History-Based Specification and Verification of Scalable Concurrent and Distributed Systems

Crystal Chang Din[1(✉)], S. Lizeth Tapia Tarifa[2], Reiner Hähnle[1], and Einar Broch Johnsen[2]

[1] Department of Computer Science, Technische Universität Darmstadt, Darmstadt, Germany
{haehnle,crystald}@cs.tu-darmstadt.de
[2] Department of Informatics, University of Oslo, Oslo, Norway
{sltarifa,einarj}@ifi.uio.no

Abstract. The ABS modelling language targets concurrent and distributed object-oriented systems. The language has been designed to enable scalable formal verification of detailed executable models. This paper provides evidence for that claim: it gives formal specifications of safety properties in terms of histories of observable communication for ABS models as well as formal proofs of those properties. We illustrate our approach with a case study of a Network-on-Chip packet switching platform. We provide an executable formal model in ABS of a generic $m \times n$ mesh chip with an unbounded number of packets and verify several crucial properties. Our concern is formal verification of unbounded concurrent systems. In this paper we show how scalable verification can be achieved by compositional and local reasoning about history-based specifications of observable behavior.

1 Introduction

In this paper we address the formal verification of unbounded concurrent systems and show how *scalable* verification of functional behavior can be achieved by means of compositional and local reasoning about history-based specifications of observable behavior. To focus on high-level design, we consider models of the targeted systems. These models should be sufficiently abstract to facilitate reasoning, yet sufficiently concrete to faithfully reflect the data and control flow of the targeted system. ABS is a formal, executable modeling language for concurrent and distributed systems [26], specifically targeting this level of abstraction: (i) it combines functional, imperative, and object-oriented programming styles, allowing intuitive, modular, high-level modeling of concepts, domain and data; (ii) ABS models are fully executable and model system behavior

Supported by the EU projects FP7-610582 *Envisage: Engineering Virtualized Services* (http://www.envisage-project.eu) and FP7-612985 *UpScale: From Inherent Concurrency to Massive Parallelism through Type-based Optimizations* (http://www.upscale-project.eu).

M. Butler et al. (Eds.): ICFEM 2015, LNCS 9407, pp. 217–233, 2015.
DOI: 10.1007/978-3-319-25423-4_14

precisely [3]; (iii) ABS can model synchronous as well as asynchronous communication; (iv) ABS has been developed to provide the foundations for scalable formal verification: there is a program logic as well as a compositional proof system [17] that makes possible to prove global system properties by reasoning about object-local invariants; (v) ABS comes with an IDE and a range of analysis as well as productivity tools [41], specifically, there is a formal verification tool called KeY-ABS [18].

For scalable verification, we focus on behavioral properties specified in terms of communication histories. Communication histories have been used to give fully abstract semantics to concurrent object-oriented systems (e.g., [25]), describing observable behavior while abstracting from implementation detail. A fully abstract semantics captures the minimal information needed to characterize equivalence in all program contexts [32]. Hence, communication histories are the natural choice of specification formalism for compositional verification. We specify monitor-like invariants relating local states to local observable behavior, and compose specifications purely in terms of communication histories.

We provide empirical evidence of our scalability claim by way of a case study on a *Network-on-Chip* (NoC) [30] packet switching platform called ASPIN (Asynchronous Scalable Packet Switching Integrated Network) [37]. Our goal is to prove the correctness of an ABS model of an ASPIN NoC of *arbitrary, unbounded size* with respect to safety properties expressed in terms of communication histories. Concretely, we prove that "no packets are lost" and that "a packet is never sent in a circle". The main contributions of this paper are (i) a *formal model* of a generic $m \times n$ mesh ASPIN chip in ABS with unbounded number of packets, as well as a packet routing algorithm; (ii) the *formal specification using communication histories* of safety properties which together ensure that no packets are lost; and (iii) *compositional* and highly automated formal proofs, done with KeY-ABS, that the ABS model of ASPIN fulfills these safety properties.[1]

ABS was developed with the explicit aim to enable scalable verification of detailed, precisely modeled, executable, concurrent systems. Our paper shows that this claim is justified. Our work is the first *compositional* verification (in the sense made precise in Sect. 6) of a generic NoC model unbounded in the number of nodes and packets. It has been achieved with manageable effort and thus shows that our approach based on deductive verification is a viable alternative for the verification of concurrent systems.

Paper overview: Sect. 2 briefly introduces the modeling language ABS and Sect. 3 details formal specification based on communication histories, Sect. 4 provides background on deductive verification with expressive program logics, and Sect. 5 presents the ASPIN NoC case study. Section 6 explains how we achieved the formal specification and verification of the case study and gives details about the exact properties proved as well as the necessary effort. Section 7 sketches some directions for future work, Sect. 8 discusses related work and Sect. 9 concludes.

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

2 The ABS Modeling Language

ABS [26] is a behavioral specification language for developing abstract executable models of concurrent, distributed, and object-oriented systems. ABS offers a clean integration of concurrency and object orientation based on concurrent object groups (COGs). ABS permits synchronous as well as asynchronous communication [27], akin to Actors [1] and Erlang processes [7]. ABS offers a range of complementary modeling alternatives in a concurrent and object-oriented framework that integrates algebraic datatypes and functional and imperative programming styles with a Java-like syntax and a formal semantics [26]. Compared to object-oriented programming languages, ABS abstracts from low-level implementation choices such as imperative data structures. Compared to design-oriented languages like UML diagrams, it models data-sensitive control flow and it is executable. We now briefly introduce the functional and imperative layers of ABS.

The functional layer of ABS is used to model computations on the internal data of concurrent objects. It allows modelers to abstract from implementation details of imperative data structures at an early stage in the software design and thus allows data manipulation without committing to a particular low-level implementation choice. This layer combines a simple language for parametric algebraic data types (ADTs) and a pure first-order functional language which includes *expressions* such as variables, values, constructors, functions, and case expressions. ABS has a library with four predefined basic types (Bool, Int, String and Unit), and parametric datatypes (e.g., lists, sets, and maps). The predefined datatypes come with arithmetic and comparison operators, and the parametric datatypes have built-in standard functions. The type Unit is used as a return type for methods without explicit return value. All other types and functions are user-defined.

The imperative layer of ABS addresses concurrency, communication, and synchronization in the system design, and defines interfaces, classes, and methods in an object-oriented style. In ABS, each concurrent object group (COG) has its own thread of execution where one process is active and the others are suspended on a process queue. Classes can be *active* in the sense that their run method, if defined, automatically triggers a process upon creation. *Statements* are standard for sequential composition $s_1; s_2$, and for **skip**, **if**, **while**, and **return** constructs. In addition, ABS includes statements **await** and **suspend** for the explicit suspension of active processes, so scheduling in ABS is *cooperative*. The statement **suspend** unconditionally suspends the execution of the active process and moves this process to the queue. The statement **await** g conditionally suspends execution: the guard g controls thread release and consists of Boolean conditions and return tests (explained in the next paragraph). Just like expressions, the evaluation of guards is side-effect free. However, if g evaluates to false, the process is *suspended* and the execution thread becomes idle. When the execution thread is idle, an enabled task may be selected from the process queue by means of a default scheduling policy. The language also includes COG creation

new $C(\overline{e})$, method calls $o!m(\overline{e})$, and future dereferencing $fr.$**get** (here $\overline{e}$ denotes a lists of expressions).

Communication and *synchronization* are decoupled in ABS. Communication is based on asynchronous method calls, denoted by assignments of the form $fr{=}o!m(\overline{e})$ to future variables fr. Here, o is an object expression, m a method name, and $\overline{e}$ are expressions providing actual parameter values for the method invocation. (Local calls are written **this**$!m(\overline{e})$.) A future denotes a "mailbox" where the return value to the method call can be retrieved. After calling $fr{=}o!m(\overline{e})$, the variable fr refers to the corresponding future and the caller may proceed *without blocking*. Two operations on future variables control synchronization in ABS [13]. First, the guard **await** fr? *suspends the active process* unless a return to the call associated with fr has arrived, allowing other processes in the COG to execute. Second, the return value is retrieved by the expression $fr.$**get**, which *blocks all execution* in the COG until the return value is available. For example, the statement sequence $fr{=}o!m(\overline{e});x{=}fr.$**get** contains no suspension statement and, therefore, encodes commonly used *blocking calls*, abbreviated $x{=}o.m(e)$ (often referred to as synchronous calls). Futures are first-class citizens of ABS and can be passed around as method parameters. If the return value of a call is of no interest, the call may occur directly as a statement $o!m(e)$ with no associated future variable. This corresponds to asynchronous message passing. The details of the sequential execution of several threads inside a COG are not used in the verification techniques showcased in this paper and therefore we focus on single-object COGs (i.e., concurrent objects) in the sequel.

3 Observable Behavior

A distributed system can be specified by the externally observable behavior of its constituents. The behavior of each component is reflected in the possible *communication histories* over observable events [22]. Theoretically this is justified, because communication histories can be used for fully abstract semantics of object-oriented languages [25]. Here, we strive for *compositional* communication histories of asynchronously communicating systems. Therefore, it is appropriate to record separate events for object creation, method invocation, reaction upon a method call, resolving a future, and for fetching the value of a future. Each of these events is witnessed by merely one object, namely the generating object.

Figure 1 illustrates the relation among the observable events associated with an asynchronous method call. Assume that an object o calls a method m on an object o' with parameter values $\overline{e}$, and assume that u denotes the identity of the associated future. An invocation message is sent from o to o' when the method is invoked. This is reflected by the *invocation event* $invEv(o, o', u, m, \overline{e})$, generated by o. An *invocation reaction event* $invREv(o, o', u, m, \overline{e})$ is generated by o' once m starts to execute. When m has terminated, object o' generates the *future event* $futEv(o', u, m, e)$, reflecting that u receives the return value e. The *fetching event* $fetREv(o, u, e)$ is generated by o once the value of the resolved future is accessed. References u to futures bind all four event types together

and allow to filter out those events from an event history that relate to the same method invocation. Since future identities may be passed to other objects o'', these objects may also fetch the future value; this is reflected by the event $fetREv(o'', u, e)$, generated by o'' in Fig. 1. Based on these events, we formalize the notion of a communication history.

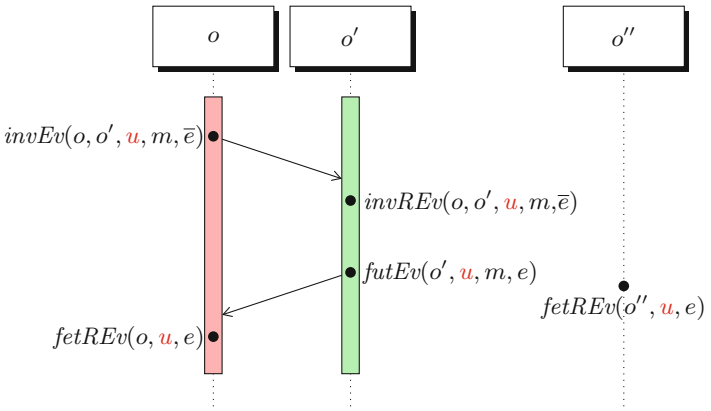

Fig. 1. Communication events and when they occur in the history

Definition 1 (Communication History). *The* communication history H *of a system of objects O is a sequence of events, as defined above, such that each event in H is generated by an object in O.*

For a history H, we let H/o abbreviate the projection of H to the events generated by o. Since each event is generated by a single object, it follows that the projections of a history to two different objects are disjoint.

Definition 2 (Local History). *For a (global) history H and an object o, the projection H/o is the* local history *of o.*

For a method call with future u, the possible ordering of the associated events is described by the regular expression

$$invEv(o, o', u, m, \overline{e}) \cdot invREv(o, o', u, m, \overline{e}) \cdot futEv(o', u, m, e)[\cdot fetREv(\_, u, e)]^*$$

for some fixed o, o', m, $\overline{e}$, e, and where "$\cdot$" denotes concatenation of events, "$\_$" denotes arbitrary values. Thus, the return value from a method call may be read several times (or not at all), each time with the same value, namely the value given in the preceding future event.

A communication history H is *wellformed* if the order of communication events follows the pattern defined above, the identities of generated futures are fresh, and the communicating objects are non-null.

Lemma 1. *The global history H of a system modeled with ABS and derived from its operational semantics, is wellformed.*

The formal definition of wellformedness and a proof of Lemma 1 are given in [16].

Invariants. Safety properties [4] take the form of *history invariants*, which are predicates over all finite sequences in the (prefix-closed) set of possible histories.

The class invariant serves as a contract for a class in ABS: Class invariants express a relation between the internal state of class instances and their observable communication. Class invariants are specified by a predicate over the class attributes and the local history. A class invariant must hold after the initialization of an object, it must be maintained by all methods, and it must hold at all processor release points (i.e., `await`, `suspend`) [15].

A global history invariant can be obtained from the class invariants associated with all objects in the system, adding wellformedness of the global history. This is made more precise in Sect. 6.2.

4 Deductive Verification

KeY-ABS [18] is a deductive verification system for constructing formal proofs about ABS programs, based on the KeY theorem prover [8]. A formal proof is a sequence of reasoning steps to show the truth of a formula (a theorem). The formal proof must lead without gaps from axioms to the theorem by applying proof rules.

The program logic of KeY-ABS is first-order dynamic logic for ABS (ABSDL) [17,18]. For a sequence of executable ABS statements $\mathcal{S}$ and ABSDL formulae P and Q, the formula $P \rightarrow [\mathcal{S}]Q$ expresses: If the execution of $\mathcal{S}$ starts in a state where the assertion P holds and the program terminates normally, then the assertion Q holds in the final state. Thus, 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 uses a Gentzen-style sequent calculus to prove ABSDL formulae. In sequent notation $P \rightarrow [\mathcal{S}]\,Q$ is written

$$\Gamma, P \vdash [\mathcal{S}]\,Q, \Delta$$

where Γ and Δ stand for (possibly empty) sets of side formulae. A sequent calculus as realized in ABSDL essentially constitutes a symbolic interpreter for ABS. For example, the assignment rule for local program variables is

$$\frac{\Gamma \vdash \{\mathtt{v} := \mathtt{e}\}[\mathtt{rest}]\phi, \Delta}{\Gamma \vdash [\mathtt{v} = \mathtt{e}; \mathtt{rest}]\phi, \Delta}$$

where v is a local program variable and e is a pure (side effect-free) expression. This rule rewrites the formula by moving the assignment from the program into a so-called *update* [8], as $\{\mathtt{v} := \mathtt{e}\}$ shown above, which captures state changes. The symbolic execution continues with the remaining program `rest`. Updates can be viewed as explicit substitutions that accumulate in front of the modality during symbolic program execution. Updates can only be applied to formulae or terms. Once the program to be verified has been completely executed and the modality is empty, the accumulated updates are applied to the formula after the modality, resulting in a pure first-order formula. Below we show a more complex proof rule, which captures asynchronous method invocation:

$$\mathsf{asyncCall}\ \frac{\begin{array}{l}\Gamma \vdash (o \neq \mathtt{null} \wedge \mathtt{wf}(h)), \Delta \\ \Gamma \vdash (\mathtt{futureIsFresh}(u, h) \rightarrow \\ \qquad \{\mathtt{fr} := u \,||\, h := h \cdot \mathit{invEv}(\mathit{this}, o, u, m, \overline{e})\}[\mathtt{rest}]\phi), \Delta\end{array}}{\Gamma \vdash [\mathtt{fr} = o\,!\mathtt{m}(\overline{e}); \mathtt{rest}]\phi, \Delta}$$

The rule has two premisses and splits the proof in two cases. The first premiss (on top) ensures that the callee is non-null and the current history h is wellformed. The second case introduces a constant u which represents the future generated for the result of this method invocation. The left side of the implication ensures that u is fresh in h and the right side updates the history by appending the *invocation event* generated by this call. We refer to [17] for the other ABSDL rules as well as soundness and completeness proofs of the ABSDL calculus.

```
type Pos = Pair<Int, Int>; // (x,y) coordinates
type Packet = Pair<Int, Pos>; // (id, destination)
type Buffer = Int;
data Direction = N | W | S | E | NONE;
          // north, west, south, east, the direction for not moving
data Port = P(Bool inState , Bool outState, Router rId, Buffer buff);
          // (input port state, output port state, neighbor router id, buffer size)
type Ports = Map<Direction, Port>;
```

Fig. 2. ADTs for the ASPIN model in ABS

5 The Network-on-Chip Case Study

Network-on-Chip (NoC) [30] is a packet switching platform for single chip systems which scales well to an arbitrary number of resources (e.g., CPU, memory). The NoC architecture is an $m \times n$ mesh of switches and resources which are placed on the slots formed by the switches. The NoC architecture is essentially the on-chip communication infrastructure.

Asynchronous Scalable Packet Switching Integrated Network (ASPIN) [37] is an example of a NoC with routers and processors. ASPIN has physically distributed routers in each core. Each router is connected to four neighboring routers

```
interface Router{
     Unit setPorts(Router e, Router w, Router n, Router s);
     Unit getPk(Packet pk, Direction srcPort);}

class RouterImp(Pos address, Int buffSize) implements Router {
    Ports ports = EmptyMap;
    Set<Packet> receivedPks = EmptySet; // received packages

    Unit setPorts(Router e, Router w, Router n, Router s){
      ports = map[Pair(N, P(True, True, n, 0)), Pair(S, P(True, True, s, 0)),
                  Pair(E, P(True, True, e, 0)), Pair(W, P(True, True, w, 0))];}

    Unit getPk(Packet pk, Direction srcPort){
      if (addressPk(pk) != address) {
        await buff(lookup(ports,srcPort)) < buffSize;
        ports = put(ports,srcPort,increaseBuff(lookup(ports,srcPort)));
        this!redirectPk(pk,srcPort);}
      else { // record that packet was successfully received
        receivedPks = insertElement(receivedPks, pk); } }

    Unit redirectPk(Packet pk, Direction srcPort){
      Direction direc = xFirstRouting(addressPk(pk), address);
      await (inState(lookup(ports,srcPort)) == True)
              && (outState(lookup(ports,direc)) == True);
      ports = put(ports, srcPort, inSet(lookup(ports, srcPort), False));
      ports = put(ports, direc, outSet(lookup(ports, direc), False));
      Router r = rId(lookup(ports, direc));
      Fut<Unit> f = r!getPk(pk, opposite(direc)); await f?;
      ports = put(ports, srcPort, decreaseBuff(lookup(ports, srcPort)));
      ports = put(ports, srcPort, inSet(lookup(ports, srcPort), True));
      ports = put(ports, direc, outSet(lookup(ports, direc), True)); } }
```

Fig. 3. A model of an ASPIN router using ABS

and each core is locally connected to one router. ASPIN routers are split into five separate modules (north, south, east, west, and local) with ports that have input and output channels and buffers. ASPIN uses input buffering for storage: each input channel has an independent FIFO buffer. Packets arriving from different neighboring routers (and from the local core) are stored in the respective FIFO buffer. Communication between routers uses a four-phase handshake protocol with request and acknowledgment messages between neighboring routers to transfer a packet. In ASPIN, the distributed X-first algorithm routes packets from input channels to output channels: packets first move along the X (horizontal) axis of the grid, and afterwards along the Y (vertical) axis to reach their destination. We model the functionality and routing algorithm of ASPIN in ABS starting from a model by Sharifi *et al.* [35,36], written in Rebeca [38]. In Sect. 6 we will formally verify our model using ABSDL.

We model each router as a concurrent object that communicates with other routers through asynchronous method calls. The algebraic data types used in our model are given in Fig. 2. We abstract from the local communication to cores, so each router has four ports and each port has an input and output channel, the identifier rId of the neighbor router, and a buffer. Packets are modeled as pairs that contain the packet identifier and the final destination coordinate.

```
def Direction xFirstRouting(Pos destination, Pos current) =
case x(current) < x(destination) {
  True => E;
  False => case x(current) > x(destination) {
             True => W;
             False => case y(current) < y(destination) {
                        True => S;
                        False => case y(current) > y(destination) {
                                   True => N;
                                   False => NONE; }; }; }; };
```

Fig. 4. X-first routing algorithm in ABS

The ABS model of a router is shown in Fig. 3. Method setPorts initializes the ports in a router and connects it to the neighbor routers. Packets are transferred using a protocol expressed by two methods redirectPk and getPk. The internal method redirectPk is called by the router to redirect a packet to a neighbor router. The X-first routing algorithm in Fig. 4 selects the port direc (and consequently the neighbor router). The parameter srcPort determines the local input buffer in which the packet is temporarily stored. As part of the communication protocol, the input channel of srcPort and the output channel of direc are blocked until the neighbor router confirms receipt of the packet, using f = r!getPk(...); **await** f? statements to simulate request and acknowledgment messages (here r is the Id of the neighbor router). The method getPk checks if the final destination of the packet is the current router, if so, it stores the packet, otherwise it temporarily stores the packet in the srcPort buffer and redirects it. The model uses standard library functions for maps and sets (e.g., put and lookup) and observers as well as other functions over the ADTs (e.g., addressPk, inState, decreaseBuff).

6 Formal Specification and Verification of the Case Study

We now formalize and verify safety properties for the ABS NoC model in ABSDL using the KeY-ABS verification tool. The application is based on the theory presented in Sects. 3 and 4, ensuring the correctness of the results. Our approach uses local reasoning about *RouterImp* objects and establishes a system invariant over the global history from invariants over the local histories of each object.

6.1 Local Reasoning

Observe that the four-event semantics for asynchronous communication outlined in Sect. 3 keeps the local histories of different objects disjoint. This makes it possible to reason locally about each object in terms of the local histories. Lemmas 2 and 3 present the history-based class invariants for RouterImp. We then discuss the proof obligations verified by KeY-ABS that stem from reasoning about our model in terms of these class invariants. Figure 5 illustrates the explanations.

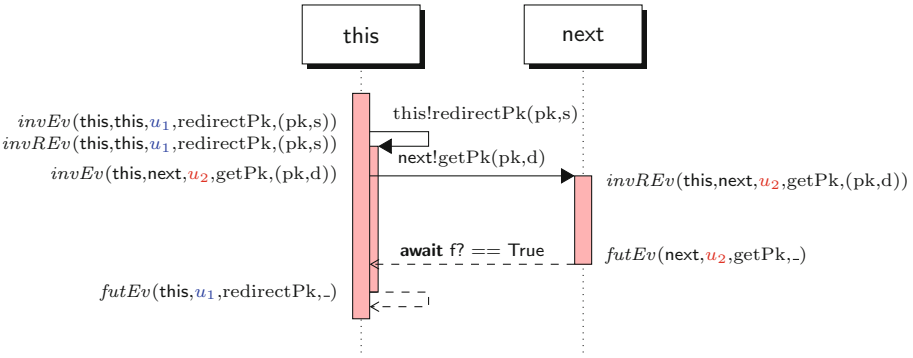

Fig. 5. Communication history between a router and its neighboring router **next**, to which the package is sent

Lemma 2. *Every time a router R terminates an execution of the* getPk *method, R must either have sent an internal invocation to redirect the packet or have stored the packet in its* receivedPks *set.*

We formalize this lemma as an ABSDL formula (slightly beautified):

$$
\begin{aligned}
&\forall i_1, u \,.\, 0 \leq i_1 < len(h) \wedge futEv(this, u, \mathsf{getPk}, \_) = at(h, i_1) \\
&\Rightarrow \\
&\quad \exists i_2, pk \,.\, 0 \leq i_2 < i_1 \wedge invREv(\_, this, u, \mathsf{getPk}, (pk, \_)) = at(h, i_2) \wedge \\
&\quad ((dest(pk) \neq address(this) \Rightarrow \\
&\qquad \exists i_3 \,.\, i_2 < i_3 < i_1 \wedge invEv(this, this, \_, \mathsf{redirectPk}, (pk, \_)) = at(h, i_3)) \vee \\
&\quad (dest(pk) = address(this) \Rightarrow pk \in \mathsf{receivedPks}))
\end{aligned}
$$

Here, "_" denotes a value without interest. The function $len(s)$ returns the length of sequence s, $at(s, i)$ the element located at index i of sequence s, $dest(pk)$ the destination address of packet pk, and $address(r)$ the address of router r.

This formula expresses that for every future event ev_1 of getPk with future identifier u in history h (capturing a termination of getPk), there is a corresponding invocation reaction event ev_2 that contains the sent packet pk. This is achieved by pattern matching with u in the preceding history. If *this* router is the destination of pk, then pk must be in its receivedPks set, otherwise an *invocation event* of redirectPk containing pk must occur in the history between ev_1 and ev_2. This invariant captures the properties of the state and is prefix-closed.[2]

[2] In the heap model of KeY-ABS, a heap value can potentially be modified when a process is released. Therefore, to prove the above property we need a slightly stronger invariant expressing that the address of a router in the heap is *rigid* (cannot be modified by any other process). Due to a current technical limitation of the tool, we proved the invariant for a slightly simplified model where the router address is a parameter of getPk. This modification does not affect the overall behavior of the model and will be lifted in future work.

Lemma 3. *Every time a router R terminates an execution of the* redirectPk *method, the input and output channels used to redirect the fetched packet are released, and the packet has been redirected to a neighbor router through an invocation of the* getPk *method.*

Again, we formalize this lemma as an ABSDL formula:

$$
\begin{aligned}
&\forall u\,.\,futEv(this, u, \mathsf{redirectPk}, \_) = at(h, len(h) - 1)\\
&\Rightarrow\\
&\exists i_1, i_2, pk, \mathsf{srcP}, \mathsf{dirP}\,.\,0 < i_1 < i_2 < len(h) - 1 \wedge\\
&\quad (invREv(this, this, u, \mathsf{redirectPk}, (pk, \mathsf{srcP})) = at(h, i_1) \wedge\\
&\qquad invEv(this, \_, \_, \mathsf{getPk}, (pk, opposite(\mathsf{dirP}))) = at(h, i_2)) \wedge\\
&\qquad\quad (inState(lookup(ports, \mathsf{srcP})) \wedge outState(lookup(ports, \mathsf{dirP})))
\end{aligned}
$$

This formula expresses that whenever the last event in the history h is a *future event* of redirectPk method (capturing termination of redirectPk), there are corresponding invocation reaction and invocation events which we find by pattern matching with the same future and packet in the previous history. The source port srcP and the direction port dirP used in the latest execution of redirectPk can be found in these two events. The input channel of srcP and the output channel of dirP must be released in the current state. This invariant captures the properties of the current state and is prefix-closed.

All three methods of RouterImp satisfy both invariants. The statistics for verifying the lemmas by KeY-ABS is given below (in terms of the proof size):

	setPorts		getPk		redirectPk	
	nodes	branches	nodes	branches	nodes	branches
Lemma 2	1638	12	11540	108	27077	200
Lemma 3	214	1	1845	11	4634	34

KeY-ABS provides heuristics and proof strategies that automate large parts of the proof construction. The remaining user input typically consists of universal and existential quantifier instantiations.

6.2 System Specification

A system property of an ABS program can be formulated as a global history invariant, which holds for all finite sequences in the prefix-closed set of possible global histories. The global history of an ABS program consists of the local histories of each object in the system, and is wellformed according to Lemma 1. We now want to derive a global system specification from the history-based class invariants of the system's objects.

The basis for local reasoning in the proof system for ABS is that class invariants must be satisfied at process release points and after method termination

(see Sect. 3), but class invariants need not be prefix-closed. Consequently, a local history invariant is in general weaker than the class invariant. For compositional reasoning, we may therefore need to weaken the class invariants in order to transform class invariants into prefix-closed history invariants. The system invariant can then be obtained directly from the history invariants of the composed objects since the local histories are disjoint. The proof rule for compositional reasoning about ABS programs is given and proved sound in [17], by which we obtain a system invariant below for the NoC model.

Let $I_{this}(h)$ denote the conjunction of the class invariants $I_{getPk}(this, h)$ and $I_{redirectPk}(this, h)$, defined in Lemmas 2 and 3, where h is the local history of *this* object. The class invariants are already prefix-closed and need not be weakened. Define a system invariant $I(H)$ as the conjunction of the instantiated class invariants of all RouterImp objects r in the system:

$$I(H) \triangleq \texttt{wf}(H) \wedge \bigwedge_{(r:\mathsf{RouterImp}) \in new_{ob}(H)} I_r(H/r)$$

Here, H denotes the global history of the system and $I_r(H/r)$ denotes the history invariant of r applied to the local history H/r of a router r as obtained by projection from H (Definition 2). The function $new_{ob}(H)$ returns the set of RouterImp objects generated within the system execution, as captured by H. History wellformedness, denoted $\texttt{wf}(H)$, ensures a proper ordering of the events that belong to the same method invocation. Each wellformed interleaving of the local histories represents a possible global history. As a consequence, we obtain:

Theorem 1. *Every time a router R terminates an execution of the redirectPk method, the pair of input and output channels used to redirect the fetched packet are released, and a neighbor router of R must either have sent an internal invocation to redirect the packet further or have stored the packet in its receivedPks set. Hence, the network does not drop any packets.*

More Properties. Besides Theorem 1 we proved in a similar fashion that a packet always moves towards its destination. This follows from two lemmas that hold locally and are proven with KeY-ABS: (i) whenever a router redirects a packet then it moves one step closer to its destination, and (ii) when a packet arrives at its destination then its distance to it becomes zero. The proof of (i) for redirectPk has 5178 nodes and 80 branches, the one of (ii) for getPk has 13401 nodes and 110 branches. As corollary we obtain that a packet is never sent in a circle.

Effort. The modeling of the NoC case study in ABS took ca. two person weeks. Formal specification and verification was mainly done by the first author of this paper who at the time was not experienced with the verification tool KeY-ABS. The effort for formal specification was ca. two person weeks and for formal verification of Lemmas 2, 3 ca. one person month, but this included training to use the tool effectively. Subsequent specification and verification of the property that a packet always moves towards its destination merely took one working day.

7 Future Work

Deadlock Analysis. In addition to history-based invariants, it is conceivable to prove other properties, such as deadlock-freedom. Deadlocks may occur in a system, for example, when a shared buffer between processes is full and one process can decrease the buffer size only if the other process increases the buffer size. This situation is prevented in the ABS model by disallowing self-calls before decreasing the size of the buffer (the method invocation of *getPk* within *redirectPk* in our model is an external call). It is possible to argue informally that our ABS model of NoC is indeed deadlock-free, but a formal proof with KeY-ABS is future work. The main obstacle is that deadlocks are a global property and one would need to find a way to encode sufficient conditions for deadlock-freedom into the local histories. There are deadlock analyzers for ABS [20], but these, like other approaches to deadlock analysis of concurrent systems, work only for a fixed number of objects.

Extensions of the Model. The ASPIN chip model presented in this paper can be extended with time (e.g., delays and deadline annotations) and scheduling (e.g., FIFO, EDF, user-defined, etc.) using Real-Time ABS [9]. A timed model would allow to run simulations and obtain results about the performance of the model. Adding scheduling to the model would make it possible to reason about the ordering of sent packets (using FIFO scheduling) or to express priority of packets. It is also possible to change the routing algorithm (Fig. 4) without the need to alter the RouterImp class (Fig. 3). Then one may compare the performance of different routing algorithms by means of simulations.

8 Related Work

Early work on verifying concurrent systems was non-compositional: interference freedom tests were used for shared variable concurrency [34] and cooperation tests for synchronous message passing [6]. Compositional approaches were introduced for shared variables in the form of rely-guarantee [28] and for synchronous message passing in the form of assumption-commitment [33]. Extending these principles for compositional verification, object invariants can be used to achieve modularity (e.g., [24]). Communication histories first appeared in the object-oriented setting [12] and then for CSP [22]. Soundararajan developed an axiomatic proof system for CSP using histories and projections [39], and Zwiers developed the first sound and complete proof system using histories [43]. Reasoning about asynchronous method calls and cooperative scheduling using histories was first done for Creol [19] and later adapted to Dynamic Logic [2]. Din introduced a proof system based on four communication events, significantly simplifying the proof rules [15] and extended the approach to futures [16,17]. This four-event proof system is the basis for KeY-ABS [18].

The pure history-based proof system of ABS requires strong hiding of local state: the state of other objects can only be accessed through method calls, so

shared state is internal and controlled by cooperative scheduling. Consequently, specifications can be purely local. More expressive specifications require significantly more complex proof systems; e.g., modifies-clauses in Boogie [24] or fractional permissions [21] in Chalice [31]. To specify fully abstract interface behavior these systems need to simulate histories in an ad hoc manner (e.g., [24, Fig. 1]). A combination of permission-based separation logic [5] and histories has recently been proposed for modular reasoning about multithread concurrency [42].

Formal analysis of NoC systems is usually done in specialized formalisms. Notably, xMAS is a language with a small set of primitives for specifying abstract microarchitectural models of communication fabrics [14]. It supports, for example, deadlock detection [40], model checking in Verilog by inferring inductive invariants for xMAS models [11], and compositional model-checking of bounded latency properties [23]. Among the approaches based on general specification formalisms, ACL2 has been used for non-compositional analysis of, e.g., message loss and deadlock-free routing (e.g., [10]). Event-B has been used to model and gradually refine 3D NoC systems in [29], and invariants for the models are verified using the Rodin tool. Similar to our work their modeling approach does not assume a specific number of routers. In contrast to our work their approach is based on a global specification of behavior which includes the assumption that a message can only be transferred a finite number of times before it reaches its destination (technically, their *switch* event is "anticipated").

Sharifi *et al.* [35,36] used the actor-based language Rebeca to study deadlock-freedom and successful package sending for the ASPIN chip and the X-first routing algorithm by means of non-compositional model-checking techniques. They work with configurations of fixed size, which triggered our interest in the verification of ASPIN models in a compositional and scalable manner. Compared to the Rebeca model, the ASPIN model in ABS is decoupled from the routing algorithm and uses object-oriented modeling concepts and high-level concurrency control, which makes it more compact and easier to comprehend. In contrast to most previous work, our approach works for an *unbounded* number of objects and it is valid for *generic* NoC models for any $m \times n$ mesh in the ASPIN chip as well as any number of sent packets.

9 Conclusion

We presented an approach to scalable verification of unbounded concurrent and distributed systems which allows *global safety properties* to be established using *local* verification rules and symbolic execution. The approach is realized in the proof system KeY-ABS, developed for the ABS modeling language. We demonstrated the viability of our verification approach by proving the correctness of safety properties for an ABS model of an ASPIN NoC of arbitrary, unbounded size. This is possible in our proof system, because each class invariant is independent of its class instances and properties are specified in terms of local communication histories. The paper develops a formal model of the case study, explains how local specifications are formalized using communication histories, and uses

KeY-ABS to obtain formal proofs of global properties such as "no packets are lost" and "a packet is never sent in a circle". This is, to the best of our knowledge, the first time that scalable, history-based reasoning techniques have been applied to NoC systems. Our work also shows that a general purpose modeling language and verification framework for concurrent and distributed systems is adequate for NoC systems. After an initial modeling and training effort, system properties can be specified and verified within hours or few days.

Acknowledgements. The authors gratefully acknowledge valuable discussions with Richard Bubel.

References

1. Agha, G.A.: ACTORS: A Model of Concurrent Computations in Distributed Systems. The MIT Press, Cambridge (1986)
2. Ahrendt, W., Dylla, M.: A system for compositional verification of asynchronous objects. Sci. Comput. Program. **77**(12), 1289–1309 (2012)
3. Albert, E., de Boer, F.S., Hähnle, R., Johnsen, E.B., Schlatte, R., Tapia Tarifa, S.L., Wong, P.Y.H.: Formal modeling of resource management for cloud architectures: an industrial case study using real-time ABS. J. SOCA **8**(4), 323–339 (2014)
4. Alpern, B., Schneider, F.B.: Defining liveness. Inf. Process. Lett. **21**(4), 181–185 (1985)
5. Amighi, A., Haack, C., Huisman, M., Hurlin, C.: Permission-based separation logic for multithreaded Java programs. LMCS **11**, 1–66 (2015)
6. Apt, K.R., Francez, N., de Roever, W.P.: A proof system for communicating sequential processes. ACM TOPLAS **2**(3), 359–385 (1980)
7. Armstrong, J.: Programming Erlang. Pragmatic Bookshelf (2007)
8. Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software. LNCS (LNAI), vol. 4334. Springer, Heidelberg (2007)
9. Bjørk, J., de Boer, F.S., Johnsen, E.B., Schlatte, R., Tapia, S.L.: User-defined schedulers for real-time concurrent objects. Innovations Syst. Softw. Eng. **9**(1), 29–43 (2013)
10. Borrione, D., Helmy, A., Pierre, L., Schmaltz, J.: A formal approach to the verification of networks on chip. EURASIP J. Embed. Syst. **2009**, 2:1–2:14 (2009)
11. Chatterjee, S., Kishinevsky, M.: Automatic generation of inductive invariants from high-level microarchitectural models of communication fabrics. Formal Methods Syst. Des. **40**(2), 147–169 (2012)
12. Dahl, O.-J.: Can program proving be made practical? In: Les Fondements de la Programmation, pp. 57–114. IRIA, December 1977
13. de Boer, F.S., Clarke, D., Johnsen, E.B.: A complete guide to the future. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 316–330. Springer, Heidelberg (2007)
14. Chatterjee, S., Kishinevsky, M., Ogras, Ü.Y.: xMAS: quick formal modeling of communication fabrics to enable verification. IEEE Des. Test Comput. **29**(3), 80–88 (2012)
15. Din, C.C., Dovland, J., Johnsen, E.B., Owe, O.: Observable behavior of distributed systems: component reasoning for concurrent objects. J. Logic Algebraic Program. **81**(3), 227–256 (2012)

16. Din, C.C., Owe, O.: A sound and complete reasoning system for asynchronous communication with shared futures. J. Logical Algebraic Methods Program. **83**(5–6), 360–383 (2014)
17. Din, C.C., Owe, O.: Compositional reasoning about active objects with shared futures. Formal Aspects Comput. **27**(3), 551–572 (2015)
18. Din, C.C., Bubel, R., Hähnle, R.: KeY-ABS: a deductive verification tool for the concurrent modelling language ABS. In: Felty, A., Middeldorp, A. (eds.) Automated Deduction - CADE-25. LNCS, vol. 9195, pp. 517–526. Springer, Switzerland (2015)
19. Dovland, J., Johnsen, E.B., Owe, O.: Verification of concurrent objects with asynchronous method calls. In: Proceedings of International Conference on Software Science, Technology & Engineering (SwSTE 2005), pp. 141–150. IEEE Press, February 2005
20. Giachino, E., Laneve, C., Lienhardt, M.: A framework for deadlock detection in core ABS. Softw. Syst. Model. 1–36 (2015). Springer. doi:10.1007/s10270-014-0444-y
21. Heule, S., Leino, K.R.M., Müller, P., Summers, A.J.: Abstract read permissions: fractional permissions without the fractions. In: Giacobazzi, R., Berdine, J., Mastroeni, I. (eds.) VMCAI 2013. LNCS, vol. 7737, pp. 315–334. Springer, Heidelberg (2013)
22. Hoare, C.A.R.: Communicating Sequential Processes. Prentice Hall, Upper Saddle River (1985)
23. Holcomb, D.E., Seshia, S.A.: Compositional performance verification of network-on-chip designs. IEEE Trans. CAD Integr. Circ. Syst. **33**(9), 1370–1383 (2014)
24. Jacobs, B., Piessens, F., Leino, K.R.M., Schulte, W.: Safe concurrency for aggregate objects with invariants. In: Proceedings of SEFM, pp. 137–147. IEEE (2005)
25. Jeffrey, A., Rathke, J.: Java JR: fully abstract trace semantics for a core Java language. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 423–438. Springer, Heidelberg (2005)
26. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: a core language for abstract behavioral specification. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) Formal Methods for Components and Objects. LNCS, vol. 6957, pp. 142–164. Springer, Heidelberg (2011)
27. Johnsen, E.B., Owe, O.: An asynchronous communication model for distributed concurrent objects. Softw. Syst. Model. **6**(1), 35–58 (2007)
28. Jones, C.B.: Development methods for computer programmes including a notion of interference. Ph.D. thesis, Oxford University, UK, June l981
29. Kamali, M., Petre, L., Sere, K., Daneshtalab, M.: Refinement-based modeling of 3D NoCs. In: Arbab, F., Sirjani, M. (eds.) FSEN 2011. LNCS, vol. 7141, pp. 236–252. Springer, Heidelberg (2012)
30. Kumar, S., Jantsch, A., Millberg, M., Öberg, J., Soininen, J., Forsell, M., Tiensyrjä, K., Hemani, A.: A network on chip architecture and design methodology. In: Proceedings of VLSI, pp. 117–124 (2002)
31. Leino, K.R.M., Müller, P., Smans, J.: Verification of concurrent programs with Chalice. In: Aldini, A., Barthe, G., Gorrieri, R. (eds.) Foundations of Security Analysis and Design V. LNCS, vol. 5705, p. 195. Springer, Heidelberg (2009)
32. Milner, R.: Fully abstract models of typed λ-calculi. Theoret. Comput. Sci. **4**, 1–22 (1977)
33. Misra, J., Chandy, K.M.: Proofs of networks of processes. IEEE Trans. Softw. Eng. **7**(4), 417–426 (1981)
34. Owicki, S.S., Gries, D.: An axiomatic proof technique for parallel programs I. Acta Informatica **6**, 319–340 (1976)

35. Sharifi, Z., Mohammadi, S., Sirjani, M.: Comparison of NoC routing algorithms using formal methods. In: Proceedings of Parallel and Distributed Processing Techniques and Applications (PDPTA 2013), vol. 2, pp. 474–482. CSREA Press (2013)
36. Sharifi, Z., Mosaffa, M., Mohammadi, S., Sirjani, M.: Functional and performance analysis of network-on-chips using actor-based modeling and formal verification. ECEASST **66**, 16 (2013)
37. Sheibanyrad, A., Greiner, A., Panades, I.M.: Multisynchronous and fully asynchronous NoCs for GALS architectures. IEEE Des. Test Comput. **25**(6), 572–580 (2008)
38. Sirjani, M., Jaghoori, M.M.: Ten years of analyzing actors: Rebeca experience. In: Agha, G., Danvy, O., Meseguer, J. (eds.) Formal Modeling: Actors, Open Systems, Biological Systems. LNCS, vol. 7000, pp. 20–56. Springer, Heidelberg (2011)
39. Soundararajan, N.: Axiomatic semantics of communicating sequential processes. ACM TOPLAS **6**(4), 647–662 (1984)
40. Verbeek, F., Schmaltz, J.: Hunting deadlocks efficiently in microarchitectural models of communication fabrics. In: International Conference on Formal Methods in Computer-Aided Design (FMCAD 2011), pp. 223–231. FMCAD Inc. (2011)
41. Wong, P.Y.H., Albert, E., Muschevici, R., Proença, J., Schäfer, J., Schlatte, R.: The ABS tool suite: modelling, executing and analysing distributed adaptable object-oriented systems. STTT **14**(5), 567–588 (2012)
42. Zaharieva-Stojanovski, M., Huisman, M., Blom, S.: Verifying functional behaviour of concurrent programs. In: Proceedings of 16th Workshop on Formal Techniques for Java-Like Programs (FTfJP 2014), pp. 4:1–4:6. ACM (2014)
43. Zwiers, J.: Compositionality, Concurrency and Partial Correctness: Proof Theories for Networks of Processes, and Their Relationship. LNCS, vol. 321. Springer, Heidelberg (1989)