10.1016/0304-3975(92)90185-I

[8] Nikolaos Bezirgiannis, Frank S. de Boer, and Stijn de Gouw. 2017. Human-in-the-Loop Simulation of Cloud Services. In *Proc. 6th IFIP WG 2.14 European Conference on Service-Oriented and Cloud Computing (ESOCC 2017) (Lecture Notes in Computer Science, Vol. 10465)*, Flavio De Paoli, Stefan Schulte, and Einar Broch Johnsen (Eds.). Springer, 143–158.

https://doi.org/10.1007/978-3-319-67262-5_11

[9] Nikolaos Bezirgiannis, Frank S. de Boer, Einar Broch Johnsen, Ka I Pun, and Silvia Lizeth Tapia Tarifa. 2019. Implementing SOS with Active Objects: A Case Study of a Multicore Memory System. In *Proc. 22nd Intl. Conf. on Fundamental Approaches to Software Engineering (FASE 2019) (Lecture Notes in Computer Science, Vol. 11424)*, Reiner Hähnle and Wil M. P. van der Aalst (Eds.). Springer, 332–350. https://doi.org/10.1007/978-3-030-16722-6_20

[10] Shiji Bijo, Einar Broch Johnsen, Ka I Pun, Christoph Seidl, and Silvia Lizeth Tapia Tarifa. 2018. *Deployment by Construction for Multicore Architectures with Locks.* Research report 491. Department of Informatics, University of Oslo.

[11] Shiji Bijo, Einar Broch Johnsen, Ka I Pun, Christoph Seidl, and Silvia Lizeth Tapia Tarifa. 2018. Deployment by Construction for Multicore Architectures. In *Proc. 8th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation. Modeling (ISoLA 2018), Part I (Lecture Notes in Computer Science, Vol. 11244)*, Tiziana Margaria and Bernhard Steffen (Eds.). Springer, 448–465. https://doi.org/10.1007/978-3-030-03418-4_26

[12] Shiji Bijo, Einar Broch Johnsen, Ka I Pun, and S. Lizeth Tapia Tarifa. 2016. A Maude Framework for Cache Coherent Multicore Architectures. In *Proc. 11th International Workshop on Rewriting Logic and Its Applications (WRLA 2016) (Lecture Notes in Computer Science, Vol. 9942)*. Springer, 47–63. https://doi.org/10.1007/978-3-319-44802-2_3

[13] Shiji Bijo, Einar Broch Johnsen, Ka I Pun, and S. Lizeth Tapia Tarifa. 2017. A Formal Model of Parallel Execution on Multicore Architectures with Multilevel Caches. In *Proc. 14th International Conference on Formal Aspects of Component Software (FACS 2017) (Lecture Notes in Computer Science, Vol. 10487)*. Springer, 58–77. https://doi.org/10.1007/978-3-319-68034-7_4

[14] Shiji Bijo, Einar Broch Johnsen, Ka I Pun, and Silvia Lizeth Tapia Tarifa. 2019. A formal model of data access for multicore architectures with multilevel caches. *Science of Computer Programming* 179 (2019), 24–53. https://doi.org/10.1016/J.SCICO.2019.04.003

[15] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The gem5 Simulator. *SIGARCH Computer Architecture News* 39, 2 (2011), 1–7. https://doi.org/10.1145/2024716.2024718

[16] Joakim Bjørk, Frank S. de Boer, Einar Broch Johnsen, Rudolf Schlatte, and Silvia Lizeth Tapia Tarifa. 2013. User-defined schedulers for real-time concurrent objects. *Innov. Syst. Softw. Eng.* 9, 1 (2013), 29–43. https://doi.org/10.1007/S11334-012-0184-5

[17] Denis Bogdanas and Grigore Rosu. 2015. K-Java: A Complete Semantics of Java. In *Proc. 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2015)*, Sriram K. Rajamani and David Walker (Eds.). ACM, 445–456. https://doi.org/10.1145/2676726.2676982

[18] Trevor E. Carlson, Wim Heirman, and Lieven Eeckhout. 2011. Sniper: Exploring the Level of Abstraction for Scalable and Accurate Parallel Multi-core Simulation. In *Proc. International Conference for High Performance Computing, Networking, Storage and Analysis (SC)* (Seattle, Washington). ACM, Article 52, 12 pages. https://doi.org/10.1145/2063384.2063454

[19] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott (Eds.). 2007. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic.* Lecture Notes in Computer Science, Vol. 4350. Springer. https://doi.org/10.1007/978-3-540-71999-1

[20] David E. Culler, Jaswinder Pal Singh, and Anoop Gupta. 1999. *Parallel computer architecture - a hardware / software approach.* Morgan Kaufmann.

[21] Ferruccio Damiani, Reiner Hähnle, Eduard Kamburjan, and Michael Lienhardt. 2017. A Unified and Formal Programming Model for Deltas and Traits. In *Proc. 20th International Conference on Fundamental Approaches to Software Engineering (FASE 2017) (Lecture Notes in Computer Science, Vol. 10202)*, Marieke Huisman and Julia Rubin (Eds.). Springer, 424–441. https://doi.org/10.1007/978-3-662-54494-5_25

[22] Frank S. de Boer and Marcello M. Bonsangue. 2021. Symbolic execution formally explained. *Formal Aspects Comput.* 33, 4-5 (2021), 617–636. https://doi.org/10.1007/s00165-020-00527-y

[23] Frank S. de Boer, Vlad Serbanescu, Reiner Hähnle, Ludovic Henrio, Justine Rochas, Crystal Chang Din, Einar Broch Johnsen, Marjan Sirjani, Ehsan Khamespanah, Kiko Fernandez-Reyes, and Albert Mingkun Yang. 2017. A Survey of Active Object Languages. *ACM Comput. Surv.* 50, 5 (2017), 76:1–76:39. https://doi.org/10.1145/3122848

[24] Stijn de Gouw, Frank S. de Boer, Richard Bubel, Reiner Hähnle, Jurriaan Rot, and Dominic Steinhöfel. 2019. Verifying OpenJDK's Sort Method for Generic Collections. *J. Autom. Reason.* 62, 1 (2019), 93–126. https://doi.org/10.1007/S10817-017-9426-4

[25] Stijn de Gouw, Jacopo Mauro, Behrooz Nobakht, and Gianluigi Zavattaro. 2016. Declarative Elasticity in ABS. In *Proc. 5th IFIP WG 2.14 European Conference Service-Oriented and Cloud Computing (ESOCC 2016) (Lecture Notes in Computer Science, Vol. 9846)*, Marco Aiello, Einar Broch Johnsen, Schahram Dustdar, and Ilche Georgievski (Eds.). Springer, 118–134. https://doi.org/10.1007/978-3-319-44482-6_8

[26] Crystal Chang Din, Richard Bubel, and Reiner Hähnle. 2015. KeY-ABS: A Deductive Verification Tool for the Concurrent Modelling Language ABS. In *proc. 25th International Conference on Automated Deduction (CADE-25) (Lecture Notes in Computer Science, Vol. 9195)*, Amy P. Felty and Aart Middeldorp (Eds.). Springer, 517–526. https://doi.org/10.1007/978-3-319-21401-6_35

[27] Jonas Eckhardt, Tobias Mühlbauer, José Meseguer, and Martin Wirsing. 2015. Semantics, distributed implementation, and formal analysis of KLAIM models in Maude. *Sci. Comput. Program.* 99 (2015), 24–74. https://doi.org/10.1016/j.scico.2014.10.001

[28] Chucky Ellison and Grigore Rosu. 2012. An executable formal semantics of C with applications. In *Proc. 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2012)*, John Field and Michael Hicks (Eds.). ACM, 533–544. https://doi.org/10.1145/2103656.2103719

[29] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. 2009. *Semantics Engineering with PLT Redex.* The MIT Press.

[30] Reiner Hähnle. 2012. The Abstract Behavioral Specification Language: A Tutorial Introduction. In *Proc. FMCO 2012 (Lecture Notes in Computer Science, Vol. 7866).* Springer, 1–37. https://doi.org/10.1007/978-3-642-40615-7_1

[31] Carl Hewitt, Peter Bishop, and Richard Steiger. 1973. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *Proc. 3rd International Joint Conference on Artificial Intelligence (IJCAI'73).* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 235–245. http://dl.acm.org/citation.cfm?id=1624775.1624804

[32] Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. 2011. ABS: A Core Language for Abstract Behavioral Specification. In *Proc. 9th International Symposium on Formal Methods for Components and Objects (FMCO 2010) (Lecture Notes in Computer Science, Vol. 6957)*, Bernhard Aichernig, Frank S. de Boer, and Marcello M. Bonsangue (Eds.). Springer, 142–164. https://doi.org/10.1007/978-3-642-25271-6_8

[33] Einar Broch Johnsen, Jia-Chun Lin, and Ingrid Chieh Yu. 2016. Comparing AWS Deployments Using Model-Based Predictions. In *Proceedings 7th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA) (Lecture Notes in Computer Science, Vol. 9953)*, Tiziana Margaria and Bernhard Steffen (Eds.). Springer, 482–496. https://doi.org/10.1007/978-3-319-47169-3_39

[34] Einar Broch Johnsen, Rudolf Schlatte, and S. Lizeth Tapia Tarifa. 2015. Integrating deployment architectures and resource consumption in timed object-oriented models. *Journal of Logical and Algebraic Methods in Programming* 84, 1 (2015), 67–91. https://doi.org/10.1016/j.jlamp.2014.07.001

[35] Eduard Kamburjan. 2018. Detecting Deadlocks in Formal System Models with Condition Synchronization. *Electron. Commun. Eur. Assoc. Softw. Sci. Technol.* 76 (2018), 19 pages. https://doi.org/10.14279/tuj.eceasst.76.1070

[36] Eduard Kamburjan, Reiner Hähnle, and Sebastian Schön. 2018. Formal modeling and analysis of railway operations with active objects. *Sci. Comput. Program.* 166 (2018), 167–193. https://doi.org/10.1016/J.SCICO.2018.07.001

[37] Eduard Kamburjan, Marco Scaletta, and Nils Rollshausen. 2023. Deductive verification of active objects with Crowbar. *Sci. Comput. Program.* 226 (2023), 102928. https://doi.org/10.1016/j.scico.2023.102928

[38] Eduard Kamburjan and Jonas Stromberg. 2019. Tool Support for Validation of Formal System Models: Interactive Visualization and Requirements Traceability. In *F-IDE@FM (EPTCS, Vol. 310).* Open Publishing Association, 70–85. https://doi.org/10.4204/EPTCS.310.8

[39] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115. https://doi.org/10.1145/1538788.1538814

[40] Xavier Leroy. 2009. A Formally Verified Compiler Back-end. *J. Autom. Reasoning* 43, 4 (2009), 363–446. https://doi.org/10.1007/s10817-009-9155-4

[41] Yan Li, Vivy Suhendra, Yun Liang, Tulika Mitra, and Abhik Roychoudhury. 2009. Timing Analysis of Concurrent Programs Running on Shared Cache Multi-Cores. In *Proc. 30th IEEE Real-Time Systems Symposium (RTSS 2009)*, Theodore P. Baker (Ed.). IEEE Computer Society, 57–67. https://doi.org/10.1109/RTSS.2009.32

[42] Jia-Chun Lin, Jacopo Mauro, Thomas Brox Røst, and Ingrid Chieh Yu. 2017. A Model-Based Scalability Optimization Methodology for Cloud Applications. In *Proc. 7th International Symposium on Cloud and Service Computing (SC^2 2017).* IEEE Computer Society, 163–170. https://doi.org/10.1109/SC2.2017.32

[43] Jia-Chun Lin, Ingrid Chieh Yu, Einar Broch Johnsen, and Ming-Chang Lee. 2016. ABS-YARN: A Formal Framework for Modeling Hadoop YARN Clusters. In *Proc. 19th Intl. Conf. on Fundamental Approaches to Software Engineering (FASE 2016) (Lecture Notes in Computer Science, Vol. 9633)*, Perdita Stevens and Andrzej Wasowski (Eds.). Springer, 49–65. https://doi.org/10.1007/978-3-662-49665-7_4

[44] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. 2005. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. *SIGARCH Computer Architecture News* 33, 4 (2005), 92–99. https://doi.org/10.1145/1105734.1105747

[45] José Meseguer and Grigore Rosu. 2007. The rewriting logic semantics project. *Theoretical Computer Science* 373, 3 (2007), 213–237. https://doi.org/10.1016/j.tcs.2006.12.018

[46] José Meseguer and Grigore Rosu. 2013. The rewriting logic semantics project: A progress report. *Inf. Comput.* 231 (2013), 38–69. https://doi.org/10.1016/j.ic.2013.08.004

[47] Jason E. Miller, Harshad Kasture, George Kurian, Charles Gruenwald, Nathan Beckmann, Christopher Celio, Jonathan Eastep, and Anant Agarwal. 2010. Graphite: A distributed parallel simulator for multicores. In *Proc. 16th International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE Computer Society, 1–12. https://doi.org/10.1109/HPCA.2010.5416635

[48] J. Gregory Morrisett, David Walker, Karl Crary, and Neal Glew. 1999. From system F to typed assembly language. *ACM Trans. Program. Lang. Syst.* 21, 3 (1999), 527–568. https://doi.org/10.1145/319301.319345

[49] Peter Csaba Ölveczky. 2017. *Designing Reliable Distributed Systems - A Formal Methods Approach Based on Executable Modeling in Maude.* Springer. https://doi.org/10.1007/978-1-4471-6687-0

[50] Gordon D. Plotkin. 2004. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming* 60-61 (2004), 17–139.

[51] Grigore Rosu. 2017. $\mathbb{K}$: A Semantic Framework for Programming Languages and Formal Analysis Tools. In *Dependable Software Systems Engineering.* IOS Press, 186–206. https://doi.org/10.3233/978-1-61499-810-5-186

[52] Rudolf Schlatte, Einar Broch Johnsen, Eduard Kamburjan, and Silvia Lizeth Tapia Tarifa. 2021. Modeling and Analyzing Resource-Sensitive Actors: A Tutorial Introduction. In *Proc. COORDINATION 2021 (Lecture Notes in Computer Science, Vol. 12717)*. Springer, 3–19. https://doi.org/10.1007/978-3-030-78142-2_1

[53] Rudolf Schlatte, Einar Broch Johnsen, Eduard Kamburjan, and Silvia Lizeth Tapia Tarifa. 2022. The ABS simulator toolchain. *Sci. Comput. Program.* 223 (2022), 102861. https://doi.org/10.1016/J.SCICO.2022.102861

[54] Rudolf Schlatte, Einar Broch Johnsen, Jacopo Mauro, Silvia Lizeth Tapia Tarifa, and Ingrid Chieh Yu. 2018. 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 (Lecture Notes in Computer Science, Vol. 10865)*. Springer, 107–121.

[55] Traian-Florin Serbanuta, Grigore Rosu, and José Meseguer. 2009. A rewriting logic approach to operational semantics. *Inf. Comput.* 207, 2 (2009), 305–340. https://doi.org/10.1016/j.ic.2008.03.026

[56] Marjan Sirjani, Ali Movaghar, Amin Shali, and Frank S. de Boer. 2004. Modeling and Verification of Reactive Systems using Rebeca. *Fundam. Informaticae* 63, 4 (2004), 385–410. http://content.iospress.com/articles/fundamenta-informaticae/fi63-4-05

[57] Yan Solihin. 2015. *Fundamentals of Parallel Multicore Architecture* (1st ed.). Chapman & Hall/CRC. https://doi.org/10.1201/b20200

[58] Daniel J. Sorin, Mark D. Hill, and David A. Wood. 2011. *A Primer on Memory Consistency and Cache Coherence* (1st ed.). Morgan & Claypool Publishers. https://doi.org/10.2200/S00346ED1V01Y201104CAC016

[59] Gianluca Turin, Andrea Borgarelli, Simone Donetti, Ferruccio Damiani, Einar Broch Johnsen, and Silvia Lizeth Tapia Tarifa. 2023. Predicting Resource Consumption of Kubernetes Container Systems using Resource Models. *Journal of Systems & Software* 203 (Sept. 2023), 111750. https://doi.org/10.1016/j.jss.2023.111750

[60] Lars Tveito, Einar Broch Johnsen, and Rudolf Schlatte. 2020. Global Reproducibility through Local Control for Distributed Active Objects. In *Proc. 23rd International Conference on Fundamental Approaches to Software Engineering (FASE 2020) (Lecture Notes in Computer Science, Vol. 12076)*, Heike Wehrheim and Jordi Cabot (Eds.). Springer, 140–160. https://doi.org/10.1007/978-3-030-45234-6_7

A GLOBAL CONFLUENCE

We prove confluence in an abstract setting which captures the general semantics of actor languages and discuss how to embed the semantics of ABS into the setting of abstract actor semantics.

Definition 3 (Abstract Actors). *For $n \in \mathbb{N}$, we let $A_n = (\Sigma_n, M_n, T_n)$ denote an* actor*, where Σ_n is a set of* local states *and M_n a multiset of* local messages*. Each* message *$m \in M_n$ denotes a partial function $\Sigma_n \rightarrow \Sigma_n$ which describes its activation. We assume that the multisets M_n are mutually disjoint and let $M = \bigcup_n M_n$. The* local transition *function T_n is a partial function $\Sigma_n \rightarrow (\Sigma_n \times M)$.*

Assuming an infinite number of actors allows to model dynamic actor creation by activation: an actor A_i is 'dormant' if its message queue is empty and its current local state does not enable its transition relation T_i. Thus we can model the creation of an actor by sending a dormant actor a message to activate it. For a multiset q, we denote by $add(m, q)$ and $delete(m, q)$ the operations that add and remove an element m from the multiset q, respectively.

Definition 4 (Abstract Actor Semantics). *Given a set of actors $A_n = (\Sigma_n, M_n, T_n)$ (for $n \in \mathbb{N}$), a* local configuration *of actor A_n is a pair (σ, q), where $\sigma \in \Sigma_n$ and $q \in M_n \rightarrow \mathbb{N}$ is a multiset of messages in M_n. The set of* global configurations *of the actors A_n (for $n \in \mathbb{N}$) is specified by the Cartesian product $\mathcal{C} = \Pi_{n=1}^{\omega}(\Sigma_n \times (M_n \rightarrow \mathbb{N}))$. For $C \in \mathcal{C}$, let $C(i)$ denote its i'th component, so $C(i) = (\sigma_i, q_i)$. The* global transition relation *$C \rightarrow_i C'$ is defined as follows:*

- *$T_i(\sigma_i) = (\sigma_i', m)$, with $m \in M_j$, and*
  - *$C(n) = C'(n)$, for $n \notin \{i, j\}$,*
  - *$C'(i) = (\sigma_i', q_i)$ and $C'(j) = (\sigma_j, add(m, q_j))$, if $i \neq j$,*
  - *$C'(i) = (\sigma_i', add(m, q_i))$, if $i = j$.*
- *$T_i(\sigma_i)$ is undefined and there exists a message $m \in q_i$ such that*
  - *$C(n) = C'(n)$, for $n \neq i$,*
  - *$C'(i) = (m(\sigma_i), delete(m, q_i))$.*

Note that in the second clause we (implicitly) assume that $m(\sigma_i)$ is defined (that is, a message can only be activated when it is enabled).

We observe that abstract actor semantics have the following properties:

- Encapsulation the local state: only local transitions affect local state;
- Monotonicity of local transitions, which are not affected by adding messages: only local transitions remove a message from a queue; and
- The queues obey the basic algebraic laws of adding and deleting elements from a multiset:
  - $add(m, add(m', M)) = add(m', add(m, M))$
  - $add(m, delete(m', M)) = delete(m', add(m, M))$, for $m' \in M$ (that is, $M(m') > 0$)

These observations allow us to prove the following confluence property for abstract actor semantics:

Lemma 1 (Abstract actor confluence). *Let C, C_1 and C_2 be global configurations of actors A_n (for $n \in \mathbb{N}$). If $C \rightarrow_i C_1$ and $C \rightarrow_j C_2$, for $i \neq j$, then there exists a configuration C' such that $C_1 \rightarrow_j C'$ and $C_2 \rightarrow_i C'$.*

Proof. To prove the lemma, we provide the construction of C'. Let $C(n) = (\sigma_n, q_n)$, for every n, and $C_1(i) = (\sigma_i', q_i')$ and $C_2(j) = (\sigma_j', q_j')$. Then, for every $n \notin \{i, j\}$, $C'(n) = (\sigma_n, q_n')$ where q_n' results from q_n by adding the message $m \in M_n$ for which $T_k(\sigma_k) = (\sigma_k', m)$, $k \in \{i, j\}$. For $n \in \{i, j\}$, we define $C'(n) = (\sigma_n', q')$, where q' results from q_n' by adding the message $m \in M_n$ for which $T_k(\sigma_k) = (\sigma_k', m)$, $k \in \{i, j\}$ and $k \neq n$. It is straightforward to check that C' satisfies the above confluence property. ◻

To conclude this section, we discuss how to embed the semantics of ABS into the setting of abstract actors. The local state σ_n of an instance n of an ABS class specifies the statement to be executed as well as the values of the instance variables and the local variables. As described above, global transitions are generated from local transitions $T_n(\sigma_n) = (\sigma'_n, m)$, where m denotes a message which consists of a method name and an assignment of values to the local variables (including the actual parameters). These local transitions define the usual *small-step* semantics of the specified statement in the local state. For example, to describe a basic assignment, we assume an empty message ϵ so that if $T_i(\sigma_i) = (\sigma'_i, \epsilon)$ then $C \longrightarrow_i C'$, where $C(i) = (\sigma_i, q_i)$, $C(n) = C'(n)$, for $n \neq i$, and $C'(i) = (\sigma'_i, q_i)$.

Of particular interest is the embedding of method calls, **await**-statements and futures. An (asynchronous) method call can simply be modeled by a message which specifies the method (name) and the actual parameters (the sets of messages for different actors can be made disjoint by assuming that the actor identity is among the actual parameters).

We can model **await**-statements by sending a message to the actor itself. To this end, we wrap the continuation of each **await**-statement in a corresponding, auxiliary method. Calling such a method generates a message m that can only be activated in a local state that satisfies the Boolean guard of the **await**-statement, that is, $m(\sigma)$ is defined only if the local state satisfies the Boolean guard. In general, the activation of a message updates the statement to be executed (as specified by the corresponding method) and the local variables.

Futures can be represented by auxiliary actors that provide two methods Set and Get. The future generated by an asynchronous method call is passed as an actual parameter of the corresponding message. The return value can then be transmitted by calling the Set method of the future. This value can be retrieved by calling the Get method with the calling actor as actual parameter. This allows for a *callback* of a special auxiliary method which finally stores the returned value, assuming a suitable additional data structure as part of the local state (e.g., a set of pairs of a completed future and its value). An **await**-statement on a future then can be modeled by an invocation of the Get method of the future and a message that wraps the continuation of the await statement and that can only be activated in a local state which stores the return value (obtained by the callback described above). A **get**-operation on a future can be modeled by a semaphore which blocks the actor till the future has been completed. More specifically, this semaphore holds the future uniquely associated with the method invocation that is executing the **get**-operation. In general, the activation of any message of a user-defined ABS actor, *different from the above callback*, requires that the semaphore is 'free' or that the corresponding method invocation holds the semaphore.

Observe that the *stable points* of an ABS object (Definition 1) precisely characterize the states in which the local transition function of the corresponding abstract actor is undefined, so the further execution of the actor depends on its queue.

B CORRECTNESS

Section 5.2.2 has given the general explanation of the proof method and the analysis of three representative cases of the proof. We here consider the remaining cases of the proof. Recall that the confluence property of ABS (Theorem 1) allows us to reason about the parallel execution of objects in terms of the sequential execution between stable points in each object separately. The symbolic execution can then follow standard syntax-driven symbolic transition steps for sequential program statements, as described in Section 5.2.2. Therefore, we do not detail the individual steps but describe the initial and final symbolic configurations of the symbolic execution between stable points in the ABS program.

Proof of Theorem 2. It suffices to verify the annotations of the run method of class Core and the methods of the Cache class that correspond to the *dst* instructions in terms of the simulation relation α. These methods are fetch, fetchB, flush, and flushW. For each of these methods, we perform a syntax-directed analysis of all paths from one stable point to the next, which amounts to reasoning about the symbolic execution of sequential fragments of code. For each method and each case, we specify the corresponding figure with the ABS code, and line numbers for the sequential segment between the stable points considered, and the corresponding rule in the TS model (see Appendix C). In the symbolic execution of the code fragments of the ABS program we assume that the data structures of ABS coincide with those of the TS model, e.g., the ABS lookup function coincides with the *status* function of the TS model. Further, we abstract from the low-level symbolic transitions which are straightforward instances of the general rules, but tedious to detail.

Method run*, Figure 10.* For the run method, we need to distinguish the different cases of the **switch**-statement on Line 3: Read(n), ReadBl(n), Write(n) and WriteBl(n). For each of these, there are stable points associated with the synchronous calls to the first-level cache (Lines 7, 14, 18, 22, 26 and 31) and there are **if**- and **switch**-statements that introduce further branching.

Case: Figure 10, Lines 1–7, silent step. Covered in Section 5.2.2.

Case: Figure 10, Lines 7–37, PrRd_2. Covered in Section 5.2.2.

Case: Figure 10, Lines 7–37, PrRd_1. The annotation removed==False : PrRd_1 is associated with the synchronous call to method remove_inv on Line 7. Consider the initial symbolic configuration

$$\langle cid, \sigma_1, (\tau, S), Q_1\rangle, \langle caid, \sigma_2, idle, Q_2\rangle \mid \textbf{true}$$

where (τ, S) is the process about to synchronously call the remove_inv method. Note that the initial symbolic state $\langle cid, \sigma_1, (\tau, S), Q_1\rangle$ of the core coincides with the symbolic state resulting from the symbolic execution of the path discussed in the case right above. Therefore, we may assume that $\tau(rest) = \ell$ and $\sigma_1(currentTask) = \textbf{read}(n); \ell$, for some logical variable ℓ. Let $\sigma_2(cacheMemory) = M$, for some fresh logical variable M (note that we can abstract from the local variable *rst* since it is not used anymore). Note that for any symbolic representation of a Core instance, the singly linked list structure is modeled by $\sigma_1(l1) = next(cid)$. The condition removed==False allows us to statically identify the path which consists of first applying the transition rule for the symbolic execution of the synchronous call. This gives rise to the symbolic configuration

$$\begin{aligned}&\langle cid, \sigma_1, (\tau[removed \mapsto \mathsf{False}], S'), Q_1\rangle,\ \langle caid, \sigma_2, idle, Q_2\rangle\\ &\mid next(cid) = caid \wedge status(M, n) \in \{sh, mo\}\end{aligned}$$

where S' denotes the remaining statement to be executed. Applying symbolic execution rules for assignments, **if**-statements and asynchronous calls to S', we obtain the symbolic configuration

$$\begin{aligned}&\langle cid, \sigma_1[currentTask \mapsto \ell], idle, Q_1{+}\mathsf{run}\rangle,\ \langle caid, \sigma_2, idle, Q_2\rangle\\ &\mid next(cid) = caid \wedge status(M, n) \in \{sh, mo\}\end{aligned}$$

Let run denote the corresponding run-process, which is added to the set of pending processes Q_1 via the + operator. Applying α to the above initial and final symbolic configurations, we obtain

$$(cid \bullet \mathbf{read}(n); \ell),\ (caid \bullet M \bullet Q_2)$$

and

$$(cid \bullet \ell),\ (caid \bullet M \bullet Q_2)$$

which correspond to the configurations of the transition of rule PrRd$_1$ (modulo renaming of the logical variables). The path condition corresponds to the premises of the rule.

Case: Figure 10, Lines 1–14, silent step. This case is the same as the case for the silent step captured by Lines 1–7 proven in Section 5.2.2. Consider the path starting from the initial stable point of the run method of a Core object and leading to the synchronous call to the getStatus method. Let $\langle cid, \sigma, (\tau, S), Q\rangle$ be a symbolic instance of class Core, where $\sigma(currentTask) = \ell$ for some fresh logical variable ℓ, and Q is a logical variable representing the initial process queue. The process (τ, S) results from the activation of the run method, where S denotes the body of the run method. We specify here only the initial symbolic value of the relevant field *currentTask*, note that for example the local variables *rst* and *rest* are initialized by the **switch**-statement. It follows from the above symbolic transitions of the **if**- and **switch**-statements that

$$\langle cid, \sigma, (\tau, S), Q\rangle \mid \mathbf{true} \Rightarrow \langle cid, \sigma, (\tau[rst \mapsto first(\ell), rest \mapsto tail(\ell)], S'), Q\rangle \mid \phi$$

where S' denotes the remaining statement to be executed and ϕ the path condition ℓ!=Nil $\wedge\ \ell = first(\ell); tail(\ell) \wedge first(\ell) = \mathbf{readBl}(n)$, where the first conjunct is generated by the **if**-statement, the second conjunct by the first **switch**-statement on the field currentTask, and the last conjunct by the **switch**-statement on the local variable *rst*. Applying the abstraction function α to both configurations of this symbolic transition gives $cid \bullet \ell$, thus this transition corresponds to a silent step in the TS model (note that no rule is associated with the initial stable point of the run method).

Case: Figure 10, Lines 14–37, PrRd$_3$. The annotation status!=Nothing: PrRd$_3$ is associated with the synchronous call to method getStatus on Line 14. Consider the initial symbolic configuration

$$\langle cid, \sigma_1, (\tau, S), Q_1\rangle, \langle caid, \sigma_2, idle, Q_2\rangle \mid \mathbf{true}$$

where (τ, S) is the process about to synchronously call the getStatus method. Note that the initial symbolic state $\langle cid, \sigma_1, (\tau, S), Q_1\rangle$ of the core coincides with the symbolic state resulting from the symbolic execution of the path discussed above in the case right above. Therefore, we may assume that $\tau(rest) = \ell$ and $\sigma_1(currentTask) = \mathbf{readBl}(n); \ell$, for some logical variable ℓ. In addition, let $\sigma_2(cacheMemory) = M$, for some fresh logical variable M (note that we can abstract from the local variable *rst* since it is not used anymore). Note further that for any symbolic representation of a Core instance, the singly linked list structure is modeled by $\sigma_1(l1) = next(cid)$. The condition status!=Nothing allows us to statically identify the path which first applies the transition rule for the symbolic execution of the synchronous call. This gives rise to the symbolic configuration

$$\langle cid, \sigma_1, (\tau', S'), Q_1\rangle,\ \langle caid, \sigma_2, idle, Q_2\rangle \mid next(cid) = caid \wedge n \in dom(M)$$

where τ' extends τ by binding *status* to some logical variable that does not equal $\perp$, indicating that the given address n can be found in M, and S' is the remaining statement of the method. Proceeding with the symbolic execution of the statement S, we obtain (by applying the rules for **if**-statements, assignments and asynchronous calls)

$$\begin{aligned}&\langle cid, \sigma_1[currentTask \mapsto \mathbf{read}(n); \ell], idle, Q_1+\text{run}\rangle,\ \langle caid, \sigma_2, idle, Q_2\rangle\\ &\quad \mid next(cid) = caid \wedge n \in dom(M)\end{aligned}$$

Applying α to the above initial and final symbolic configurations, we obtain

$$(cid \bullet \mathbf{readBl}(n); \ell),\ (caid \bullet M \bullet Q_2)$$

and

$$(cid \bullet \textbf{read}(n);\ell),\ (caid \bullet M \bullet Q_2)$$

which correspond to the configurations of the transition of rule PrRd$_3$ (modulo renaming of the logical variables). The path condition corresponds to the premises of the rule.

To conclude this case, we observe that the symbolic execution of the path identified by the condition status==Nothing does not affect the above initial symbolic configuration, and corresponds to a silent step in the TS multicore model.

Case: Figure 10, Lines 1–18, silent step . This case is the same as the case for the silent step captured by Lines 1–14 above. Consider the path starting from the initial stable point of the run method of a Core object and leading to the synchronous call to the getStatus method. Let $\langle cid, \sigma, (\tau, S), Q\rangle$ be a symbolic instance of class Core, where $\sigma(currentTask) = \ell$ for some fresh logical variable ℓ, and Q is a logical variable representing the initial process queue. The process (τ, S) results from the activation of the run method, where S denotes the body of the run method. We specify here only the initial symbolic value of the relevant field *currentTask*, note that for example the local variables *rst* and *rest* are initialized by the **switch**-statement. It follows from the above symbolic transitions of the **if**- and **switch**-statements that

$$\langle cid, \sigma, (\tau, S), Q\rangle \mid \textbf{true} \Rightarrow \langle cid, \sigma, (\tau[rst \mapsto first(\ell), rest \mapsto tail(\ell)], S'), Q\rangle \mid \phi$$

where S' denotes the remaining statement to be executed and ϕ the path condition ℓ!=Nil $\wedge\ \ell = first(\ell); tail(\ell) \wedge first(\ell) = \textbf{write}(n)$, where the first conjunct is generated by the **if**-statement, the second conjunct is generated by the first **switch**-statement on the field currentTask, and the last conjunct by the **switch**-statement on the local variable *rst*. Applying the abstraction function α to both configurations of the above symbolic transition gives $cid \bullet \ell$, thus this transition corresponds to a silent step in the TS model. (No rule is associated with the initial stable point of the run method.)

Case: Figure 10, Lines 18–37, PrWr$_1$. The annotation status==Just(Mo):PrWr$_1$ is associated with the synchronous call to getStatus on Line 18. Consider the initial symbolic configuration

$$\langle cid, \sigma_1, (\tau, S), Q_1\rangle,\ \langle caid, \sigma_2, idle, Q_2\rangle\ \mid \textbf{true}$$

where (τ, S) is the process about to synchronously call the getStatus method. The initial symbolic state $\langle cid, \sigma_1, (\tau, S), Q_1\rangle$ of the core coincides with the symbolic state resulting from the symbolic execution of the path discussed above in the case right above. Therefore, we may assume that $\tau(rest) = \ell$ and $\sigma_1(currentTask) = \textbf{write}(n);\ell$, for some logical variable ℓ. In addition, let $\sigma_2(cacheMemory) = M$, for some fresh logical variable M (note that we can abstract from the local variable *rst* since it is not used anymore). Further, for any symbolic representation of a Core instance, the singly linked list structure is modeled by $\sigma_1(l1) = next(cid)$. The condition status==Just(Mo):PrWr$_1$ allows us to statically identify the path which first applies the transition rule for the symbolic execution of the synchronous call. This gives rise to the symbolic configuration

$$\langle cid, \sigma_1, (\tau', S'), Q_1\rangle,\ \langle caid, \sigma_2, idle, Q_2\rangle \mid next(cid) = caid \wedge status(M, n) = mo$$

where τ' extends τ by binding *status* to some logical variable that equals Just(Mo), indicating that the given address n can be found modified in M, and S' is the remaining statement of the method. Proceeding with the symbolic execution of the statement S', we obtain (by applying the rules for assignments, **switch**-statements and asynchronous calls)

$$\begin{aligned}&\langle cid, \sigma_1[currentTask \mapsto \ell], idle, Q_1\text{+run}\rangle,\ \langle caid, \sigma_2, idle, Q_2\rangle\\ &\quad \mid next(cid) = caid \wedge status(M, n) = mo\end{aligned}$$

Let run represent the corresponding run-process, which is added to the set of pending processes Q_1 via the + operator. Applying α to the above initial and final symbolic configurations, we obtain

$$(cid \bullet \textbf{write}(n); \ell),\ (caid \bullet M \bullet Q_2)$$

and

$$(cid \bullet \ell),\ (caid \bullet M \bullet Q_2)$$

which correspond to the configurations of the transition of rule PrWr$_1$ (modulo renaming of the logical variables). The path condition corresponds to the premises of the rule.

Case: Figure 10, Lines 18–22, silent step . We start with the initial symbolic configuration

$$\langle cid, \sigma_, (\tau, S), Q\rangle,\ \langle caid, \sigma', idle, Q'\rangle\ \mid \textbf{true}$$

where (τ, S) is the process about to synchronously call the getStatus method. Note that the initial symbolic state $\langle cid, \sigma, (\tau, S), Q\rangle$ of the core coincides with the symbolic state resulting from the symbolic execution of the path discussed above in the case right above. Therefore, we may assume that $\tau(rest) = \ell$ and $\sigma(currentTask) = \textbf{write}(n); \ell$, for some logical variable ℓ. In addition, let $\sigma'(cacheMemory) = M$, for some fresh logical variable M (note that we can abstract from the local variable rst since it is not used anymore). As observed above, we have that for any symbolic representation of a Core instance, the singly linked list structure is modeled by $\sigma(l1) = next(cid)$. Applying the symbolic transition for the **switch**-statement, we obtain the symbolic configuration

$$\langle cid, \sigma, (\tau', S'), Q\rangle,\ \langle caid, \sigma', idle, Q'\rangle \mid next(cid) = caid \wedge status(M, n) = sh\}$$

where τ' extends τ by binding $status$ to some logical variable and S' is the remaining statement of the method. Applying the abstraction function α, we obtain

$$\begin{aligned}\alpha(\langle cid, \sigma, (\tau, S), Q\rangle) &= cid \bullet \textbf{write}(n); \ell\\ \alpha(\langle caid, \sigma', idle, Q'\rangle) &= caid \bullet M \bullet Q'\\ \alpha(\langle cid, \sigma, (\tau', S'), Q\rangle) &= cid \bullet \textbf{write}(n); \ell\end{aligned}$$

This transition corresponds to a silent step in the TS model.

Case: Figure 10, Lines 22–37, SynchX. Covered in Section 5.2.2, SynchX subsumes PrWr$_2$.

Case: Figure 10, Lines 18–26, silent step. We start with the initial symbolic configuration

$$\langle cid, \sigma_1, (\tau, S), Q_1\rangle,\ \langle caid, \sigma_2, idle, Q_2\rangle \mid \textbf{true}$$

where $\sigma_1(currentTask) = \textbf{write}(n); \ell$ and $\sigma_2(cacheMemory) = M$ for logical variables ℓ and M, Q_1 and Q_2 are symbolic representations of the process queues, and (τ, S) is the symbolic representation of the activation of the run method. Applying the symbolic transition for the **switch**-statement, we obtain the following symbolic configuration

$$\langle cid, \sigma_1, (\tau', S'), Q_1\rangle,\ \langle caid, \sigma_2, idle, Q_2\rangle \mid next(cid) = caid \wedge status(M, n) \in \{inv, \bot\}$$

where τ' extends τ by binding $status$ to some logical variable and S' is the remaining statement of the method. Applying the abstraction function α, we obtain

$$\begin{aligned}\alpha(\langle cid, \sigma_1, (\tau, S), Q_1\rangle) &= cid \bullet \textbf{write}(n); \ell\\ \alpha(\langle caid, \sigma_2, idle, Q_2\rangle) &= caid \bullet M \bullet Q_2\\ \alpha(\langle cid, \sigma_1, (\tau', S'), Q_1\rangle) &= cid \bullet \textbf{write}(n); \ell\end{aligned}$$

This transition corresponds to a silent step in the TS model.

Case: Figure 10, Lines 26–37, PrWr$_3$. The annotation removed==True : PrWr$_3$ is associated with the synchronous call to method remove_inv on Line 26. Consider the initial symbolic configuration

$$\langle cid, \sigma_1, (\tau, S), Q_1\rangle,\ \langle caid, \sigma_2, idle, Q_2\rangle \mid \textbf{true}$$

where $\sigma_1(\mathit{currentTask}) = \textbf{write}(n);\ell$ and $\sigma_2(\mathit{cacheMemory}) = M$ for logical variables ℓ and M, Q_1 and Q_2 are symbolic representations of the process queues, and (τ, S) is the process about to synchronously call the remove_inv method. The symbolic transition for the **if**-statement and the asynchronous call then gives us the following symbolic configuration

$$\langle cid, \sigma_1', idle, Q_1\rangle,\ \langle caid, \sigma_2', idle, Q_2 + \mathrm{fetch}(n)\rangle \mid next(cid) = caid \wedge status(M, n) \notin \{\bot, in\}$$

where $\sigma_1'(\mathit{currentTask}) = \textbf{writeBl}(n);\ell$ and $\sigma_2' = \sigma_2[\mathit{cacheMemory} \mapsto M[n \mapsto \bot]]$. Applying the abstraction function α, we obtain

$$\begin{aligned}
&\alpha(\langle cid, \sigma_1, (\tau, S), Q_1\rangle) = cid \bullet \textbf{write}(n);\ell\\
&\alpha(\langle caid, \sigma_2, idle, Q_2\rangle) = caid \bullet M \bullet Q_2\\
&\alpha(\langle cid, \sigma_1', idle, Q_1\rangle) = cid \bullet \textbf{writeBl}(n);\ell\\
&\alpha(\langle caid, \sigma_2, idle, Q_2 + \mathrm{fetch}(n)\rangle) = caid \bullet M[n \mapsto \bot] \bullet Q_2 + \textbf{fetch}(n)
\end{aligned}$$

Consequently, the symbolic transition corresponds to the transition of $\textsc{PrWr}_3$ in the TS model.

To conclude this case, we observe that the symbolic execution of the path identified by the condition removed==False does not affect the above initial symbolic configuration, and corresponds to a silent step in the TS multicore model.

Case: Figure 10, Lines 1–31, silent step. Consider the path starting from the initial stable point of the run method of a Core object and leading to the synchronous call to the getStatus method. Let $\langle cid, \sigma, (\tau, S), Q\rangle$ be a symbolic instance of class Core, where $\sigma(\mathit{currentTask}) = \ell$ for some fresh logical variable ℓ, and Q is a logical variable representing the initial process queue. The process (τ, S) results from the activation of the run method, where S denotes the body of the run method. We specify here only the initial symbolic value of the relevant field *currentTask*, note that for example the local variables *rst* and *rest* are initialized by the **switch**-statement. It follows from the above symbolic transitions of the **if**- and **switch**-statements that

$$\langle cid, \sigma, (\tau, S), Q\rangle \mid \textbf{true} \Rightarrow \langle cid, \sigma, ((\tau[rst \mapsto first(\ell), rest \mapsto tail(\ell)], S'), Q\rangle \mid \phi$$

where S' denotes the remaining statement to be executed and ϕ the path condition ℓ!=Nil $\wedge\ \ell = first(\ell); tail(\ell) \wedge first(\ell) = \textbf{writeBl}(n)$, where the first conjunct is generated by the **if**-statement, the second conjunct is generated by the first **switch**-statement on the field currentTask, and the last conjunct by the **switch**-statement on the local variable *rst*. Applying the abstraction function α to both configurations of the above symbolic transition gives $cid \bullet \ell$, thus this transition corresponds to a silent step in the TS model. (No rule is associated with the initial stable point of the run method.)

Case: Figure 10, Lines 31–37, $\textsc{PrWr}_4$. The annotation status!=Nothing:$\textsc{PrWr}_4$ is associated with the synchronous call to method getStatus on Line 31. Consider the initial symbolic configuration

$$\langle cid, \sigma_1, (\tau, S), Q_1\rangle, \langle caid, \sigma_2, idle, Q_2\rangle \mid \textbf{true}$$

where (τ, S) is the process about to synchronously call the getStatus method. Note that the initial symbolic state $\langle cid, \sigma_1, (\tau, S), Q_1\rangle$ of the core coincides with the symbolic state resulting from the symbolic execution of the path discussed above. Therefore, we may assume that $\tau(rest) = \ell$ and $\sigma_1(\mathit{currentTask}) = \textbf{writeBl}(n);\ell$, for some logical variable ℓ. Further, let $\sigma_2(\mathit{cacheMemory}) = M$, for some fresh logical variable M (note that we can abstract from the local variable *rst* since it is not used anymore). Further, note that for any symbolic representation of a Core instance, the singly linked list structure is modeled by $\sigma_1(l1) = next(cid)$. The condition status!=Nothing allows us to statically identify the path which consists of first applying the transition rule for the symbolic execution of the synchronous call. Proceeding with the symbolic execution of the statement S, we obtain (by applying the rules for **if**-statement and assignment)

$$\begin{aligned}
&\langle cid, \sigma_1[\mathit{currentTask} \mapsto \textbf{write}(n);\ell], idle, Q_1{+}\mathrm{run}\rangle, \langle caid, \sigma_2, idle, Q_2\rangle\\
&\quad \mid next(cid) = caid \wedge n \in dom(M)
\end{aligned}$$

Applying α to the above initial and final symbolic configurations, we obtain

$$(cid \bullet \textbf{writeBl}(n); \ell),\ (caid \bullet M \bullet Q_2)$$

and

$$(cid \bullet \textbf{write}(n); \ell),\ (caid \bullet M \bullet Q_2)$$

which correspond to the configurations of the transition of rule $\textsc{PrWr}_4$ (modulo renaming of the logical variables). The path condition corresponds to the premises of the rule.

To conclude this case, we observe that the symbolic execution of the path identified by the condition status==Nothing does not affect the above initial symbolic configuration, and corresponds to a silent step in the TS multicore model.

Method fetch*, Figure 14.* This method involves synchronous calls to the auxiliary methods broadcast (Figure 14) and swap (Figure 15). The method broadcast describes an instance of the global synchronization pattern (Figure 4). The method sendRd of the bus asynchronously calls the method receiveRd, see Figure 16, of all caches (except for the calling cache), using the barrier synchronization (again, see Figure 4, Section 2). The swap method is an instance of the test-and-set pattern, shown in Figure 2.

Case: Figure 14, Lines 1–4, silent step. Consider the path starting from the initial stable point of the fetch method for a symbolic parameter value n, and leading to the synchronous call to the remove_inv method on Line 4. Let $\langle cid, \sigma, (\tau, S), Q\rangle$ be a symbolic instance of class Cache, where $\sigma(cacheMemory) = M$ for some fresh logical variables $caid'$ and M, and Q is a logical variable representing symbolically the process queue. The process (τ, S) results from the activation of the fetch method, where S denotes the body of the fetch method. It follows from the symbolic execution of the **switch**-statement that

$$\langle caid, \sigma, (\tau, S), Q\rangle \mid \textbf{true} \Rightarrow \langle caid, \sigma, (\tau[nextCache \mapsto next(caid)], S'), Q\rangle \mid next(caid)] \neq \bot,$$

where S' denotes the remaining statement to be executed and $\sigma(nextLevel) = next(caid)$ (which models the singly-linked list structure). Applying the abstraction function α to both configurations of the above symbolic transition gives $caid \bullet M \bullet Q + \mathsf{fetch}(n)$ (note that the active process is added as a *dst* instruction), thus this transition corresponds to a silent step in the TS model.

Case: Figure 14, Lines 4–7, LC-Miss. The annotation removed==True : LC-Miss is associated to the synchronous call to remove_inv on Line 4. We consider the initial symbolic configuration

$$\langle caid_1, \sigma_1, (\tau, S), Q_1\rangle,\ \langle caid_2, \sigma_2, idle, Q_2\rangle\ \mid \textbf{true}$$

where the process (τ, S) is about to synchronously call the remove_inv method. Note that the initial symbolic state of $caid_1$ corresponds to the symbolic state resulting from the symbolic execution of the path discussed for the previous case above, so we may assume that $\tau(nextCache) = next(caid_1)$. We let $\sigma_2(cacheMemory) = M_1$ and $\sigma_2(cacheMemory) = M_2$ for fresh logical variables M_1 and M_2. The condition removed==True allows us to statically infer from the code of the remove_inv method (Section 5, Figure 11) that the initial status of the given address in the next-level cache is either Nothing or Just(In), which identifies the path leading to the following symbolic configuration by the symbolic transition rules for **if**-statements and synchronous calls:

$$\begin{gathered}\langle caid_1, \sigma_1, (\tau[removed \mapsto \mathsf{True}], S'), Q_1{+}\mathsf{fetchBl}(n)\rangle,\\ \langle caid_2, \sigma_2[cacheMemory \mapsto M_2[n \mapsto \bot]], idle, Q_2{+}\mathsf{fetch}(n)\rangle\\ \mid next(caid_1) = caid_2 \wedge status(M_2, n) \in \{inv, \bot\}\end{gathered}$$

```
1  Unit fetch(Address n){
2    switch (nextLevel) {
3      Just(nextCache) => {
4        removed = nextCache.remove_inv(n); // removed==True: LC-Miss
5        if (removed) {
6          nextCache!fetch(n);
7          this!fetchBl(n);
8        }
9        else {
10         Pair<Address,Status> selected = cacheline(cacheMemory, n);
11         Maybe<Status> s = nextCache.swap(n,selected);
12         // s!=Nothing & fst(selected)==n: LC-Hit2;
13         // s!=Nothing & fst(selected)!=n: LC-Hit1
14         if (s != Nothing) {
15           if (fst(selected)!=n) { cacheMemory = removeKey(cacheMemory,fst(selected)); }
16           cacheMemory = put(cacheMemory, n, fromJust(s));
17         }
18         else this!fetch(n);
19       }
20     }
21     _ => {
22       this.broadcast(n); // Synch
23       this!fetchBl(n);
24     }}
25 }
26
27 Unit broadcast(Address n) {
28   await bus!lock();
29   bus.sendRd(this, n);
30   bus.release();
31 }
```

Fig. 14. The annotated fetch method.

where S' is the remaining body of the fetch method. Note that the local variable *removed* is updated in τ, while σ_2 is updated by setting the status of the address n to $\bot$ in M as a result of the synchronous call to method remove_inv. Applying the abstraction function α to both configurations, we get

$$\begin{aligned}
&\alpha(\langle caid_1, \sigma_1, (\tau, S), Q_1\rangle) = (caid_1 \bullet M_1 \bullet Q_1 + \textbf{fetch}(n))\\
&\alpha(\langle caid_2, \sigma_2, idle, Q_2\rangle) = (caid_2 \bullet M_2 \bullet Q_2)\\
&\alpha(\langle caid_1, \sigma_1, (\tau[removed \mapsto \mathsf{True}], S'), Q_1{+}\mathsf{fetchBl}(n)\rangle\\
&\qquad = (caid_1 \bullet M_1 \bullet Q_1 + \textbf{fetchBl}(n))\\
&\alpha(\langle caid_2, \sigma_2[cacheMemory \mapsto M_2[n \mapsto \bot], idle, Q_2{+}\mathsf{fetch}(n)\rangle)\\
&\qquad = (caid_2 \bullet M_2[n \mapsto \bot] \bullet Q_2 + \textbf{fetch}(n))
\end{aligned}$$

Thus this transition, modulo renaming of the logical variables, corresponds to the rule LC-Miss (the above path condition corresponds with the premises of the rule).

Case: Figure 14, Lines 4–11, silent step. Now consider the path leading from the execution of the remove_inv method via the **else**-branch of the subsequent **if**-statement to the evaluation of the select function and then to the (synchronous) call of the swap method (Line 11). As in the previous

```
Maybe<Status> swap(Address n_out, Pair<Address,Status> n_in) {
  Maybe<Status> tmp = Nothing;
  switch (lookup(cacheMemory,n_out)) {
    Nothing => skip;
    Just(In) => skip;
    _ => {
      tmp = lookup(cacheMemory,n_out);
      cacheMemory = removeKey(cacheMemory,n_out);
      if (fst(n_in)!=n_out) {
        cacheMemory = put(cacheMemory, fst(n_in), snd(n_in));
      }
    }}
  return tmp;
}
```

Fig. 15. The swap method.

```
Unit receiveRd(Address n,IBarrier start,IBarrier end) {
  // lookup(cacheMemory,n))==Just(Mo): FLUSH-ONE-LINE;
  // lookup(cacheMemory,n))!=Just(Mo): IGNORE-FLUSH-ONE-LINE
  await start!synchronize();
  switch (lookup(cacheMemory,n)) {
    Just(Mo) => this!flush(n);
    _ => skip;
  }
  end!synchronize();
}
```

Fig. 16. The annotated receiveRd method.

case, we consider the initial symbolic configuration

$$\langle caid_1, \sigma_1, (\tau, S), Q_1\rangle,\ \langle caid_2, \sigma_2, idle, Q_2\rangle \ \mid \mathbf{true}$$

where the process (τ, S) is about to synchronously call the remove_inv method. As in the previous case, we assume that $\tau(nextCache) = next(caid_1)$. We let $\sigma_1(cacheMemory) = M_1$ and $\sigma_2(cacheMemory) = M_2$ for fresh logical variables M_1 and M_2. The condition removed==False allows us to statically deduce from the code of remove_inv that the initial status of the given address in the next-level cache is neither Nothing nor Just(In), i.e., it is either Just(Sh) or Just(Mo), which identifies the path leading to the following symbolic configuration by the symbolic transition rules for **if**-statements and synchronous calls:

$$\langle caid_1, \sigma_1, (\tau', S'), Q_1\rangle,\ \langle caid_2, \sigma_2, idle, Q_2\rangle \ \mid next(caid_1) = caid_2 \wedge status(M_2, n) \in \{sh, mo\}$$

where $\tau' = \tau[removed \mapsto \text{False}, selected \mapsto cacheline(M_1, n)]$, i.e., the local variable *selected* is updated with the logical expression $cacheline(M_1, n)$, which we define by $cacheline(M, n) = (select(M, n), status(M, select(M, n)))$, and S' denotes the remaining method body. Applying the abstraction function α, we obtain $\alpha(\langle caid_1, \sigma_1, (\tau', S'), Q_1\rangle) = caid_1 \bullet M_1 \bullet Q_1 + \text{fetch}(n)$. Consequently, this symbolic transition corresponds to a silent step in the TS model.

Case: Figure 14, Lines 11–26, LC-Hit$_1$. The annotation s!=Nothing ∧ fst(selected)!=n : LC-Hit$_1$ is associated to the synchronous call to swap on Line 11. Consider the initial symbolic configuration

$$\langle caid_1, \sigma_1, (\tau, S), Q_1\rangle,\ \langle caid_2, \sigma_2, idle, Q_2\rangle$$

assuming that $\tau(nextCache) = next(caid_1))$, $\sigma_1(cacheMemory) = M_1$, and $\sigma_2(cacheMemory) = M_2$ for fresh logical variables M_1 and M_2, and that the process (τ, S) is about to synchronously call the swap method. The conditions s!=Nothing and fst(selected)!=n allow us to statically deduce the path taken by the symbolic execution of the swap method in the symbolic configuration of $caid_2$ (see Figure 15). The symbolic transition rules for assignment, **switch**- and **if**-statements then result in the update of M_2 to $M_2[select(M_1, n) \mapsto status(M_1, select(M_1, n)), n \mapsto \bot]$ by the call of the swap method with the path conditions $n!{=}select(M_1, n)$ and $status(M_2, n) \notin \{\bot, inv\}$; the latter is equivalent to $status(M_2, n) \in \{sh, mo\}$. The symbolic execution of the method swap returns the logical expression $status(M_2, n)$ which is assigned to the local variable s, and ultimately bound to n in M_1. Thus, the symbolic execution of fetch then updates M_1 to $M_1[select(M_1, n) \mapsto \bot, n \mapsto status(M_2, n)]$, and we obtain the symbolic configuration

$$\begin{aligned}&\langle caid_1, \sigma_1[cacheMemory \mapsto M_1[select(M_1, n) \mapsto \bot, n \mapsto status(M_2, n)]], idle, Q_1\rangle,\\ &\langle caid_2, \sigma_2[cacheMemory \mapsto M_2[select(M_1, n) \mapsto status(M_1, select(M_1, n)), n \mapsto \bot]], idle, Q_2\rangle\\ &\qquad \mid next(caid_1) = caid_2 \land status(M_2, n) \in \{sh, mo\} \land select(M_1, n) \neq n\end{aligned}$$

Applying the abstraction function α, we obtain

$$\begin{aligned}&\alpha(\langle caid_1, \sigma_1, (\tau, S), Q_1\rangle) = (caid_1 \bullet M_1 \bullet Q_1 + \textbf{fetch}(n))\\ &\alpha(\langle caid_2, \sigma_2, idle, Q_2\rangle) = (caid_2 \bullet M_2 \bullet Q_2)\\ &\alpha(\langle caid_1, \sigma_1[cacheMemory \mapsto M_1[select(M_1, n) \mapsto \bot, n \mapsto status(M_2, n)]], idle, Q_1\rangle)\\ &\qquad = (caid_1 \bullet M_1[select(M_1, n) \mapsto \bot, n \mapsto status(M_2, n)] \bullet Q_1)\\ &\alpha(\langle caid_2, \sigma_2[cacheMemory \mapsto M_2[select(M_1, n) \mapsto, n \mapsto \bot]], idle, Q_2\rangle)\\ &\qquad = (caid_2 \bullet M_2[select(M_1, n) \mapsto status(M_1, select(M_1, n)), n \mapsto \bot] \bullet Q_2)\end{aligned}$$

It follows that the symbolic transition (and its path condition) corresponds to LC-Hit$_1$.

Case: Figure 14, Lines 11–26, LC-Hit$_2$. The annotation s!=Nothing ∧ fst(selected)==n : LC-Hit$_2$ is associated to the synchronous call to swap on Line 11. In contrast to the previous case, the **if**-statement on Line 15 is not executed because of the condition fst(selected)==n. We consider the initial symbolic configuration

$$\langle caid_1, \sigma_1, (\tau, S), Q_1\rangle,\ \langle caid_2, \sigma_2, idle, Q_2\rangle \mid \textbf{true}$$

assuming that $\tau(nextCache) = next(caid_1)$, $\sigma_1(cacheMemory) = M_1$, and $\sigma_2(cacheMemory) = M_2$ for fresh logical variables M_1 and M_2, and that the process (τ, S) is about to synchronously call the swap method. The condition s!=Nothing ∧ fst(selected)==n allows us to statically deduce the path taken by the symbolic execution of the swap method in $caid_2$ (see Figure 15). In particular, the annotation fst(selected)==n corresponds to the path conditions $select(M_1, n) = n$. When executing swap, the symbolic transition rules for assignment, **switch**- and **if**-statements then result in the update of M_2 to $M_2[n \mapsto \bot]$. The execution of the assignment (Line 16 of fetch) results in the update of M_1 to $M_1[n \mapsto status(M_2, n)]$, and we obtain the symbolic configuration

$$\begin{aligned}&\langle caid_1, \sigma_1[cacheMemory \mapsto M_1[n \mapsto status(M_2, n)]], idle, Q_1\rangle,\\ &\langle caid_2, \sigma_2[cacheMemory \mapsto M_2[n \mapsto \bot]], idle, Q_2\rangle\\ &\qquad \mid next(caid_1) = caid_2 \land status(M_2, n) \in \{sh, mo\} \land select(M_1, n) = n\end{aligned}$$

Applying the abstraction function α, we obtain

$$\begin{aligned}
&\alpha(\langle caid_1, \sigma_1, (\tau, S), Q_1\rangle) = (caid_1 \bullet M_1 \bullet Q_1 + \mathbf{fetch}(n))\\
&\alpha(\langle caid_2, \sigma_2, idle, Q_2\rangle) = (caid_2 \bullet M_2 \bullet Q_2)\\
&\alpha(\langle caid_1, \sigma_1[cacheMemory \mapsto M_1[n \mapsto status(M_2, n)]], idle, Q_1\rangle)\\
&\qquad = (caid_1 \bullet M_1[n \mapsto status(M_2, n)] \bullet Q_1)\\
&\alpha(\langle caid_2, \sigma_2[cacheMemory \mapsto M_2[n \mapsto \bot]], idle, Q_2\rangle) = (caid_2 \bullet M_2[n \mapsto \bot] \bullet Q_2)
\end{aligned}$$

It follows that the symbolic transition (and its path condition) corresponds to rule LC-Hit$_2$.

Case: Figure 14, Lines 11–26, silent step. This case corresponds to the execution path which leads from the return of the swap method via the **else**-branch of the **if**-statement (so s==Nothing) to the end of the fetch method. We consider the initial symbolic configuration

$$\langle caid_1, \sigma_1, (\tau, S), Q_1\rangle,\ \langle caid_2, \sigma_2, idle, Q_2\rangle \mid \mathbf{true}$$

assuming that $\tau(nextCache) = next(caid_1)$, $\sigma_1(cacheMemory) = M_1$, and $\sigma_2(cacheMemory) = M_2$ for fresh logical variables M_1 and M_2, and that the process (τ, S) is about to synchronously call the swap method. We infer statically from the code of the swap method in Figure 15 and the condition of the **else**-branch on Line 18 that the given address n of the fetch method, in the next-level cache is either Nothing or Just(In). Thus, this execution path leads to the following symbolic configuration:

$$\langle caid_1, \sigma_1, idle, Q_1{+}\mathsf{fetch}(n)\rangle,\ \langle caid_2, \sigma_2, idle, Q_2\rangle \mid next(caid_1) = caid_2 \wedge status(M_2, n) \in \{inv, \bot\}$$

Applying the abstraction function α, we obtain

$$\begin{aligned}
&\alpha(\langle caid_1, \sigma_1, (\tau, S), Q_1\rangle) = (caid_1 \bullet M_1 \bullet Q_1 + \mathbf{fetch}(n))\\
&\alpha(\langle caid_2, \sigma_2, idle, Q_2\rangle) = (caid_2 \bullet M_2 \bullet Q_2)\\
&\alpha(\langle caid_1, \sigma_1, idle, Q_1{+}\mathsf{fetch}(n)\rangle) = (caid_1 \bullet M_1 \bullet Q_1 + \mathbf{fetch}(n))
\end{aligned}$$

Clearly, this symbolic transition corresponds to a silent step in the TS model.

Case: Figure 14, Lines 21–23, Synch. This path corresponds to the execution of the synchonous self-call of the broadcast method followed by the asynchonous self-call of the fetchBl method. Abstracting from implementation of the broadcast mechanism this method reduces to a **skip**-statement. Thus the symbolic execution of this path is described by the following transition:

$$\langle caid, \sigma, (\tau, S), Q\rangle \mid next(caid) = \bot \quad\Rightarrow\quad \langle caid, \sigma, (\tau, S), Q{+}\mathsf{fetchBl}(n)\rangle \mid next(caid) = \bot$$

where $\sigma(cacheMemory) = M$, for some logical variable M and $\tau(nextCache) = next(caid)$ (note that the path condition $next(caid) = \bot$ follows from the execution path leading to the call of the broadcast method, which consists of the execution of the **switch**-statement and as such corresponds to a silent step in the TS model). Applying the abstraction function α we obtain the transition rule LLC-Miss

$$(caid \bullet M \bullet Q + \mathbf{fetch}(n)) \longrightarrow (caid \bullet M \bullet Q + \mathbf{fetchBl}(n))$$

of the TS model. It remains to check that applying the abstraction function α to the symbolic execution of the receiveRd method corresponds to the rules Flush-One-Line and Ignore-Flush-One-Line of the TS model. Let $\langle caid, \sigma, (\tau, S), Q\rangle \mid \mathbf{true}$ be a symbolic configuration, where S is the (abstracted) body of receiveRd. Clearly, symbolic execution of S, i.e., its **switch**-statement, leads to either

$$\langle caid, \sigma, idle, Q + \mathsf{flush}(n)\rangle \mid status(M, n) = mo$$

or

$$\langle caid, \sigma, idle, Q\rangle \mid status(M, n) \neq mo$$

Applying the abstraction function α to the first case gives the Flush-One-Line rule, and in the second case the Ignore-Flush-One-Line rule.

```
1  Unit fetchBl(Address n) {
2    switch (nextLevel) {
3      // nextLevel==Nothing &
4      // select(cacheMemory, n) !=n &
5      // cacheline(cacheMemory, n)==Pair(_,Mo): FetchBl3;
6      Just(nextCache) => {
7        Maybe<Status> status = nextCache.getStatus(n);
8        // status!=Nothing: LC-Fetch-Unblock
9        if (status == Nothing){ this!fetchBl(n); }
10       else { this!fetch(n); }
11     }
12     _ => {
13       Pair<Address,Status> selected = cacheline(cacheMemory, n);
14       if (fst(selected)==n) {
15           Status status = mainMemory.getStatus(n);
16           // select(cacheMemory, n) ==n : FetchBl1
17           cacheMemory = put(cacheMemory,n,status);
18       }
19       else {
20           switch (selected) {
21               Pair(selected_n,Mo) => {
22                   this!flush(selected_n);
23                   this!fetchW(n,selected_n); }
24               Pair(selected_n,_) => {
25                   Status status = mainMemory.getStatus(n);
26                   // select(cacheMemory, n) !=n & select(cacheMemory, n)!=Pair(_,Mo) : FetchBL2
27                   cacheMemory = removeKey(cacheMemory,selected_n);
28                   cacheMemory = put(cacheMemory,n,status);
29               }}
30       }}}
31 }
```

Fig. 17. The annotated fetchBl method of class Cache.

Method fetchBl, *Figure 17.*

CASE: FIGURE 17, LINES 1–7, SILENT STEP. Consider the path starting from initial stable point of the fetchBl method and leading to the synchronous call to the getStatus method of another Cache object on Line 7. We consider the initial symbolic configuration

$$\langle caid, \sigma, (\tau, S), Q\rangle \mid \mathbf{true}$$

Here, Q is the symbolic representation of the process queue and the process (τ, S) is the method activation of fetchBl, where S denotes the body of the fetch method. Assume that $\sigma(nextLevel) = next(caid)$ and $\sigma(cacheMemory) = M$ for some fresh logical variable M. The symbolic transition rule for the **switch**-statement results in the symbolic configuration

$$\langle caid, \sigma, (\tau[nextCache \mapsto next(caid)], S'), Q\rangle \mid next(caid) \neq \bot$$

where S' denotes the remaining statements to be executed. Applying the abstraction function α, we obtain

$$\alpha(\langle caid, \sigma, (\tau, S), Q\rangle) = caid \bullet M \bullet Q + \mathbf{fetchBl}(n)$$
$$\alpha(\langle caid, \sigma, (\tau', S'), Q\rangle) = caid \bullet M \bullet Q + \mathbf{fetchBl}(n)$$

Clearly, the symbolic transition corresponds to a silent step in the TS model.

CASE: FIGURE 17, LINES 7–9, SILENT STEP. Consider the execution path from the synchronous call to the getStatus method to the asynchronous self-call to the fetchBl method (Line 9) in the **then**-branch of the **if**-statement. Observe that the local variables of the active process correspond to those of the final state of the previous case; we may assume that $\tau(\mathit{nextCache}) = \mathit{next}(\mathit{caid}_1)$ and consider the initial symbolic configuration

$$\langle \mathit{caid}_1, \sigma_1, (\tau, S), Q_1\rangle,\ \langle \mathit{caid}_2, \sigma_2, \mathit{idle}, Q_2\rangle\ \mid \mathbf{true}$$

where additionally $\sigma_1(\mathit{cacheMemory}) = M_1$ and $\sigma_2(\mathit{cacheMemory}) = M_2$ for fresh logical variables M_1 and M_2, and the process (τ, S) is about to synchronously call getStatus with the symbolic value n. The condition status==Nothing allows us to statically determine that $\mathit{status}(M_2, n) = \bot$ from the symbolic execution of the synchronous call to getStatus, and identify the resulting configuration

$$\begin{array}{l}\langle \mathit{caid}_1, \sigma_1, (\tau[\mathit{status} \mapsto \bot], S'), Q_1 + \mathsf{fetchBl}(n)\rangle,\\ \langle \mathit{caid}_2, \sigma_2, \mathit{idle}, Q_2\rangle \mid \mathit{next}(\mathit{caid}_1) = \mathit{caid}_2 \land \mathit{status}(M_2, n) = \bot\end{array}$$

where S' denotes the remaining statements to be executed. Applying the abstraction function α, we obtain

$$\begin{array}{l}\alpha(\langle \mathit{caid}_1, \sigma_1, (\tau, S), Q_1\rangle) = \mathit{caid}_1 \bullet M_1 \bullet Q_1 + \mathbf{fetchBl}(n)\\ \alpha(\langle \mathit{caid}_2, \sigma_2, \mathit{idle}, Q_2\rangle) = \mathit{caid}_2 \bullet M_2 \bullet Q_2\\ \alpha(\langle \mathit{caid}_1, \sigma_1, (\tau[\mathit{status} \mapsto \bot], S'), Q_1 + \mathsf{fetchBl}(n)\rangle) = \mathit{caid}_1 \bullet M_1 \bullet Q_1 + \mathbf{fetchBl}(n)\end{array}$$

Thus, we obtain a *silent* step in the TS model.

CASE: FIGURE 17, LINES 7–10, LC-FETCH-UNBLOCK. The annotation status!=Nothing. corresponds to the execution path from the getStatus method to the asynchronous self-call to the fetch method (Line 10) in the **else**-branch of the **if**-statement. Again, we may assume that $\tau(\mathit{nextCache}) = \mathit{next}(\mathit{caid}_1)$ and consider the initial symbolic configuration

$$\langle \mathit{caid}_1, \sigma_1, (\tau, S), Q_1\rangle,\ \langle \mathit{caid}_2, \sigma_2, \mathit{idle}, Q_2\rangle\ \mid \mathbf{true}$$

where additionally $\sigma_1(\mathit{cacheMemory}) = M_1$ and $\sigma_2(\mathit{cacheMemory}) = M_2$ for fresh logical variables M_1 and M_2, and the process (τ, S) is about to synchronously call getStatus with the symbolic value n. The condition status!=Nothing allows us to statically determine that $\mathit{status}(M_2, n) \neq \bot$ from the symbolic execution of the synchronous call to getStatus, and identify the resulting configuration

$$\begin{array}{l}\langle \mathit{caid}_1, \sigma_1, (\tau[\mathit{status} \mapsto s], S'), Q_1 + \mathsf{fetch}(n)\rangle,\\ \langle \mathit{caid}_2, \sigma_2, p_2, Q_2\rangle \mid \mathit{next}(\mathit{caid}_1) = \mathit{caid}_2 \land \mathit{status}(M_2, n) \neq \bot\end{array}$$

where S' denotes the remaining statements to be executed and s the symbolic return value from the getStatus method. Applying the abstraction function α, we obtain

$$\begin{array}{l}\alpha(\langle \mathit{caid}_1, \sigma_1, (\tau, S), Q_1\rangle) = \mathit{caid}_1 \bullet M_1 \bullet Q_1 + \mathbf{fetchBl}(n)\\ \alpha(\langle \mathit{caid}_2, \sigma_2, \mathit{idle}, Q_2\rangle) = \mathit{caid}_2 \bullet M_2 \bullet Q_2\\ \alpha(\langle \mathit{caid}_1, \sigma_1, (\tau[\mathit{status} \mapsto s], S'_1), Q_1 + \mathsf{fetch}(n)\rangle) = \mathit{caid}_1 \bullet M_1 \bullet Q_1 + \mathbf{fetch}(n)\end{array}$$

Consequently, we obtain rule LC-FETCH-UNBLOCK in Figure 22.

CASE: FIGURE 17, LINES 1–15, SILENT STEP. From the **switch**-statement on Line 2 we know that *caid* is the last-level cache, so this path is associated to the annotation nextLevel==Nothing. It starts from the initial stable point of fetchBl and leads via the execution of select (Line 13) to the synchronous call to getStatus on an instance of Memory (Line 15). Assume (as in all the other above cases) that $\sigma(\mathit{nextLevel}) = \mathit{next}(\mathit{caid})$ and $\sigma(\mathit{cacheMemory}) = M$ for a fresh logical variable M. We consider an initial symbolic configuration

$$\langle \mathit{caid}, \sigma, (\tau, S), Q\rangle \mid \mathbf{true}$$

where the process (τ, S) is the activation of fetchBl(n) for a symbolic value n. The symbolic execution results in the following configuration:

$$\langle caid, \sigma, (\tau[selected \mapsto cacheline(M,n)], S'), Q\rangle \mid next(caid) = \bot \wedge select(M,n) = n$$

where S' denotes the remaining statements to be executed. Applying the abstraction function α, we obtain

$$\alpha(\langle caid, \sigma, (\tau, S), Q\rangle) = caid \bullet M \bullet Q + \textbf{fetchBl}(n)$$
$$\alpha(\langle caid, \sigma, (\tau[selected \mapsto cacheline(M,n)], S'), Q\rangle) = caid \bullet M \bullet Q + \textbf{fetchBl}(n)$$

Thus, the symbolic execution corresponds to a *silent* step in the TS model.

Case: Figure 17, Lines 15–31, FetchBl$_1$. Now consider the path starting from the synchronous call to the getStatus method on Line 15 to the termination of the fetchBl method, with the annotation select(cacheMemory,n) ==n. We know from the previous case that *caid* is the last-level cache and consider the initial symbolic configuration

$$\langle caid, \sigma_1, (\tau, S), Q_1\rangle,\ \langle main, \sigma_2, idle, Q_2\rangle\ \mid next(caid) = \bot$$

where $\sigma_1(cacheMemory) = M_1$ and $\sigma_2(mainMemory) = M_2$ for fresh logical variables M_1 and M_2, such that the process (τ, S) is about make the synchronous call to the getStatus method. The annotation select(cacheMemory,n)==n allows us to infer the following symbolic configuration:

$$\langle caid, \sigma_1[cacheMemory \mapsto M_1[n \mapsto status(select(M_2,n))]], idle, Q_1\rangle,\ \langle main, \sigma_2, idle, Q_2\rangle$$
$$\mid next(caid) = \bot \wedge select(M_1,n) = n$$

Applying the abstraction function α, we obtain

$$\alpha(\langle caid, \sigma_1, (\tau, S), Q_1\rangle) = caid \bullet M_1 \bullet Q_1 + \textbf{fetchBl}(n)$$
$$\alpha(\langle main, \sigma_2, idle, Q_2\rangle) = M_2$$
$$\alpha(\langle caid, \sigma_1[cacheMemory \mapsto M_1[n \mapsto status(select(M_2,n))]], idle, Q_1\rangle) =$$
$$caid \bullet M_1[n \mapsto status(select(M_2,n))] \bullet Q_1$$

We obtain rule FetchBl$_1$ in Figure 22, where the premisses are given by the path condition of the symbolic execution.

Case: Figure 17, Lines 1–31, FetchBl$_3$. We consider the path starting from the initial stable point of the fetchBl method, with the annotations nextLevel==Nothing, select(cacheMemory,n)!=n and select(cacheMemory,n)==Pair(_,Mo). We consider the initial symbolic configuration

$$\langle caid, \sigma_1, (\tau, S), Q_1\rangle,\ \langle main, \sigma_2, idle, Q_2\rangle \mid \textbf{true}$$

where $\sigma_1(cacheMemory) = M_1$ and $\sigma_2(mainMemory) = M_2$ for fresh logical variables M_1 and M_2, and the process (τ, S) is the instance of the fetch(n) method. This path goes via the **else**-branch (Line 19) and the first case of the **switch**-statement (Line 21). As before, *caid* is the last-level cache, and we derive (Line 12) that $next(caid) = \bot$. The annotations nextLevel==Nothing, select(cacheMemory,n)!=n and select(cacheMemory, n)==Pair(_,Mo) allow us to derive the following configuration

$$\langle caid, \sigma_1, idle, Q_1 + \text{flush}(m) + \text{fetchW}(n,m)\rangle,$$
$$\langle main, \sigma_2, idle, Q_2\rangle \mid next(caid) = \bot \wedge select(M_1,n) \neq n \wedge status(M_1, select(M_1,n)) = mo$$

Applying the abstraction function α, we obtain

$$\alpha(\langle caid, \sigma_1, (\tau, S), Q_1\rangle) = caid \bullet M_1 \bullet Q_1 + \textbf{fetchBl}(n)$$
$$\alpha(\langle main, \sigma_2, idle, Q_2\rangle) = M_2$$
$$\alpha(\langle caid, \sigma_1, idle, Q_1 + \text{flush}(m) + \text{fetchW}(n,m)\rangle) = caid \bullet M_1 \bullet Q_1 + \textbf{flush}(m) + \textbf{fetchW}(n,m)$$

Consequently, we obtain rule FetchBl$_3$.

Case: Figure 17, Lines 1–25, silent step. Now consider the path starting from the initial stable point of the fetchBl method, which leads via the execution of the select function on Line 13 to the synchronous call to the getStatus method of the main memory on Line 25. We consider the initial symbolic configuration

$$\langle caid, \sigma_1, (\tau, S), Q_1\rangle,\ \langle main, \sigma_2, idle, Q_2\rangle \mid \mathbf{true}$$

where $\sigma_1(cacheMemory) = M_1$ and $\sigma_2(mainMemory) = M_2$ for fresh logical variables M_1 and M_2, and the process (τ, S) is the instance of the fetch(n) method. As before, *caid* is the last-level cache, and we derive (Line 12) that $next(caid) = \bot$. We obtain the following symbolic configuration:

$$\begin{array}{l}\langle caid, \sigma_1, (\tau[selected \mapsto cacheline(M_1, n)], S'), Q\rangle,\\ \langle main, \sigma_2, idle, Q_2\rangle \mid next(caid) = \bot\ \wedge select(M_1, n) \neq n \wedge status(M_1, select(M_1, n)) \neq mo\end{array}$$

where m and st are symbolic values and S' denotes the remaining statements to be executed. Applying the abstraction function α, we obtain

$$\begin{array}{l}\alpha(\langle caid, \sigma_1, (\tau, S), Q_1\rangle) = caid \bullet M_1 \bullet Q_1 + \mathbf{fetchBl}(n)\\ \alpha(\langle main, \sigma_2, idle, Q_2\rangle) = M_2\\ \alpha(\langle caid, \sigma_1, (\tau[selected \mapsto cacheline(M_1, n)], S'), Q\rangle) = caid \bullet M_1 \bullet Q_1 + \mathbf{fetchBl}(n)\end{array}$$

Thus, we here obtain a *silent* step in the TS model.

Case: Figure 17, Lines 25–31, FetchBl$_2$. Now consider the path starting from the synchronous call to getStatus on Line 25 to the termination of the fetchBl method. Note that the initial symbolic state of the instance cache *caid* corresponds to the symbolic state resulting from the symbolic execution of the path discussed for the previous case above, and we can use its path conditions. Thus, we consider the initial symbolic configuration

$$\begin{array}{l}\langle caid, \sigma_1, (\tau, S), Q\rangle,\\ \langle main, \sigma_2, idle, Q_2\rangle \mid next(caid) = \bot\ \wedge select(M_1, n) \neq n \wedge status(M_1, select(M_1, n)) \neq mo\end{array}$$

with symbolic variables as above, and where the process (τ, S) is the instance of the fetch(n) method. Applying symbolic execution, we obtain the following symbolic configuration:

$$\begin{array}{l}\langle caid, \sigma_1[cacheMemory \mapsto M_1[m \mapsto \bot, n \mapsto status(M_2, n)]], idle, Q_1\rangle,\\ \langle main, \sigma_2, p_2, Q_2\rangle \mid next(caid) = \bot\ \wedge select(M_1, n) \neq n \wedge status(M_1, select(M_1, n)) \neq mo\end{array}$$

Applying the abstraction function α, we obtain

$$\begin{array}{l}\alpha(\langle caid, \sigma_1, (\tau, S), Q\rangle) = caid \bullet M_1 \bullet Q_1 + \mathbf{fetchBl}(n)\\ \alpha(\langle main, \sigma_2, idle, Q_2\rangle) = M_2\\ \alpha(\langle caid, \sigma_1[cacheMemory \mapsto M_1[select(M_1, n) \mapsto \bot, n \mapsto status(M_2, n)]], idle, Q_1\rangle) =\\ caid \bullet M_1[select(M_1, n) \mapsto \bot, n \mapsto status(M_2, n)] \bullet Q_1\end{array}$$

Consequently we obtain rule FetchBl$_2$ in Figure 22, modulo renaming of logical variables.

Method fetchW*, Figure 18.*

Case: Figure 18, Lines 1–2, Silent Step. Consider the path starting from the initial stable point of a method call to fetchW(n,m) and leading to the **await**-statement on Line 2. We consider an initial symbolic configuration

$$\langle caid, \sigma, (\tau, S), Q\rangle \mid \mathbf{true}$$

where $\sigma(cacheMemory) = M$ for some fresh logical variable M, Q is a symbolic representation of the process queue and (τ, S) is the symbolic representation of the fetchW method. By symbolic execution, we obtain the symbolic configuration

$$\langle caid, \sigma, idle, Q{+}(\tau, S)\rangle \mid \mathbf{true}.$$

```
1  Unit fetchW(Address n,Address m){
2    await (lookupDefault(cacheMemory,m, In)!=Mo); // FetchW
3    this!fetchBl(n);
4  }
```

Fig. 18. The annotated fetchW method of class Cache.

```
1  Unit flush(Address n) {
2    // lookup(cacheMemory,n)!=Just(Mo): Flush2;
3    switch (lookup(cacheMemory,n)) {
4      Just(Mo) => {
5        mainMemory.setStatus(n,Sh); // Flush1
6        cacheMemory = put(cacheMemory,n,Sh);
7      }
8      _ => skip;
9  }}
```

Fig. 19. The annotated flush method of class Cache.

Applying the abstraction function α, we obtain

$$\begin{aligned}\alpha(\langle caid, \sigma, (\tau, S), Q\rangle) &= caid \bullet M \bullet Q + \mathbf{fetchW}(n, m)\\ \alpha(\langle caid, \sigma, idle, Q{+}(\tau, S)\rangle) &= caid \bullet M \bullet Q + \mathbf{fetchW}(n, m)\end{aligned}$$

Clearly, this symbolic execution step corresponds to a silent step in the TS model.

Case: Figure 18, Lines 2–4, FetchW. We consider a path in which the guard of the **await**-statement on Line 2 holds and the initial symbolic configuration

$$\langle caid, \sigma, (\tau, S), Q\rangle \mid \mathbf{true}$$

where $\sigma(cacheMemory) = M$ for some fresh logical variable M, Q is a symbolic representation of the process queue and (τ, S) is the symbolic representation of the fetchW method. Symbolic execution here gives us the path condition $\text{lookupDefault}(M, m, in) \neq mo$, which is equivalent to $status(M, m) \neq mo$. By symbolic execution, we then derive the following symbolic configuration

$$\langle caid, \sigma, idle, Q{+}\text{fetchBl}(n)\rangle \mid status(M, m) \neq mo$$

Applying the abstraction function α, we obtain

$$\begin{aligned}\alpha(\langle caid, \sigma, (\tau, S), Q\rangle) &= caid \bullet M \bullet Q + \mathbf{fetchW}(n, m)\\ \alpha(\langle caid, \sigma, idle, Q{+}\text{fetchBl}(n)\rangle) &= caid \bullet M \bullet Q + \mathbf{fetchBl}(n)\end{aligned}$$

Consequently, we obtain rule FetchW in Figure 23 modulo renaming of logical variables.

Method flush, Figure 19.

Case Figure 19, Lines 1–9, Flush$_1$. Consider a path starting with the initial stable point of the method call to flush with the assumption lookup(cacheMemory,n)==Mo. We consider a symbolic configuration

$$\langle caid, \sigma_1, (\tau, S), Q_1\rangle,\ \langle main, \sigma_2, idle, Q_2\rangle \mid \mathbf{true}$$

where $\sigma_1(mainMemory) = main$, $\sigma_1(cacheMemory) = M_1$ and $\sigma_2(mainMemory) = M_2$ for fresh logical variables $main$, M_1 and M_2, Q_1 and Q_2 are symbolic representations of process queues, and (τ, S) is the symbolic representation of the method activation of flush. Our assumption lookup(cacheMemory,n)==Mo gives us the path condition lookup$(M_1, n) = mo$, which is equivalent to $status(M_1, n) = mo$ We then obtain by symbolic execution the following symbolic configuration

$$\langle caid, \sigma_1', idle, Q_1\rangle,\ \langle main, \sigma_2', idle, Q_2\rangle \mid status(M_1, n) = mo$$

where $\sigma_1' = \sigma_1[cacheMemory \mapsto M_1[n \mapsto sh]]$ and $\sigma_2' = \sigma_2[mainMemory \mapsto M_2[n \mapsto sh]]$. Applying the abstraction function α, we obtain

$$\begin{array}{l}
\alpha(\langle caid, \sigma_1, (\tau, S), Q_1\rangle) = caid \bullet M_1 \bullet Q + \mathbf{flush}(n)\\
\alpha(\langle main, \sigma_2, idle, Q_2\rangle) = M_2\\
\alpha(\langle caid, \sigma_1', (\tau, S), Q_1\rangle) = caid \bullet M_1[n \mapsto sh] \bullet Q_1\\
\alpha(\langle main, \sigma_2', p_2, Q_2\rangle) = M_2[n \mapsto sh]
\end{array}$$

Consequently, we obtain the rule Flush$_1$ in Figure 24.

Case Figure 19, Lines 1–9, Flush$_2$. For this case, we assume that lookup(cacheMemory,n)!=Mo. We consider a symbolic configuration

$$\langle caid, \sigma, (\tau, S), Q\rangle \mid \mathbf{true}$$

where $\sigma(cacheMemory) = M$ for a fresh logical variable M, Q is a symbolic representation of a process queue and (τ, S) the symbolic representation of the method activation of flush. From symbolic execution with the assumption lookup(cacheMemory,n)!=Mo, we derive the symbolic configuration

$$\langle caid, \sigma, idle, Q\rangle \mid status(M, n) \neq mo.$$

Applying the abstraction function α, we obtain

$$\begin{array}{l}
\alpha(\langle caid, \sigma, (\tau, S), Q\rangle) = caid \bullet M \bullet Q + \mathbf{flush}(n)\\
\alpha(\langle caid, \sigma, idle, Q\rangle) = caid \bullet M \bullet Q
\end{array}$$

Consequently, we obtain the rule Flush$_2$ in Figure 24.

C THE MULTICORE TS MODEL

The multicore TS model is structured in terms of separate TS models for the cores, caches, and global synchronization. In general, we assume that the unlabelled transitions which describe the behavior of the individual cores and caches are applied in the context of a configuration *cf*, and we omit the straightforward context rule here. On the other hand, for the labelled transitions we introduce explicit synchronization rules for lifting them to a particular context. We here include the complete TS multicore model, organized as follows:

- The transition rules for *cores* cover the basic instructions **read**(r), **readBl**(r), **write**(r), and **writeBl**(r) in Figure 20.
- The rules for *caches* have been further structured in terms of separate TS models for the individual *dst* instructions (Figures 21, 22, 23, and 24)
- The transition rules for *global synchronization* are structured in terms of a TS model for labelled transitions (Figure 25) and a TS model of rules for matching these labelled transitions (Figure 26).

$$(\textsc{PrRd}_1)\quad \frac{next(cid) = caid \quad status(M,n) \in \{sh, mo\}}{\begin{array}{c}(cid \bullet \mathbf{read}(n); rst),\ (caid \bullet M \bullet dst) \longrightarrow \\ (cid \bullet rst),\ (caid \bullet M \bullet dst)\end{array}}$$

$$(\textsc{PrRd}_2)\quad \frac{next(cid) = caid \quad status(M,n) \in \{inv, \bot\}}{\begin{array}{c}(cid \bullet \mathbf{read}(n); rst),\ (caid \bullet M \bullet dst) \longrightarrow \\ (cid \bullet \mathbf{readBl}(n); rst),\ (caid \bullet M[n \mapsto \bot] \bullet dst + \mathbf{fetch}(n))\end{array}}$$

$$(\textsc{PrRd}_3)\quad \frac{next(cid) = caid \quad n \in dom(M)}{\begin{array}{c}(cid \bullet \mathbf{readBl}(n); rst),\ (caid \bullet M \bullet dst) \longrightarrow \\ (cid \bullet \mathbf{read}(n); rst),\ (caid \bullet M \bullet dst)\end{array}}$$

$$(\textsc{PrWr}_1)\quad \frac{next(cid) = caid \quad status(M,n) = mo}{\begin{array}{c}(cid \bullet \mathbf{write}(n); rst),\ (caid \bullet M \bullet dst) \longrightarrow \\ (cid \bullet rst),\ (caid \bullet M \bullet dst)\end{array}}$$

$$(\textsc{PrWr}_2)\quad \frac{first(caid) = true \quad cid(caid) = c \quad status(M',n) = sh}{\begin{array}{c}(cid \bullet \mathbf{write}(n); rst),\ (caid \bullet M' \bullet dst),\ M \xrightarrow{!RdX(n)} \\ (cid \bullet rst),\ (caid \bullet M'[n \mapsto mo] \bullet dst),\ M[n \mapsto inv]\end{array}}$$

$$(\textsc{PrWr}_4)\quad \frac{next(cid) = caid \quad n \in dom(M)}{\begin{array}{c}(cid \bullet \mathbf{writeBl}(n); rst),\ (caid \bullet M \bullet dst) \longrightarrow \\ (cid \bullet \mathbf{write}(n); rst),\ (caid \bullet M \bullet dst)\end{array}}$$

$$(\textsc{PrWr}_3)\quad \frac{next(cid) = caid \quad status(M,n) \in \{inv, \bot\}}{\begin{array}{c}(cid \bullet \mathbf{write}(n); rst),\ (caid \bullet M \bullet dst) \longrightarrow \\ (cid \bullet \mathbf{writeBl}(n); rst),\ (caid \bullet M[n \mapsto \bot] \bullet dst + \mathbf{fetch}(n))\end{array}}$$

Fig. 20. Transition rules for **read**(r), **readBl**(r), **write**(r), and **writeBl**(r).

$$\textsc{(LC-Hit}_1\textsc{)}\quad \frac{\begin{array}{c} next(caid) = caid' \quad select(M,n) = m \\ n \neq m \quad s = status(M,m) \quad s' = status(M',n) \quad s' \in \{sh, mo\} \end{array}}{\begin{array}{c} (caid \bullet M \bullet dst + \mathbf{fetch}(n)),\ (caid' \bullet M' \bullet dst') \rightarrow \\ (caid \bullet M[m \mapsto \bot, n \mapsto s'] \bullet dst),\ (caid' \bullet M'[n \mapsto \bot, m \mapsto s] \bullet dst') \end{array}}$$

$$\textsc{(LC-Hit}_2\textsc{)}\quad \frac{\begin{array}{c} next(caid) = caid' \quad select(M,n) = n \\ s' = status(M',n) \quad s' \in \{sh, mo\} \end{array}}{\begin{array}{c} (caid \bullet M \bullet dst + \mathbf{fetch}(n)),\ (caid' \bullet M' \bullet dst') \rightarrow \\ (caid \bullet M[n \mapsto s'] \bullet dst),\ (caid' \bullet M'[n \mapsto \bot] \bullet dst') \end{array}}$$

$$\textsc{(LC-Miss)}\quad \frac{next(caid) = caid' \quad status(M',n) \in \{inv, \bot\}}{\begin{array}{c} (caid \bullet M \bullet dst + \mathbf{fetch}(n)),\ (caid' \bullet M' \bullet dst') \rightarrow \\ (caid \bullet M \bullet dst + \mathbf{fetchBl}(n)),\ (caid' \bullet M'[n \mapsto \bot] \bullet dst' + \mathbf{fetch}(n)) \end{array}}$$

$$\textsc{(LLC-Miss)}\quad \frac{next(caid) = \bot}{(caid \bullet M \bullet dst + \mathbf{fetch}(n)) \xrightarrow{!Rd(n)} (caid \bullet M \bullet dst + \mathbf{fetchBl}(n))}$$

Fig. 21. Transition rules for $\mathbf{fetch}(n)$.

$$\textsc{(FetchBl}_1\textsc{)}\quad \frac{\begin{array}{c} next(caid) = \bot \\ select(M,n) = n \quad s = status(\overline{M},n) \end{array}}{\begin{array}{c} (caid \bullet M \bullet dst + \mathbf{fetchBl}(n)),\ \overline{M} \rightarrow \\ (caid \bullet M[n \mapsto s] \bullet dst),\ \overline{M} \end{array}}$$

$$\textsc{(FetchBl}_3\textsc{)}\quad \frac{\begin{array}{c} next(caid) = \bot \\ select(M,n) = n' \quad n' \neq n \quad status(M,n') = mo \end{array}}{\begin{array}{c} (caid \bullet M \bullet dst + \mathbf{fetchBl}(n)) \rightarrow \\ (caid \bullet M \bullet dst + \mathbf{flush}(n') + \mathbf{fetchW}(n,n')) \end{array}}$$

$$\textsc{(FetchBl}_2\textsc{)}\quad \frac{\begin{array}{c} next(caid) = \bot \quad select(M,n) = n' \quad n' \neq n \\ status(M,n') \neq mo \quad s = status(\overline{M},n) \end{array}}{\begin{array}{c} (caid \bullet M \bullet dst + \mathbf{fetchBl}(n)),\ \overline{M} \rightarrow \\ (caid \bullet M[n' \mapsto \bot, n \mapsto s] \bullet dst),\ \overline{M} \end{array}}$$

$$\textsc{(LC-Fetch-Unblock)}\quad \frac{next(caid) = caid' \quad n \in dom(M')}{\begin{array}{c} (caid \bullet M \bullet dst + \mathbf{fetchBl}(n)),\ (caid' \bullet M' \bullet dst') \rightarrow \\ (caid \bullet M \bullet dst + \mathbf{fetch}(n)),\ (caid' \bullet M' \bullet dst') \end{array}}$$

Fig. 22. Transition rules for $\mathbf{fetchBl}(n)$.

$$\textsc{(FetchW)}\quad \frac{status(M,n') \neq mo}{(caid \bullet M \bullet dst + \mathbf{fetchW}(n,n')) \rightarrow (caid \bullet M \bullet dst + \mathbf{fetchBl}(n))}$$

Fig. 23. Transition rule for $\mathbf{fetchW}(n,n')$.

$$\frac{\textsc{(Flush}_1\textsc{)}\quad status(M,n) = mo}{(caid \bullet M \bullet dst + \mathbf{flush}(n)),\ \overline{M} \longrightarrow (caid \bullet M[n \mapsto sh] \bullet dst),\ \overline{M}[n \mapsto sh]}
\qquad
\frac{\textsc{(Flush}_2\textsc{)}\quad status(M,n) \neq mo}{(caid \bullet M \bullet dst + \mathbf{flush}(n)) \longrightarrow (caid \bullet M \bullet dst)}$$

Fig. 24. Transition rules for **flush**(n).

$$\frac{\textsc{(Invalidate-One-Line)}\quad status(M,n) = sh}{caid \bullet M \bullet dst \xrightarrow{?RdX(n)} caid \bullet M[n \mapsto inv] \bullet dst}
\qquad
\frac{\textsc{(Ignore-Invalidate-One-Line)}\quad status(M,n) \in \{inv, \bot\}}{caid \bullet M \bullet dst \xrightarrow{?RdX(n)} caid \bullet M \bullet dst}$$

$$\frac{\textsc{(Flush-One-Line)}\quad status(M,n) = mo}{caid \bullet M \bullet dst \xrightarrow{?Rd(n)} caid \bullet M \bullet dst + \mathbf{flush}(n)}
\qquad
\frac{\textsc{(Ignore-Flush-One-Line)}\quad status(M,n) \neq mo}{caid \bullet M \bullet dst \xrightarrow{?Rd(n)} caid \bullet M \bullet dst}$$

Fig. 25. Labelled input transitions.

$$\frac{\textsc{(Synch)}\quad \overline{Ca} \xrightarrow{!Rd(n)} \overline{Ca'}}{\langle \overline{CR},\ \overline{Ca},\ M\rangle \longrightarrow \langle \overline{CR},\ \overline{Ca'},\ M\rangle}
\qquad
\frac{\textsc{(SynchX)}\quad CR \notin \overline{CR_1} \quad CR, \overline{Ca}, M \xrightarrow{!RdX(n)} CR', \overline{Ca'}, M'}{\langle \overline{CR_1} \cup \{CR\},\ \overline{Ca},\ M\rangle \longrightarrow \langle \overline{CR_1} \cup \{CR'\},\ \overline{Ca'},\ M'\rangle}$$

$$\frac{\textsc{(Synch-DistX)}\quad Ca_1 \notin \overline{Ca} \quad CR, \overline{Ca}, M \xrightarrow{!RdX(n)} CR', \overline{Ca'}, M' \quad Ca_1 \xrightarrow{?RdX(n)} Ca_2}{CR, \overline{Ca} \cup \{Ca_1\}, M \xrightarrow{!RdX(n)} CR', \overline{Ca'} \cup \{Ca_2\}, M'}
\qquad
\frac{\textsc{(Synch-Dist)}\quad Ca_1 \notin \overline{Ca} \quad \overline{Ca} \xrightarrow{!Rd(n)} \overline{Ca'} \quad Ca_1 \xrightarrow{?Rd(n)} Ca_2'}{\overline{Ca} \cup \{Ca_1\} \xrightarrow{!Rd(n)} \overline{Ca'} \cup \{Ca_2\}}$$

Fig. 26. Transition rules for global synchronization/broadcast.