KeY-ABS: A Deductive Verification Tool for the Concurrent Modelling Language ABS*

Crystal Chang Din, Richard Bubel, and Reiner Hähnle

Department of Computer Science, Technische Universität Darmstadt, Germany
{crystald,bubel,haehnle}@cs.tu-darmstadt.de

Abstract. We present KeY-ABS, a tool for deductive verification of concurrent and distributed programs written in ABS. KeY-ABS allows to verify data dependent and history-based functional properties of ABS models. In this paper we give a glimpse of system workflow, tool architecture, and the usage of KeY-ABS. In addition, we briefly present the syntax, semantics and calculus of KeY-ABS Dynamic Logic (ABSDL). The system is available for download.

1 Introduction

KeY-ABS is a deductive verification system for the concurrent modelling language ABS [1, 8, 11]. It is based on the KeY theorem prover [2]. KeY-ABS provides an interactive theorem proving environment and allows one to prove properties of object-oriented and concurrent ABS models. The concurrency model of ABS has been carefully engineered to admit a proof system that is modular and permits to reduce correctness of concurrent programs to reasoning about sequential ones [3, 5]. The deductive component of KeY-ABS is an axiomatization of the operational semantics of ABS in the form of a sequent calculus for first-order dynamic logic for ABS (ABSDL). The rules of the calculus that axiomatize program formulae define a symbolic execution engine for ABS. The system provides heuristics and proof strategies that automate $\geq$90% of proof construction. For example, first-order reasoning, arithmetic simplification, symbolic state simplification, and symbolic execution of loop- and recursion-free programs are performed mostly automatically. The remaining user input typically consists of universal and existential quantifier instantiations.

ABS is a rich language with Haskell-like (first-order) datatypes, Java-like objects and thread-based as well as actor-based concurrency. In contrast to model checking, KeY-ABS allows to verify complex functional properties of systems with unbounded size [6]. In this paper we concentrate on the design of the KeY-ABS prover and its usage. KeY-ABS consists of around 11,000 lines of Java code (KeY-ABS + reused parts of KeY: ca. 100,000 lines). The rule base consists of ca. 10,000 lines written in KeY's *taclet* rule description language [2]. At http://www.envisage-project.eu/?page_id=1558 the KeY-ABS tool and a screencast showing how to use it can be downloaded.

* This work was done in the context of the EU project FP7-610582 *Envisage: Engineering Virtualized Services* (http://www.envisage-project.eu)

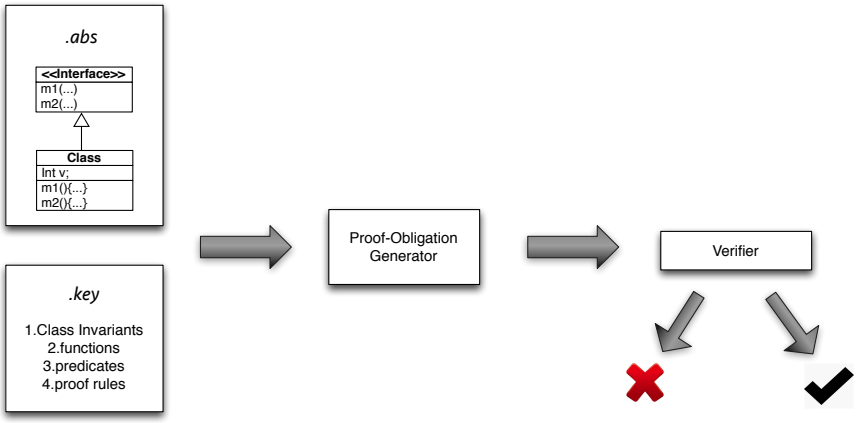

Fig. 1: Verification workflow of KeY-ABS

2 The Design of KeY-ABS

2.1 System Workflow

The input files to KeY-ABS comprise (i) an *.abs* file containing ABS programs and (ii) a *.key* file containing the class invariants, functions, predicates and specific proof rules required for this particular verification case. Given these input files, KeY-ABS opens a proof obligation selection dialogue that lets one choose a target method implementation. From the selection the proof obligation generator creates an ABSDL formula. By clicking on the **Start** button the verifier will try to automatically prove the generated formula. A positive outcome shows that the target method preserves the specified class invariants. In the case that a subgoal cannot be proved automatically, the user is able to interact with the verifier to choose proof strategies and proof rules manually. The reason for a formula to be unprovable might be that the target method implementation does not preserve one of the class invariants, that the specified invariants are too weak/too strong or that additional proof rules are required. The workflow of KeY-ABS is illustrated in Fig. 1.

2.2 The Concurrency Model of ABS

In ABS [1, 8, 11] two different kinds of concurrency are supported depending on whether two objects belong to the same or to different *concurrent object groups* (COGs). The affinity of an object to a COG is determined at creation time. The creator decides whether the object should be assigned to a new COG or to the COG of its creator. Within a COG several threads might exist, but only one of these threads (and hence one object) can be active at any time. Another thread can only take over when the current active thread explicitly releases control. In

other words, ABS realizes *cooperative scheduling* within a COG. All interleaving points occur syntactically in an ABS program in the form of an *await* or *suspend* statement by which the current thread releases control.

While one COG represents a single processor with task switching and shared memory, two different COGs run actually in parallel and are separated by a network. As a consequence, objects within the same COG may communicate either by asynchronous or by synchronous method invocation, while objects living on different COGs *must* communicate with asynchronous method invocation and message passing. Any asynchronous method invocation creates a so called *future* as its immediate result. Futures are a handle for the result value once it becomes available. Attempting to access an unavailable result blocks the current thread and its COG until the result value is available. To avoid this, retrieval of futures is usually guarded with an *await* that, instead of blocking, releases control in case of an unavailable result. Futures are first-class citizens and can be assigned to local variables, object fields, and passed as method arguments.

In contrast to current industrial programming languages such as C++ or Java which support multithreaded concurrency, ABS has a fully formalized concurrency model and natively supports distributed computation.

2.3 Verification Approach for ABS Programs

An asynchronous method call in ABS does not transfer the execution control from the caller to the callee, but leads to a new process on the called object. Remote field access is not supported by the language, so there is no shared variable communication between different objects. This is a stronger notion of privacy than in Java where other instances of the same class can access private fields. Consequently, different concurrent objects in ABS do not have aliases to the same object. Hence, state changes made by one object do not affect other objects. The concurrency model of ABS is compositional by design.

To make verification of ABS programs modular, KeY-ABS follows the monitor [9] approach. We define invariants for each ABS class and reason locally about each class by KeY-ABS. Each class invariant is required to hold after initialization in all class instances, before any process release point, and upon termination of each method call. Consequently, whenever a process is released, either by termination of a method execution or by a release point, the thread that gains execution control can rely on the class invariant to hold. The proof rule for compositional reasoning about ABS programs is given and proved sound in [5], by which we obtain system invariants from the proof results by KeY-ABS.

To write meaningful invariants of concurrent systems, it must be possible to refer to previous communication events. The observable behavior of a system can be described by *communication histories* over observable events [10]. Since message passing in ABS does not transfer the execution control from the caller to the callee, in KeY-ABS we consider separate events for method invocation, for reacting upon a method call, for resolving a future, and for fetching the value of a future. Each event can only be observed by one object, namely the object that generates it. Assume an object o calls a method on object o' and generates

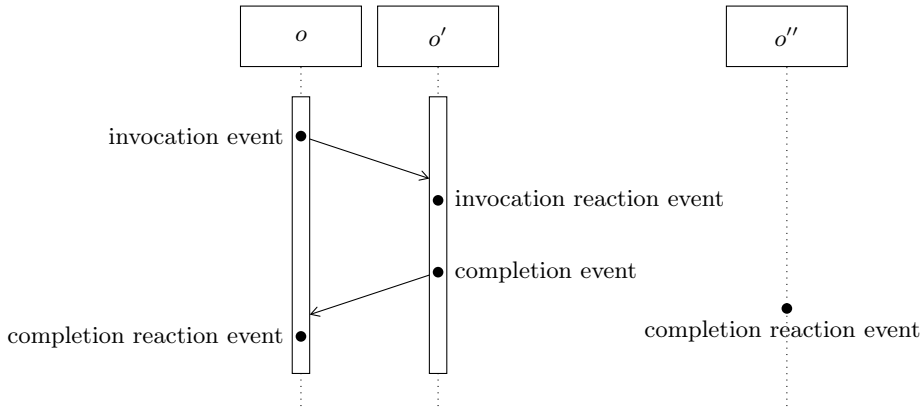

Fig. 2: History events and when they occur on objects o, o' and o''

a future identity *fr* associated to the method call. An invocation message is sent from o to o' when the method is invoked. This is reflected by the *invocation event* generated by o and illustrated by the sequence diagram in Fig. 2. An *invocation reaction event* is generated by o' once the method starts execution. When the method terminates, the object o' generates the *completion event*. This event reflects that the associated future is resolved, i.e., it contains the called method's result. The *completion reaction event* is generated by the caller o when fetching the value of the resolved future. Since future identities may be passed to a third object o'', that object may also fetch the future value, reflected by another *completion reaction event*, generated by o'' in Fig. 2. All events generated by one object forms the local history of the object. When composing objects, the local histories of the composed objects are merged to a common history, containing all the events of the composed objects [14].

2.4 Syntax and Semantics of the KeY-ABS Logic

Specification and verification of ABS models is done in ABS dynamic logic (ABSDL). ABSDL is a typed first-order logic plus a box modality: For a sequence of executable ABS statements S and ABSDL formulae P and Q, the formula $P \rightarrow [\mathsf{S}]Q$ expresses: If the execution of S starts in a state where the assertion P holds and the program terminates normally, then the assertion Q holds in the final state. Verification of an ABSDL formula proceeds by symbolic execution of S, where state modifications are handled by the *update* mechanism [2]. An *elementary update* has the form $U = \{loc := val\}$, where loc is a *location* expression and val is its new value term. Updates can only be applied to formulae or terms, i.e. $U\phi$. Semantically, the validity of $U\phi$ in state s is defined as the validity of ϕ in state s', which is state s with the modification of loc according to the update U. There are operations for sequential as well as parallel composition

of updates. Typically, loop- and recursion-free sequences of program statements can be turned into updates fully automatically. Given an ABS method m with body mb and a class invariant I, the ABSDL formula $I \rightarrow [mb]I$ expresses that the method m preserves the class invariant.

KeY-ABS natively supports concurrency in its program logic. In ABSDL we express properties of a system in terms of histories. This is realized by a dedicated, global program variable history, which contains the object local histories as a sequence of events. The history events themselves are elements of datatype HistoryLabel, which defines for each event type a constructor function. For instance, a completion event is represented as $compEv(o, fr, m, e)$ where o is the callee, fr the corresponding future, m the method name, and e the return result of the method execution. In addition to the history formalisation as a sequence of events, there are a number of built-in functions and predicates that allow to express common properties concerning histories. For example, function $getFuture(e)$ returns the future identity contained in the event e, and predicate $isInvocationEv(e)$ returns true if event e is an invocation event.

The type system of KeY-ABS reflects the ABS type system. Besides the type HistoryLabel, the type system of ABSDL contains, for example, the sequence type Seq, the root reference type any, the super type ABSAnyInterface of all ABS objects, the future type Future, and the type null, which is a subtype of all reference types. Users can define their own functions, predicates and types, which are used to represent the interfaces and abstract data types of a given ABS program.

2.5 Rule Formalisation

The user can interleave the automated proof search implemented in KeY-ABS with interactive rule application. For the latter, the KeY-ABS prover has a graphical user interface that is built upon the idea of direct manipulation. To apply a rule, the user first selects a focus of application by highlighting a (sub-)formula or a (sub-)term in the goal sequent. The prover then offers a choice of rules applicable at this focus. Rule schema variable instantiations are mostly inferred by matching. Fig. 3 shows an example of proof rule selection in KeY-ABS. The user is about to apply the $awaitExp$ rule that executes an await statement.

Another way to apply rules and provide instantiations is by drag and drop. The user simply drags an equation onto a term, and the system will try to rewrite the term with the equation. If the user drags a term onto a quantifier the system will try to instantiate the quantifier with this term.

The interaction style is closely related to the way rules are formalised in KeY-ABS. All rules are defined as *taclets* [2]. Here is a (slightly simplified) example:

\find $([\{method(source \leftarrow m, return \leftarrow (var : r, fut : u)) : \{\mathsf{return}\ exp;\}\}]\phi)$
\replacewith $(\{\mathsf{history} := seqConcat(\mathsf{history}, compEv(this, u, m, exp))\}\phi)$
\heuristics $(simplify\_prog)$

The rule symbolically executes a return statement inside a method invocation. It applies the *update* mechanism to the variable history, which is extended with

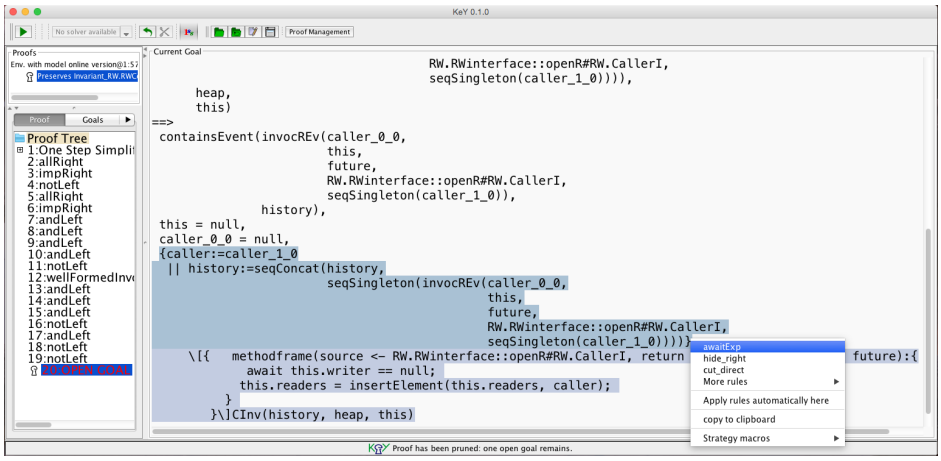

Fig. 3: Proof rule selection

a *completion event* capturing the termination and return value of the method execution. The **`find`** clause specifies the potential application focus. The taclet will be offered to the user on selecting a matching focus. The action clause **`replacewith`** modifies the formula in focus. The **`heuristics`** clause provides priority information to the parameterized automated proof search strategy. The taclet language is quickly mastered and makes the rule base easy to maintain and extend. A full account of the taclet language is given in [2].

2.6 KeY-ABS Architecture

Fig. 4 depicts the principal architecture of the KeY-ABS system. KeY-ABS is based on the KeY 2.0 platform—a verification system for Java. To be able to reuse most parts of the system, we had to generalize various subsystems and to abstract away from their Java specifics. For instance, the rule application logic of KeY made several assumptions which are valid for Java but not for other programming languages. Likewise, the specification framework of KeY, even though it provided general interfaces for contracts and invariants, made implicit assumptions that were insufficient for our communication histories and needed to be factored out. After refactoring the KeY system provides core subsystems (rule engine, proof construction, search strategies, specification

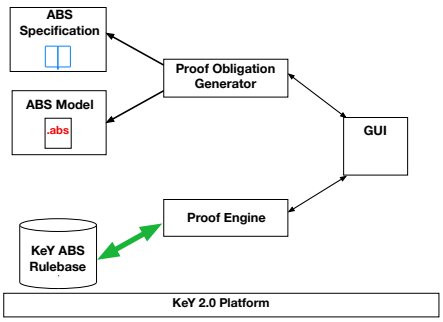

Fig. 4: The architecture of KeY-ABS

language, proof management etc.) that are independent of the specific program logic or target language. These are then extended and adapted by the ABS and Java backends.

The proof obligation generator needs to parse the source code of the ABS model and the specification. For the source code we use the parser as provided by the ABS toolkit [1, 15] with no changes. The resulting abstract syntax tree is then converted into KeY's internal representation. The specification parser for ABSDL formulas is an adapted version of the parser for JavaDL [2]. The rule base for ABSDL reuses the language-independent theories of the KeY tool, such as arithmetic, sequences and first-order logic. The rules for symbolic execution have been written from scratch for ABS as well as the formalisation of the history datatype.

3 The Usage of KeY-ABS

The ABS language was designed around a concurrency model whose analysis stays manageable. The restriction of the ABS concurrency model, specifically the fact that scheduling points are syntactically in the code, makes it possible to define a compositional specification and verification method. This is essential for being able to scale verification to non-trivial programs, because it is possible to specify and verify each ABS method separately, without the need for a global invariant. KeY-ABS follows the Design-by-Contract paradigm with an emphasis on specification of class invariants for concurrent and distributed programs in ABS. In the following we will show some examples of how and what we can specify in a class invariant.

```
class RWController implements RWinterface {
  Set<CallerI> readers = EmptySet; CallerI writer = null;

  Unit openR(CallerI caller){
    await writer == null;
    readers = insertElement(readers, caller);}

  Unit closeR(CallerI caller){
    readers = remove(readers, caller);}

  Unit openW(CallerI caller){
    await writer == null;
    writer = caller; readers = insertElement(readers, caller);}

  Unit closeW(CallerI caller){
    await writer == caller;
    writer = null; readers = remove(readers, caller);}

  String read(CallerI caller, Int key){...}

  Unit write(CallerI caller, Int key, String value){...}
}
```

Fig. 5: The controller class of the RW example in ABS

A history-based class invariant in ABSDL can relate the state of an object to the local history of the system. A simple banking system is verified in [3] by KeY-ABS, where an invariant ensures that the value of the account balance (a class attribute) always coincides with the value returned by the most recent call to a deposit or withdraw method (captured in the history). Here we use a more ambitious case study to illustrate this style of class invariant. In Fig. 5 an ABS implementation of the classic reader-writer problem [4] is shown. The RWController class provides read and write operations to clients and four methods to synchronize reading and writing activities: openR, closeR, openW and closeW.

The class attribute *readers* contains a set of clients currently with read access and *writer* contains the client with write access. The set of *readers* is extended by execution of openR or openW, and is reduced by closeR or closeW. The *writer* is added by execution of openW and removed by closeW. Two class invariants of the reader-writer example are (slightly simplified) shown in Fig. 6, in which the invariants $isReader$ and $isWriter$ express that the value of class attributes *readers* and *writer* are always equal to the set of relevant callers extracted from the current history. The keyword **invariants** opens a section where invariants can be specified. Its parameters declare program variables that can be used to refer to the history (historySV), the heap (heapSV, implicit by attribute access), and the current object (self, similar as Java's *this*).

```
\invariants(Seq historySV, Heap heapSV, ABSAnyInterface self) {
  isReader : RW.RWController {
    RW.RWController::self.readers
    = currentReaders(historySV)
  };

  isWriter : RW.RWController {
    insertElement(EmptySet, RW.CallerI::self.writer)
    = currentWriter(historySV)
  };
}
```

Fig. 6: Class invariants of the RW example

The functions $currentReaders(h)$ and $currentWriter(h)$ are defined inductively over the history h to capture a set of existing callers to the corresponding methods. The statistics of verifying these two invariants are in Fig. 7. For each of the six methods of the RWController class we show it satisfies $isReader$ and $isWriter$. For instance, the proof tree for verifying the invariant $isReader$ for method openR contains 3884 nodes and 12 branches. Verification of this case study was automatic except for a few instantiations of quantifiers and the rule application on inductive functions.[1]

[1] The complete model of the reader-writer example with all formal specifications and proofs is available at
https://www.se.tu-darmstadt.de/se/group-members/crystal-chang-din/rw.

invariants\methods	openR	closeR	openW	closeW	read	write
isReader	3884 – 12	1147 – 7	2836 – 9	3904 – 12	5459 – 26	3572 – 35
isWriter	2735 – 9	739 – 5	3891 – 12	3818 – 12	5056 – 29	4058 – 32

Fig. 7: Verification Result of RW example: # nodes – # branches

A history-based class invariant in ABSDL can also express structural properties of the history, for example, that an event ev_1 occurs in the history before an event ev_2 is generated. To formalize this kind of class invariant, quantifiers and indices of sequences are used to locate the events at certain positions of the history. Recently, we applied the KeY-ABS system to a case study of an ABS model of a Network-on-Chip (NoC) packet switching platform [12], called ASPIN (Asynchronous Scalable Packet Switching Integrated Network) [13]. It is currently the largest ABS program we have proved by KeY-ABS. We proved that ASPIN drops no packets and a packet is never sent in a circle by compositional reasoning. The ABS model, the specifications and the proof rules can be found in [6]. Both styles of class invariants mentioned above were used. The KeY-ABS verification approach to the NoC case study deals with an *unbounded* number of objects and is valid for generic NoC models for any m $\times$ n mesh in the ASPIN chip as well as any number of sent packets.

The global history of the whole system is formed by the composition of the local history of each instance of the class in the system. A *global invariant* can be obtained as a conjunction of instances of the class invariants verified by KeY-ABS for all objects in the system, adding wellformedness of the global history [7]. This allows to prove *global* safety properties of the system using *local* rules and symbolic execution, such as absence of packet loss and no circular packet sending. In contrast to model checking this allows us to deal effectively with unbounded target systems without suffering from state explosion.

4 Conclusion

We presented the KeY-ABS formal verification tool for the concurrent modelling language ABS. ABS is a rich, fully executable language with unbounded data structures and Java-like control structures as well as objects. It offers thread-based as well as actor-based concurrency with the main restriction being that scheduling points are made syntactically in the code ("cooperative scheduling"). KeY-ABS implements a compositional proof system [4, 5] for ABS. Its architecture is based on the state-of-art Java verification tool KeY and KeY-ABS reuses some of KeY's infrastructure.

KeY-ABS is able to verify global, functional properties of considerable complexity for unbounded systems. At the same time, the degree of automation is high. Therefore, KeY-ABS is a good alternative for the verification of unbounded, concurrent systems where model checking is not expressive or scalable enough.

References

1. The ABS tool suite. https://github.com/abstools/abstools. Last accessed 2015.05.17.
2. B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNCS*. Springer, 2007.
3. R. Bubel, A. Montoya, and R. Hähnle. Analysis of executable software models. In M. Bernardo, F. Damiani, R. Hähnle, E. Johnsen, and I. Schaefer, editors, *Formal Methods for Executable Software Models*, volume 8483 of *Lecture Notes in Computer Science*, pages 1–25. Springer International Publishing, 2014.
4. C. C. Din, J. Dovland, E. B. Johnsen, and O. Owe. Observable behavior of distributed systems: Component reasoning for concurrent objects. *Journal of Logic and Algebraic Programming*, 81(3):227–256, 2012.
5. C. C. Din and O. Owe. Compositional reasoning about active objects with shared futures. *Formal Aspects of Computing*, 27(3):551–572, 2015.
6. C. C. Din, S. L. T. Tarifa, R. Hähnle, and E. B. Johnsen. The NoC verification case study with KeY-ABS. Technical report, Department of Computer Science, Technische Universität Darmstadt, Germany, Feb. 2015.
7. J. Dovland, E. B. Johnsen, and O. Owe. Verification of concurrent objects with asynchronous method calls. In *Proc. IEEE Intl. Conference on Software Science, Technology & Engineering(SwSTE'05)*, pages 141–150. IEEE Computer Society Press, Feb. 2005.
8. R. Hähnle. The abstract behavioral specification language: A tutorial introduction. In E. Giachino, R. Hähnle, F. S. de Boer, and M. M. Bonsangue, editors, *Formal Methods for Components and Objects*, volume 7866 of *Lecture Notes in Computer Science*, pages 1–37. Springer Berlin Heidelberg, 2013.
9. C. A. R. Hoare. Monitors: An operating system structuring concept. *Commun. ACM*, 17(10):549–557, 1974.
10. C. A. R. Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Prentice Hall, 1985.
11. E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In B. Aichernig, F. S. de Boer, and M. M. Bonsangue, editors, *Proc. 9th International Symposium on Formal Methods for Components and Objects (FMCO 2010)*, volume 6957 of *Lecture Notes in Computer Science*, pages 142–164. Springer, 2011.
12. S. Kumar, A. Jantsch, M. Millberg, J. Öberg, J. Soininen, M. Forsell, K. Tiensyrjä, and A. Hemani. A network on chip architecture and design methodology. In *2002 IEEE Computer Society Annual Symposium on VLSI (ISVLSI 2002), 25-26 April 2002, Pittsburgh, PA, USA*, pages 117–124, 2002.
13. A. Sheibanyrad, A. Greiner, and I. M. Panades. Multisynchronous and fully asynchronous NoCs for GALS architectures. *IEEE Design & Test of Computers*, 25(6):572–580, 2008.
14. N. Soundararajan. A proof technique for parallel programs. *Theoretical Computer Science*, 31(1-2):13–29, 1984.
15. P. Y. H. Wong, E. Albert, R. Muschevici, J. Proença, J. Schäfer, and R. Schlatte. The ABS tool suite: modelling, executing and analysing distributed adaptable object-oriented systems. *STTT*, 14(5):567–588, 2012.