The NoC Verification Case Study With KeY-ABS

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

Abstract. We present a case study on scalable formal verification of distributed systems that involves a formal model of a Network-on-Chip (NoC) packet switching platform. We provide an executable model of a generic $m \times n$ mesh chip with unbounded number of packets, the formal specification of certain safety properties, and formal proofs that the model fulfills these properties. The modeling has been done in ABS, a language that is intended to permit scalable verification of detailed, precisely modeled, executable, concurrent systems. Our paper shows that this is indeed possible and so advances the state-of-art verification of NoC systems. It also demonstrates that deductive verification is a viable alternative to model checking for the verification of unbounded concurrent systems that can effectively deal with state explosion.

1 Introduction

This paper presents a case study on *scalable* formal verification of the behavior of distributed systems. We create a formal, executable model of a *Network-on-Chip* (NoC) [27] packet switching platform called ASPIN (Asynchronous Scalable Packet Switching Integrated Network) [32]. This is a practically relevant system whose correctness is of great importance for the network infrastructures where it is deployed.

We model the ASPIN router architecture in the formal, executable, concurrent modeling language ABS [21, 23]. We use ABS for a number of reasons: (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 precisely [2]; (iii) ABS can model synchronous as well as asynchronous communication; (iv) ABS has been developed to permit scalable formal verification: there is a program logic [18] as well as a compositional proof system [16] that permits 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 [34], specifically, there is a formal verification tool called KeY-ABS [7].

The main contributions of this paper are as follows: (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 of a number of safety properties which together ensure that no packets are lost; (iii) formal proofs, done with KeY-ABS, that the ABS model of ASPIN fulfills these safety properties.[3]

ABS has been developed with the explicit aim to permit scalable verification of detailed, precisely modeled, executable, concurrent systems. Our paper shows that this claim is indeed justified. In addition it advances the state-of-the-art with the first successful verification of a generic NoC model that has an unbounded number of nodes and packets. This has been achieved with manageable effort and thus shows that deductive verification is a viable alternative to model checking for the verification of concurrent systems that can effectively deal with state explosion.

The paper is organized as follows: Sect. 2 gives a brief introduction into the modeling language ABS, Sect. 3 details our formal specification approach to system behavior, Sect. 4 provides some formal background on deductive verification with expressive program logics, and Sect. 5 presents the ASPIN NoC case study. Sect. 6 explains how we achieved the formal specification and verification of the case study and gives details about the exact properties proven as well as the necessary effort. In Sect. 7 we sketch possible directions for future work and Sect. 8 discusses related work and concludes.

2 The ABS Modeling Language

ABS [21, 23] is a formal behavioral specification language with a Java-like syntax. It combines functional and imperative programming styles to develop abstract executable models. ABS targets the modeling of concurrent, distributed, and object-oriented systems. It has a formal syntax and semantics and has a clean integration of concurrency and object orientation based on concurrent object groups (COGs) [23, 29]. ABS permits synchronous as well as asynchronous communication [24] akin to Actors [1] and Erlang processes [4]. ABS offers a wide variety of complementary modeling alternatives in a concurrent and object-oriented framework that integrates algebraic datatypes, functional programming and imperative programming. Compared to object-oriented programming languages, ABS abstracts from low-level implementation choices such as imperative data structures, and compared to design-oriented languages like UML diagrams, it models data-sensitive control flow and it is executable.

In addition, ABS also provides explicit and implicit time-dependent behavior [6], the modeling of deployment variability [25] and the modeling of variability in software product line engineering [9]. However, these functionalities of the language are not used in this paper and will not be further discussed. The rest of this section focuses on the syntax of ABS which contains a functional layer and

[3] 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.

Syntactic categories.			*Definitions.*
T	in	GroundType	$T ::= B \mid D \mid I \mid D\langle\overline{T}\rangle$
A	in	Type	$A ::= N \mid T \mid N\langle\overline{A}\rangle$
x	in	Variable	$Dd ::= \textbf{data}\ D[\langle\overline{A}\rangle] = [\overline{Cons}];$
e	in	Expression	$Cons ::= Co[(\overline{A})]$
v	in	Value	$F ::= \textbf{def}\ A\ fn[\langle\overline{A}\rangle](\overline{A}\ \overline{x}) = e;$
br	in	Branch	$e ::= x \mid v \mid Co[(\overline{e})] \mid fn(\overline{e}) \mid \textbf{case}\ e\ \{\overline{br}\}$
p	in	Pattern	$v ::= Co[(\overline{v})] \mid \textbf{null}$
			$br ::= p \Rightarrow e;$
			$p ::= \_ \mid x \mid v \mid Co[(\overline{p})]$

Fig. 1. Syntax for the functional layer of ABS. Terms $\overline{e}$ and $\overline{x}$ denote possibly empty lists over the corresponding syntactic categories, and square brackets [ ] optional elements.

an imperative layer. The details of the sequential execution of several threads inside a COG is not used in the verification techniques showcased in this paper and therefore we focus on single-object COGs (i.e., concurrent objects).

2.1 The Functional Layer of ABS

The functional layer of ABS is used to model computations on the internal data of the imperative layer. 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 low-level implementation choice. The functional layer combines a simple language for parametric algebraic data types (ADTs) and a pure first-order functional language. ABS includes a library with four predefined basic types (Bool, Int, String, and Unit), and parametric datatypes, (such as 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 formal syntax of the functional language is given in Fig. 1. The *ground types* T consist of basic types B as well as names D for datatypes and I for interfaces. In general, a type A may also contain type variables N (i.e., uninterpreted type names [28]). In *datatype declarations* Dd, a datatype D has a set of constructors *Cons*, each of which has a name Co and a list of types $\overline{A}$ for their arguments. *Function declarations* F have a return type A, a function name fn, a list of parameters $\overline{x}$ of types $\overline{A}$, and a function body e. Both datatypes and functions may be polymorphic and have a bracketed list of type parameters (e.g., Set<Bool>). The layered type system allows functions in the functional layer to be defined over types A which are parametrized by type variables but only applied to ground types T in the imperative layer; e.g., the head of a list is defined for List<A> but applied to ground types such as List<Int>.

Expressions e include variables x, values v, constructor expressions $Co(\overline{e})$, function expressions $fn(\overline{e})$, and case expressions **case** e $\{\overline{br}\}$. *Values* v are expressions that have reached a normal form: constructors applied to values $Co(\overline{v})$

Syntactic categories.
s in Stmt
e in Expr
b in BoolExpr
g in Guard

Definitions.

$$
\begin{array}{rl}
P & ::= \overline{IF}\ \overline{CL}\ \{[\overline{T}\ \overline{x};]\ s\} \\
IF & ::= \textbf{interface}\ I\ \{\ [\overline{Sg}]\ \} \\
CL & ::= \textbf{class}\ C\ [(\overline{T}\ \overline{x})]\ [\textbf{implements}\ \overline{I}]\ \{\ [\overline{T}\ \overline{x};]\ \overline{M}\} \\
Sg & ::= T\ m\ ([\overline{T}\ \overline{x}]) \\
M & ::= Sg\ \{[\overline{T}\ \overline{x};]\ s\} \\
s & ::= s; s \mid \textbf{skip} \mid x = rhs \mid \textbf{if}\ b\ \{\ s\ \}\ [\textbf{else}\ \{\ s\ \}] \mid \textbf{while}\ b\ \{\ s\ \} \\
 & \quad \mid \textbf{await}\ g \mid \textbf{suspend} \mid \textbf{return}\ e \\
rhs & ::= e \mid cm \mid \textbf{new}\ C\ (\overline{e}) \\
cm & ::= e!m(\overline{e}) \mid x.\textbf{get} \\
g & ::= b \mid x? \mid g \wedge g
\end{array}
$$

Fig. 2. Syntax for the imperative layer of ABS.

or **null**. *Case expressions* match a value against a list of branches $p \Rightarrow e$, where p is a pattern. Patterns are composed of the following elements: (1) wild cards _ which match anything, (2) variables x match anything if they are free or match against the existing value of x if they are bound, (3) values v which are compared literally, and (4) constructor patterns $Co(\overline{p})$ which match Co and then recursively match the elements $\overline{p}$. The branches are evaluated in the listed order, free variables in p are bound in the expression e.

2.2 The Imperative Layer of ABS

The imperative layer of ABS addresses concurrency, communication, and synchronization in the system design, and defines interfaces, classes, and methods in an object-oriented language with a Java-like syntax. In ABS, concurrent objects (single object COGs) are *active* in the sense that their run method, if defined, starts automatically upon creation.

Statements are standard for sequential composition $s_1; s_2$, and for **skip**, **if**, **while**, and **return** constructs. Cooperative scheduling in ABS is achieved by explicitly suspending the execution of the active process. 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 processor release and consists of Boolean conditions b and return tests $x?$ (explained in the next paragraph). Just like expressions e, the evaluation of guards g is side-effect free. However, if g evaluates to false, the processor is released and the process *suspended*. When the execution thread is idle, an enabled task may be selected from the pool of suspended tasks by means of a default scheduling policy. In addition to expressions e, the right hand side of an assignment $x{=}rhs$ includes object group creation **new** $C(\overline{e})$, method calls $o!m(\overline{e})$, and future dereferencing $x.\textbf{get}$. Method calls and future dereferencing are explained in the next paragraph.

Communication and *synchronization* are decoupled in ABS. Communication is based on asynchronous method calls, denoted by assignments of the form $f{=}o!m(\overline{e})$ to future variables f of type $\textbf{Fut}\langle T\rangle$, where T corresponds to the return type of the called method m. 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**!$m(\overline{e})$.) After calling $f{=}o!m(\overline{e})$, the future variable f refers to the return value of the call, and the caller may proceed *without blocking*. Two operations on future variables control synchronization in ABS. First, the guard **await** f**?** *suspends the active process* unless a return to the call associated with f has arrived, allowing other processes in the object to execute. Second, the return value is retrieved by the expression f.**get**, which *blocks all execution* in the object until the return value is available. Futures are first-class citizens of ABS and can be passed around as method parameters. The read-only variable **destiny**() refers to the future associated with the current process [13]. The statement sequence $x{=}o!m(e)\,;v{=}x\,.\,$**get** contains no suspension statement and, therefore, encodes commonly used *blocking calls*, abbreviated $v{=}o.m(e)$ (often referred to as synchronous calls). 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 syntax of the imperative layer of ABS is given in Fig. 2. A program P consists of lists of interface and class declarations followed by a main block $\{\overline{T}\ \overline{x}; s\}$, which is similar to a method body. An interface IF has a name I and method signatures Sg. A class CL has a name C, interfaces $\overline{I}$ (specifying types for its instances), class parameters and state variables x of type T, and methods M (The *attributes* of the class are both its parameters and state variables). A method signature Sg declares the return type T of a method with name m and formal parameters $\overline{x}$ of types $\overline{T}$. M defines a method with signature Sg, local variable declarations $\overline{x}$ of types $\overline{T}$, and a statement s. Statements may access attributes, locally defined variables (including the read-only variables **this** for self-reference and **destiny**() explained above), and the method's formal parameters. There are no type variables at the imperative layer of ABS.

3 Observable Behaviour

The observable behavior of a system can be described by *communication histories* over observable events [22]. Since message passing in ABS is *asynchronous*, we consider separate events for method invocation, reacting upon a method call, resolving a future, and for fetching the value of a future. Each event can only be observed by one object, namely the generating object. Assume an object o calls a method m on object o' with input values $\overline{e}$ and where fr 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* $invEv(o, o', fr, m, \overline{e})$ generated by o and illustrated by the sequence diagram in Fig. 3. An *invocation reaction event* $invREv(o, o', fr, m, \overline{e})$ is generated by o' once the method starts execution. When the method terminates, the object o' generates the *future event* $futEv(o', fr, m, e)$. This event reflects that fr is resolved with return value e. The *fetching event* $fetREv(o, fr, e)$ is generated by o when fetching the value of the resolved future. References fr to futures bind all four event types together and allow to filter out those events from an event history that relate to

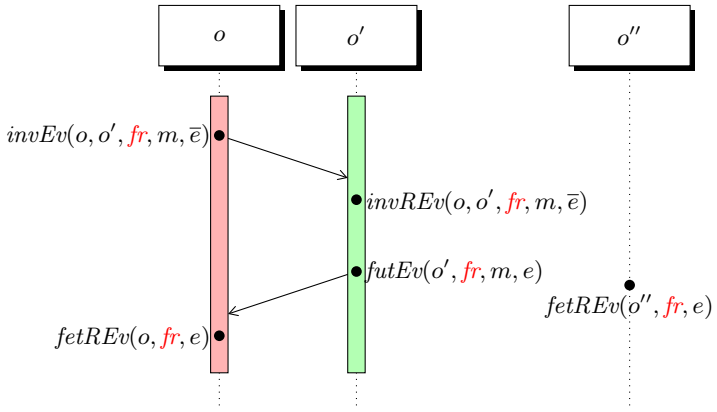

Fig. 3. History events and when they occur

the same method invocation. Since future identities may be passed to another object o'', that object may also fetch the future value, reflected by the event $fetREv(o'', fr, e)$, generated by o'' in Fig. 3.

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

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

for some fixed o, o', m, $\overline{e}$, e, and where "$\cdot$" denotes concatenation of events, "$\_$" denotes arbitrary values. Thus the result value may be read several times, each time with the same value, namely that given in the preceding future event. A communication history is *wellformed* if the order of communication events follows the pattern defined above, the identities of the generated future is fresh, and the communicating objects are non-`null`.

Invariants Class invariants express a relation between the internal state and observable communication of class instances. They are specified by a predicate over the class attributes and the local history. A class invariant must hold after initialization, it must be maintained by all methods, and it must hold at all processor release points (i.e., `await`, `suspend`).

A *global invariant* can be obtained as a conjunction of the class invariants for all objects in the system, adding wellformedness of the global history [19]. This is made more precise in Sect. 6.2 below.

4 Deductive Verification

A formal proof is a sequence of reasoning steps designed to convince the reader about the truth of some formulae, i.e., a theorem. In order to do this the proof must lead without gaps from axioms to the theorem by applying proof rules.

KeY-ABS is a deductive verification system for ABS programs based on the KeY theorem prover [5]. As a program logic it uses first-order dynamic logic for ABS (ABSDL) [7, 16]. For an ABS program $\mathcal{S}$ and ABSDL formulae P and Q, the formula $P \rightarrow [\mathcal{S}]Q$ expresses: If the execution of a program $\mathcal{S}$ starts in a state where the assertion P holds and the program terminates normally, then the assertion Q holds in the final state. Hence, $[\cdot]$ acts as a partial correctness modality operator. 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. We use a Gentzen-style sequent calculus to prove ABSDL formulae. Within a sequent we represent $P \rightarrow [\mathcal{S}]Q$ as

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

where Γ and Δ stand for (possibly empty) sets of formulae. A sequent calculus as realized in ABSDL essentially simulates a symbolic interpreter for ABS. The assignment rule for a local program variable 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 $\{\mathtt{v} := \mathtt{e}\}$, which captures state changes. The symbolic execution continues with the remaining program `rest`. Updates [5] 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 the proof rule for asynchronous method invocations:

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

This proof rule has two premisses and splits the proof into two branches. The first premiss on top ensures that the callee is non-null and the current history h is wellformed. The second branch introduces a constant u which represents the generated future as the placeholder for the method result. The left side of the implication ensures that u is fresh in h and updates the history by appending the *invocation event* for the asynchronous method call. We refer to [16] for the other ABSDL rules as well as soundness and completeness proofs of the ABSDL calculus.

5 Network-on-Chip Case Study

Network-on-Chip (NoC) [27] is a packet switching platform for single chip systems which scales well to an arbitrary number of resources (e.g., CPU, memory,

```
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, none
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. 4. ADTs for the ASPIN model in ABS

etc.). 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 essentially is the on-chip communication infrastructure. ASPIN (Asynchronous Scalable Packet Switching Integrated Network) [32] is an example of a NoC with routers and processors. ASPIN has physically distributed routers in each core. Each router is connected to four other neighboring routers 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 the storage strategy of input buffering, and each input channel is provided with 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 is established using a four-phase handshake protocol. The protocol uses request and acknowledgment messages between neighboring routers to transfer a packet. ASPIN uses the distributed X-first algorithm to route packets from input channels to output channels. Using this algorithm, packets move along the X (horizontal) direction in the grid first, and afterwards along the Y (vertical) direction to reach their destination. The X-first algorithm is claimed to be deadlock-free [32]. In this section we model the functionality and routing algorithm of ASPIN in ABS. As a starting point we use the ASPIN model by Sharifi *et al.* [30, 31]. In Sect. 6 we will formally verify our model in ABSDL.

We model each router as an object that communicates with other routers through asynchronous method calls. The abstract data types used in our model are given in Fig. 4. We abstract away 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.

The ABS model of a router is given in Fig. 5. The method setPorts initializes all the ports in a router and connects it with the corresponding neighbor routers. Packets are transferred using a protocol expressed in our model with two methods redirectPk and getPk. The internal method redirectPk is called when a router wants to redirect a packet to a neighbor router. The X-first routing algorithm in Fig. 6 decides which port direc (and as a consequence which neighbor router) to choose. The parameter srcPort determines in which input buffer the packet is temporally and locally stored. As part of the communication protocol, the input channel of srcPort and the output channel of direc are blocked until the neighbor

```
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. 5. A model of an ASPIN router using ABS

router confirms that it has gotten 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 temporally stores the packet in the srcPort buffer and redirect it. The model uses standard library functions for maps and sets (e.g, put, lookup, etc.) and observers as well as other functions over the ADTs (e.g., inState, decreaseBuff, etc.). Fig. 7 depicts a scenario with a 2×2 ASPIN chip. The sequence diagram shows how the different methods in the different routers are distributively called when a packet is sent from router R00 to router R11.

Simulation. The behavior of the ASPIN model in ABS can be analyzed using simulations. The operational semantics of ABS [23, 25] has been specified in rewriting logic which allows ABS models to be analyzed using rewriting tools. A simulation tool for ABS based on Maude [10] is part of the ABS tool set [34]. Given an initial configuration of a 4×4 mesh, we have executed test cases where: (1) a router is the destination of its own generated packet, (2) successful arrival of packets between two neighboring routers which send packets to each other, and (3) many packets sent through the same port at the same time.

```
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. 6. X-first routing algorithm in ABS

6 Formal Specification and Verification of the Case Study

In this section we formalize and verify global safety properties about our ABS NoC model in ABSDL using the KeY-ABS verification tool. This excludes any possibility of error at the level of the ABS model. Central to our verification effort are communication histories that abstractly capture the system state at any point in time [11]. Specifically, partial correctness properties are specified by finite initial segments of communication histories of the system under verification. A *history invariant* is a predicate over communication histories which holds for all finite sequences in the (prefix-closed) set of possible histories, thus expressing safety properties [3]. Our verification 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

Object-oriented programming supports modular design by providing classes as the basic modular unit. Our four event semantics (described in Sect. 3) keeps the local histories of different objects disjoint, so it is possible to reason locally about each object. For ABS programs, the class invariants must hold after initialization of all class instances, must be maintained by all methods and they must hold at all process release points so that they can serve as a method contracts. We present the class invariants for RouterImp in Lemma 1 and 2 and we show the proof obligations shown by KeY-ABS that result from the verification of our model against the class invariants. Fig. 8 illustrates the explanations.

Lemma 1. *Whenever a router R terminates an execution of the getPk method, then R must either have sent an internal invocation to redirect the packet or have stored the packet in its receivedPks set.*

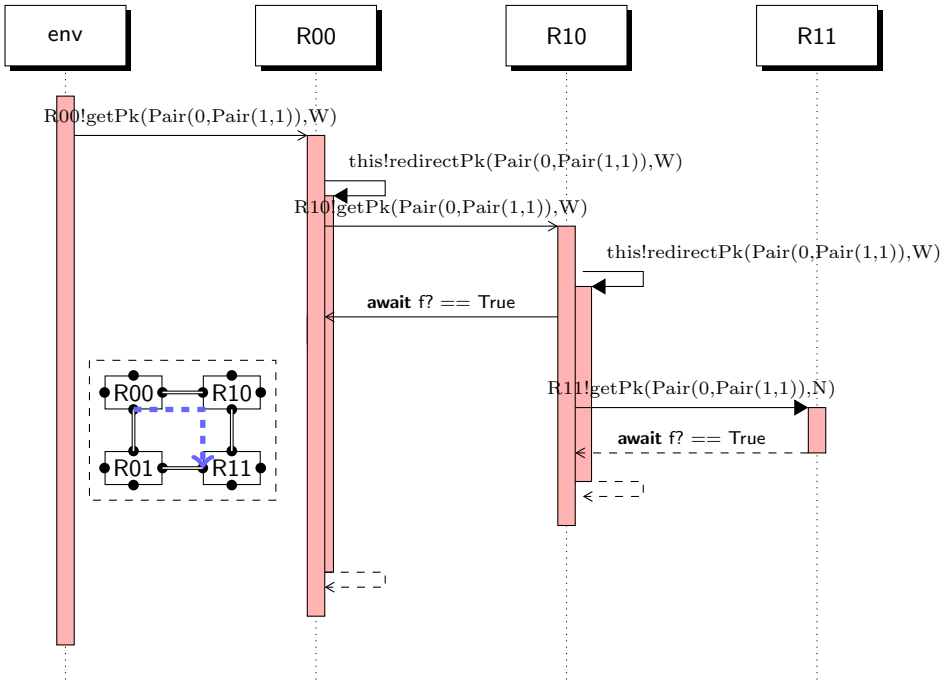

Fig. 7. A sequence diagram for a 2×2 ASPIN chip sending a packet to router R11

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

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

where "_" denotes a value that is of no interest. The function $\mathit{len}(s)$ returns the length of the sequence s, the function $\mathit{at}(s, i)$ returns the element located at the index i of the sequence s, the function $\mathit{dest}(pk)$ returns the destination address of the packet pk, and $\mathit{address}(r)$ returns the address of the router r.

This formula expresses that for every *future event* ev_1 of getPk with future identifier u found in history h we can find by pattern matching with u in the preceding history a corresponding *invocation reaction event* ev_2 that contains the sent packet pk. 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 be found in the history between events ev_1 and ev_2.

Remark 1. In the heap model of KeY-ABS, any value stored in the heap can be potentially modified while a process is released. Therefore, to prove the above

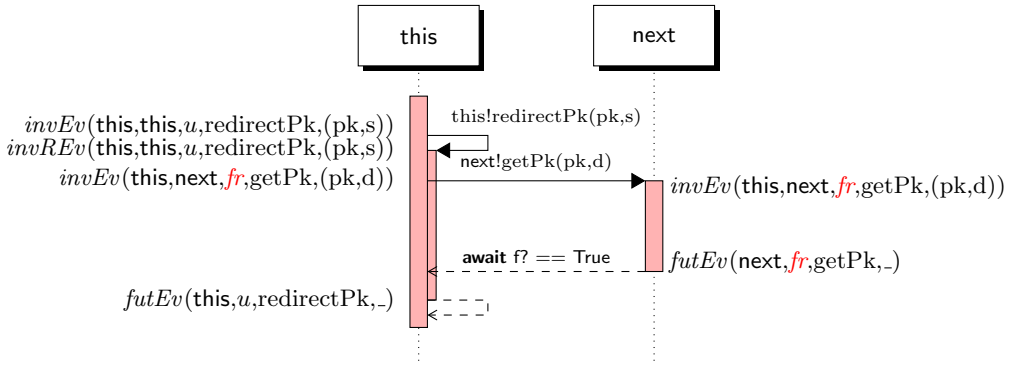

Fig. 8. Communication history between a router and its neighboring router **next** where the package is sent to

property we need a somewhat stronger invariant expressing that the address of a router stored in the heap is rigid (cannot be modified by any other process). Due to a current technical limitation, we proved the invariant for a slightly simplified version of the model where the router address is passed as a parameter of getPk. This technical modification does obviously not affect the overall behavior of the model and will be lifted in future work.

Lemma 2. *Whenever a router R terminates an execution of redirectPk, the input channel of srcPort and the output channel of direc are released.*

Again, we formalize this lemma as an ABSDL formula:

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

This formula expresses that whenever the last event in the history h is a *future event* of redirectPk method, by pattern matching with the same future and packet in the previous history, we can find the matched *invocation reaction event* and the *invocation event*. In these two events we filter out the source port $srcP$ and the direction port $dirP$ used in the latest run of redirectPk. 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. With KeY-ABS we proved that the RouterImp class of our model satisfies this invariant.

6.2 System Specification

The global history of the system is composed of the local histories of each instance of each class. However, communication between asynchronous concurrent objects is performed by asynchronous method calls, so messages may in general be delayed in the network. The observable behavior of a system includes the possibility that the order of messages received by the callee is different from the order of messages sent by the caller. The necessary assumptions about message ordering in our setting are captured by a global notion of wellformed history.

Lemma 3. *The global history H of a system S modeled with ABS and derived from its operational semantics, is wellformed, i.e., the predicate $\mathtt{wf}(H)$ holds.*

The formal definition of `wf` and a proof of the lemma are in [17], an informal definition is given in Sect. 3 above (see also Fig. 3).

Let $I_{this}(h)$ be the conjunction of $I_{getPk}(this, h)$ and $I_{redirectPk}(this, h)$, which are the class invariants defined in Lemma 1 and 2 where h is the local history of *this*. The local histories represent the activity of each concurrent object. We formulate a system invariant by the conjunction of the instantiated class invariants of all RouterImp objects r:

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

where H is the global history of the system and $I_r(H/r)$ is the object invariant of r instantiated from the class invariant $I_{this}(h)$. The local history of r is obtained by the projection H/r from the global history. The function $new_{ob}(H)$ returns the set of RouterImp objects generated within the system execution captured by H. Each wellformed interleaving of the local histories represents a possible global history. History wellformedness $\mathtt{wf}(H, new_{ob}(H))$ ensures proper ordering of those events that belong to the same method invocation. The composition rule was proven sound in [16]. As a consequence, we obtain:

Theorem 1. *Whenever a router R releases a pair of input and output channels used for redirecting a receiving packet, the next router of R must either have sent an internal invocation to redirect the packet or have stored the packet in its receivedPks set. Hence, the network does not drop any packets.*

Effort. The modelling 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 ca. one person month, but this included training to use the tool effectively. For an experienced user of KeY-ABS, we estimate that these figures would be three person days and one person week, respectively.

```
def Int distance(Pos destination, Pos current) =
    abs(x(destination) − x(current)) + abs(y(destination) − y(current));

assert (distance(addressPk(pk),addr)==distance(addressPk(pk),prevAddr)−1);
```

Fig. 9. A function to calculate the distance between the current position and the final destination of a packet for the X-first routing algorithm

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 easily be extended with time (e.g, delays and deadline annotations) and scheduling (e.g., FIFO, EDF, user-defined, etc.) using Real-Time ABS [6]. The extension with time would allow us to run simulations and obtain results about the performance of the model. Adding scheduling to the model would allow us to, for example, guarantee the ordering of the sent packets (using FIFO scheduling) or to express priority of packets. We can also easily change the routing algorithm in Fig. 6 without any need to alter the RouterImp class in Fig. 5. It is possible to compare the performance of different routing algorithms by means of simulations.

Runtime Assertion Checking. Another extension to the model could be runtime assertion checking (RAC) [18], for example, to ensure that packets make progress towards their final destination. For this one would use the distance function in Fig. 9 and simply include the assertion into the model, where addr is the address of the current router and prevAddr is the address of the previous neighbor router from where the packet was redirected. RAC is already supported by the ABS tool set and can be used for this case study, but to keep the paper focussed we decided not to report the results here.

8 Related Work and Conclusion

Previous work on formal modeling of NoC includes [8, 12, 30, 31]. The papers [30, 31], which were a starting point for our work, present a formal model of NoC in the actor-based modeling language Rebeca [26, 33]. In [30], the authors model the functional and timed behavior of ASPIN (with the X-first routing algorithm). To analyze their model, they used the model checker of Rebeca, Afra [26], to guarantee deadlock-freedom and successful packet sending for *specific* chip configurations. They also measure the maximum end-to-end latency of packets. In [31] the authors compare the performance of different routing algorithms. The ASPIN model presented in this paper does not capture timing behavior and uses the X-first routing algorithm, but timing behavior can easily be added and other routing algorithms can be plugged into the model as explained in Sect. 7. Compared to the Rebeca model, our ABS model of the ASPIN chip is deadlock-free and more compact. It is decoupled from the routing algorithm and easier to understand than the Rebeca model, because ABS permits intuitive, object-oriented modeling of the involved concepts, as well as high-level concepts for modeling concurrency. Our verification approach 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. This is possible, because we use *deductive verification* in the expressive program logic ABSDL with the verification tool KeY-ABS [7, 16] and formal specification of observable behavior [14, 15]. This allowed us to prove *global safety properties* of the system using *local* rules and symbolic execution. In contrast to model checking this allows us to deal effectively with unbounded target systems without encountering state explosion.

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

References

1. G. A. Agha. *ACTORS: A Model of Concurrent Computations in Distributed Systems*. The MIT Press, Cambridge, Mass., 1986.
2. E. Albert, F. S. de Boer, R. Hähnle, E. B. Johnsen, R. Schlatte, S. L. Tapia Tarifa, and P. Y. H. Wong. Formal modeling of resource management for cloud architectures: An industrial case study using Real-Time ABS. *Journal of Service-Oriented Computing and Applications*, 8(4):323–339, Dec. 2014.
3. B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, Oct. 1985.
4. J. Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.
5. B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNCS*. Springer, 2007.
6. J. Bjørk, F. S. de Boer, E. B. Johnsen, R. Schlatte, and S. L. Tapia Tarifa. User-defined schedulers for real-time concurrent objects. *Innovations in Systems and Software Engineering*, 9(1):29–43, 2013.

7. R. Bubel, A. Flores Montoya, and R. Hähnle. Analysis of executable software models. In M. Bernardo, F. Damiani, R. Hähnle, E. B. Johnsen, and I. Schaefer, editors, *Executable Software Models: 14th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, Bertinoro, Italy*, volume 8483 of *LNCS*, pages 1–27. Springer, June 2014.
8. Y.-R. Chen, W.-T. Su, P.-A. Hsiung, Y.-C. Lan, Y.-H. Hu, and S.-J. Chen. Formal modeling and verification for network-on-chip. In *Green Circuits and Systems (ICGCS), 2010 International Conference on*, pages 299–304, 2010.
9. D. Clarke, N. Diakov, R. Hähnle, E. B. Johnsen, I. Schaefer, J. Schäfer, R. Schlatte, and P. Y. H. Wong. Modeling spatial and temporal variability with the HATS abstract behavioral modeling language. In M. Bernardo and V. Issarny, editors, *Proc. 11th Intl. School on Formal Methods for the Design of Computer, Communication and Software Systems (SFM 2011)*, volume 6659 of *LNCS*, pages 417–457. Springer, 2011.
10. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 285:187–243, Aug. 2002.
11. O.-J. Dahl. *Verifiable Programming*. International Series in Computer Science. Prentice Hall, New York, N.Y., 1992.
12. S. Dasgupta. *Formal design and synthesis of GALS architectures*. PhD thesis, University of Newcastle, January 2008.
13. F. S. de Boer, D. Clarke, and E. B. Johnsen. A complete guide to the future. In R. de Nicola, editor, *Proc. 16th European Symposium on Programming (ESOP'07)*, volume 4421 of *LNCS*, pages 316–330. Springer, Mar. 2007.
14. 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.
15. C. C. Din, J. Dovland, and O. Owe. Compositional reasoning about shared futures. In *Software Engineering and Formal Methods - 10th International Conference, SEFM 2012, Thessaloniki, Greece, October 1-5, 2012. Proceedings*, volume 7504 of *LNCS*, pages 94–108. Springer, 2012.
16. C. C. Din and O. Owe. Compositional reasoning about active objects with shared futures. *Formal Aspects of Computing*, pages 1–22, 2014.
17. C. C. Din and O. Owe. A sound and complete reasoning system for asynchronous communication with shared futures. *Journal of Logical and Algebraic Methods in Programming*, 83(56):360 – 383, 2014. The 24th Nordic Workshop on Programming Theory (NWPT 2012).
18. C. C. Din, O. Owe, and R. Bubel. Runtime assertion checking and theorem proving for concurrent and distributed systems. In *MODELSWARD 2014 - Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development, Lisbon, Portugal, 7 - 9 January, 2014*, pages 480–487. SciTePress, 2014.
19. 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.
20. E. Giachino, C. Laneve, and M. Lienhardt. A Framework for Deadlock Detection in ABS. *Software and Systems Modeling*, 2014. To Appear.
21. R. Hähnle, M. Helvensteijn, E. B. Johnsen, M. Lienhardt, D. Sangiorgi, I. Schaefer, and P. Y. H. Wong. HATS abstract behavioral specification: The architectural

view. In B. Beckert, F. Damiani, F. Boer, and M. Bonsangue, editors, *Formal Methods for Components and Objects*, volume 7542 of *LNCS*, pages 109–132. Springer, 2013.
22. C. A. R. Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Prentice Hall, 1985.
23. 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 *LNCS*, pages 142–164. Springer, 2011.
24. E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. *Software and Systems Modeling*, 6(1):35–58, Mar. 2007.
25. E. B. Johnsen, R. Schlatte, and S. L. Tapia Tarifa. Integrating deployment architectures and resource consumption in timed object-oriented models. *Journal of Logical and Algebraic Methods in Programming*, 84(1):67–91, 2015.
26. E. Khamespanah, Z. Sabahi-Kaviani, R. Khosravi, M. Sirjani, and M. Izadi. Timed-rebeca schedulability and deadlock-freedom analysis using floating-time transition system. In *Proceedings of the 2nd edition on Programming systems, languages and applications based on actors, agents, and decentralized control abstractions, AGERE! 2012, October 21-22, 2012, Tucson, Arizona, USA*, pages 23–34. ACM, 2012.
27. 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.
28. B. C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.
29. J. Schäfer and A. Poetzsch-Heffter. JCoBox: Generalizing active objects to concurrent components. In *European Conference on Object-Oriented Programming (ECOOP 2010)*, volume 6183 of *LNCS*, pages 275–299. Springer, June 2010.
30. Z. Sharifi, S. Mohammadi, and M. Sirjani. Comparison of NoC routing algorithm using formal methods. In H. R. Arabnia, H. Ishii, M. Ito, K. Joe, H. Nishikawa, F. G. Tinetti, G. A. Gravvanis, G. Jandieri, and A. M. G. Solo, editors, *Proc. of Parallel and Distributed Processing Techniques and Applications (PDPTA 2013)*, volume 2, pages 474–482. CSREA Press, 2013.
31. Z. Sharifi, M. Mosaffa, S. Mohammadi, and M. Sirjani. Functional and performance analysis of Network-on-Chips using Actor-based modeling and formal verification. *ECEASST*, 66, 2013.
32. 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.
33. M. Sirjani and M. M. Jaghoori. Ten years of analyzing actors: Rebeca experience. In *Formal Modeling: Actors, Open Systems, Biological Systems - Essays Dedicated to Carolyn Talcott on the Occasion of Her 70th Birthday*, volume 7000 of *LNCS*, pages 20–56. Springer, 2011.
34. 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.