Unité Mixte de Recherche 5104 CNRS - INPG - UJF
Centre Equation
2, avenue de VIGNATE
F-38610 GIERES
tel : +33 456 52 03 40
fax : +33 456 52 03 50
http://www-verimag.imag.fr

Towards Certifying Deadlock-freedom of BIP Models

Jan Olaf Blech, Michaël Périn

Technical Report n^o TR-2008-1

September 26, 2008

Towards Certifying Deadlock-freedom of BIP Models

Jan Olaf Blech, Michaël Périn

September 26, 2008

Abstract

Verification and validation techniques have become popular in software and hardware development. They increase the confidence and potentially provide rich feedback on errors. However, with increasing complexity verification and validation techniques are more likely to contain errors themselves. In this paper we address the problem of guaranteeing the correctness of validation work with respect to a formal notion of correctness: We certify the absence of deadlocks in systems. Our certification is based upon an existing tool checking deadlock-freedom of BIP (behavior, interaction, priority) [BBS06] models: D-Finder [BBNS08]. BIP is a language for modelling real-time systems. Certificates are generated each time a BIP model is successfully checked for deadlock absence. They contain a proof script – an algorithm – that describes how to ensure oneself that a BIP model is indeed deadlock-free. Furthermore, they comprise system invariants of the checked BIP models implying deadlock-freedom that are used by the proof script. With the help of such a certificate third party users can ensure themselves of deadlock-freedom of their BIP models without having to trust or even take a look at the deadlock checking tool. In particular our certification methodology comprises the formalization of the notion of deadlock-freedom in the higher-order theorem prover Coq. The use of a higher-order theorem prover allows us to formalize this notion in a human readable way. The formalization of the BIP semantics, models, and their invariants in Coq, and an algorithm that checks whether the notion of deadlock-freedom indeed holds for a given BIP model are part of the methodology, too. The algorithm is instantiated to form more concrete proof scripts that are distributed as parts of our certificates. Apart from presenting the methodology we discuss first experimental results.

Keywords: vertifying properties, deadlock-freedom, BIP, theorem proving, Coq

Reviewers:

Notes:

How to cite this report:

@techreport {TR-2008-1,
title = {Towards Certifying Deadlock-freedom of BIP Models},
authors = {Jan Olaf Blech, Michaël Périn},
institution = {VERIMAG},
number = {TR-2008-1}
}

1 Introduction

Tools to ensure properties of system models and programs have become popular in many application areas. One major goal is to guarantee safety and security properties of the considered system models and programs. However, as these tools become more and more complex it is not always easy to see if they are themselves working correctly. An incorrect tool might state a wrong property about a system or a program. The approach presented in this paper is aimed at guaranteeing that distinct results of such tools are indeed correct. In our case we are looking at the tool D-Finder [BBNS08] that decides deadlock-freedom of BIP models [BBS06]. BIP is a language designed for building real-time systems consisting of heterogeneous components. Deadlock-freedom is especially crucial for systems with complex component interaction.

We use a certifying approach. This means that we generate for each successful run of the D-Finder tool a certificate comprising invariants of the BIP model checked to be deadlock-free and a proof script. Thus, each usage of the D-Finder tool for a particular system is verified after it has been performed. Using this certificate, the BIP models, an easily human understandable formalization of deadlock-freedom, and the Coq theorem prover serving as certificate checker, developers and third party users can ensure themselves of the deadlock-freedom without having to trust or even know D-Finder or its algorithms and implementation.

The automatic certificate checking is implemented using the higher-order theorem prover Coq [The07]. Coq features the ability to formalize semantics of BIP models and the notion of deadlock-freedom in a human readable way. Furthermore, Coq allows us to encode the certificate checking algorithm.

Using this methodology the only parts that have to be trusted to guarantee deadlock-freedom of a BIP model are Coq, our notion of correctness and semantics definition, and the underlying hardware and operating system.

Using certifying techniques has – among others – the following advantages over non certifying verification techniques (see e.g. [Ble08] for a more comprehensive compilation of advantages and disadvantages):

- *Easiness*, we do not need to verify the algorithm and implementation of the D-Finder tool, but only distinct runs.
- *Robustness*, if the implementation of D-Finder changes slightly there is often no need to adapt certificate generation.
- Furthermore, there is no need to give access to the tool and its algorithms to guarantee deadlock-freedom.

The drawbacks comprise the fact that one has to generate certificates and check them which may be a time consuming task [Ble08, BPH07].

1.1 Our Approach

Our methodology for guaranteeing the absence of deadlocks is depicted in Figure 1. It shows the process and infrastructure of certifying deadlock-freedom of BIP models. BIP models are passed to D-Finder, the certificate generation (denoted CertGen) and the Coq theorem prover. The D-Finder tool generates invariants and uses them to decide whether a system is deadlock free or not. The same invariants are presented to the theorem prover as part of the certificate. The certificate comprises these invariants and a proof script that is generated by the certificate generator. The Coq theorem prover uses this proof script to prove that a BIP model is indeed deadlock free. Not shown in the figure are generation mechanisms for creating theorem prover representations of BIP models and invariants.

Higher order-order theorem provers like Coq allow the use of relatively easily human readable specifications. The process of proving properties is, however, much more complicated than in first-order or special purpose theorem provers. High automatation and fast proof searching and checking are usually not primary design goals for higher-order theorem provers. The challenges encountered in this work are highly influenced by these characteristics of higher-order theorem provers.

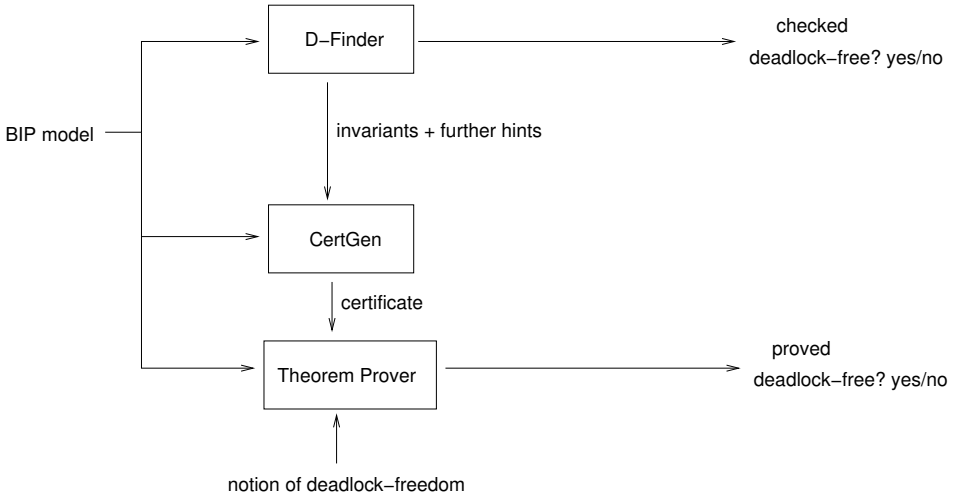

Figure 1: Our Methodology

1.2 Proving Deadlock-freedom

We break the task of verifying deadlock-freedom for a given BIP model BM down into different subtasks as shown in Figure 2. These are combined to prove a human readable formalization of deadlock-freedom correct for a given BIP model. In the figure we use the following definition of enabled states capturing BIP states from which a state transition to a succeeding state is possible:

$$Enabled_{BM}(s) \equiv \exists s'.(s, s') \in [\![BM]\!]_{BIP}$$

The $[\![BM]\!]_{BIP}$ denotes the set of possible transitions of the BIP model BM. Furthermore, we use the following definition of reachable states of a BIP system:

$$ReachableStates_{BM}(s) \equiv$$

$$\left.\begin{array}{l} s = s_{BM\,0} \vee \\ \exists s'.ReachableStates_{BM}(s') \wedge (s', s) \in [\![BM]\!]_{BIP} \end{array}\right\} \textit{smallest fixpoint}$$

It is inductively defined demanding that the initial state $s_{BM\,0}$ is reachable and each state that can be reached from it via transitive state transitions.

The task of verifying deadlock-freedom is refined as shown in Figure 2:

1. The top line in the figure shows our notion of deadlock-freedom for a BIP model. This is what we ultimately want to prove correct. We demand that all reachable states have at least one succeeding state. Thus, there is no reachable state where no succeeding state transition is possible.

2. Instead of proving the first line directly, we can conduct the proof shown in the second line. Moreover, we have to prove that whenever one proves the second line correct the first line is implied. The second line reformulates the notion of enabled states and puts a predicate $\neg DIS_{BM}$ instead. Thus, we may verify that the second line holds for a BIP model BM. In order to show that the first line is indeed implied – guaranteeing the more human readable notion of correctness – we have to show that $\forall s.\neg DIS_{BM}(s) \implies Enabled_{BM}(s)$ holds.

3. The third line introduces an invariant $II(s) \wedge CI(s)$ as a transitive step. This invariant is part of the actual certificate. To use this line in our proofs we have to show that it also implies the first line.

$$\forall s.ReachableStates_{BM}(s) \Longrightarrow Enabled_{BM}(s)$$

$$\Uparrow \quad (\forall s.\neg DIS_{BM}(s) \Longrightarrow Enabled_{BM}(s))$$

$$\forall s.ReachableStates_{BM}(s) \Longrightarrow \neg DIS_{BM}(s)$$

$$\Uparrow \quad \textit{transitivity}$$

$$\forall s.ReachableStates_{BM}(s) \Longrightarrow II(s) \wedge CI(s) \quad \textit{and} \quad \forall s.II(s) \wedge CI(s) \Longrightarrow \neg DIS_{BM}(s)$$

$$\Uparrow$$

$$\forall s.ReachableStates_{BM}(s) \Longrightarrow II(s) \wedge CI(s) \quad \textit{and} \quad \forall s.II(s) \wedge CI(s) \wedge DIS_{BM}(s) \equiv false$$

Figure 2: Verifying Deadlock-freedom: The Meta-Proof

4. The fourth line is a slight reformulation of the third. In order to use it, we have to show that it also implies the first line.

We call the implications between the lines, the meta-proof. Of course, we can use transitivity between lines to conduct this proof. It can be either done once and for all or we can also use a certifying technique to conduct it. However, unlike the meta-proof, the proof of an actual line in the figure demands its instantiation with a concrete BIP model. Hence, a certifying technique as propagated in this paper should be applied.

While some of the tasks to show deadlock-freedom seem to be trivial to us we have identified some tasks that seem especially challenging:

- The derivation of an automatic proof scheme to prove:

  $\forall s.ReachableStates_{BM}(s) \Longrightarrow II(s) \wedge CI(s)$

  in a time efficient way.

- Proving that:

  $\forall s.II(s) \wedge CI(s) \wedge DIS_{BM}(s) \equiv false$

  holds in a time efficient way.

The first item is completely new and captures the correctness of the main task of the D-Finder tool: finding invariants. The work presented in this paper concentrates on the first item. The second item is already done in the D-Finder tool. It does, however, need to be redone in Coq if one wants to keep the trusted computing base small. In this work we concentrate on the first item.

1.3 Related Work

To the authors' knowledge certifying deadlock-freedom with higher-order theorem provers has not been studied before. An early description of certifying techniques and their advantages is given in [BK95]. Certificate checkers for the results of sorting, matrix rank and greatest common divisor computations are presented.

Verification and validation tools generating certificates have been studied on model checkers. The idea first appeared in [Nam01]. A certifying model checker is regarded. The presented technique aims at generating certificates which comprise among other items invariants for use within an ad'hoc proof system. In [Nam03] the methodology is used with lifting proofs between a system and its abstraction. The generation of support sets for a model checker is described in [TC02]. These sets can be used as a

certificate and an algorithm for checking them is given. Unlike in our work the formalization of a human readable notion of correctness is not a goal in these works.

Related to the work presented in this paper is Proof-carrying code [Nec97]. It is a method to certify pieces of code to guarantee that they fulfill distinct properties – typically on access and resource management. A piece of code is given together with a certificate to code users. They can ensure themselves that the desired properties hold by using the certificate. The certificate checker contains 23000 lines of C code. In [HJM+02] a model checker is used as certificate checker. Foundational proof-carrying code [App01] is a variant of proof-carrying code that uses higher-order logic for the formulation of an operational semantics, a small set of axioms and the checking of the certificates. In particular [WAS03] focuses on the problem of keeping both, the checker and the certificate small. Their certificate checker comprises only 803 lines of C code. Like foundational proof carrying code, we want to certify that a property – deadlock-freedom – holds for a system and base this decision on a small set of axioms formalized in a higher-order theorem prover.

Certifying techniques have been studied in the context of compilers. Most notably translation validation [PSS98, ZPG+05, Nec00] provides checkers that check compilation results after each compiler run. In the original work [PSS98], however, these checkers are not using explicit certificates. Furthermore, these approaches are not based on formal semantics or some explicit notion of correctness. However, [TL08] demonstrates a formally verified certificate checker based on formal semantics formalized in Coq. Furthermore, [BPH07] discusses certifying code generation with respect to formal semantics. [BSPH07] ports this approach to verify the transformation of state transition systems.

1.4 Overview

The remainder of this paper is structured as follows: Section 2 presents our formalization of BIP models and their semantics in Coq. In Section 3 we discuss how to verify that invariants appearing in our certificates do hold for distinct BIP models. How this is used to ensure deadlock-freedom is demonstrated in Section 4. We draw a conclusion and present our directions of future work in Section 5.

2 Formalizing BIP Models in Coq

In this section we describe the formalization of BIP models in the language of the Coq theorem prover. We do a shallow embedding of BIP models in Coq. Thus, in contrast to a deep embedding, we do not have a syntactical representation, but use state transition systems to represent a BIP model's semantics. These are directly formalized in Coq.

BIP models are composed of atomic components [BBS06, BBNS08]. Atomic components are state transition systems. They can be composed into larger components. Atomic components communicate via ports with each other. Composed components are state transition systems, too.

Atomic Components

An atomic component can be represented as a state transition system given by a tuple $(L, P, T, V, \{g_\tau\}_{\tau \in T}, \{f_\tau\}_{\tau \in T})$ such that:

- $L = \{l^1, l^2, ..., l^k\}$ is a set of control locations,
- P is a set of ports,
- $T \subseteq L \times P \times L$ is a set of transitions,
- V is a set of variables. It is used by variable valuations: mappings from variables to their values. The type of a variable valuation is denoted X,
- for each $\tau \in T$ there is a guard $g_\tau : X \Rightarrow bool$ and an update function $f_\tau : X \Rightarrow X$. We call a set of transitions TE extended transitions for a set of transitions T if they contain their guard and update functions: $\tau = (l, p, l') \in T$ iff $(l, g_\tau, f_\tau, p, l') \in TE$.

The semantics of an atomic component can also be represented as a state transition system given by a tuple (Q, P, TX) such that:

- Q is a set of states, each has the type $L \times X$.
- P is a set of ports.
- TX is a set of variable valuations including transitions: $((l, x), p, (l', x')) \in TX$ iff $g_\tau(x)$ and $x' = f_\tau(x)$ for some $\tau = (l, p, l')$ and $\tau \in T$.

Atomic components are formalized very close to their mathematical definition (shown here) in Coq.

Composed Components

Atomic components B_i may be composed into bigger components in our case such components are formalized as tuples:

$$(B_1, ..., B_n)$$

They interact via interactions which we formalize in Coq as a set of tuples. Each element in this set represents one possible way of component interaction. It has the following form:

$$(p_1, ..., p_n)$$

Each component p_i in the tuple corresponds to component B_i and contains either a port that needs to be active in the corresponding atomic component in order to do a state transition or it states that the component is not involved in this interaction.

The states encountered by components composed of atomic components have the type:

$$(L \times X) \times ... \times (L \times X)$$

Figure 3 shows the definition of the BIP semantics. Furthermore, the definition of reachable states is presented. The initial state of a BIP model is denoted s_0 in this figure.

Both definitions use n extended state transition relations for atomic components TE_1 ... TE_n, and the set of interactions $interactions$. Furthermore, the definition of the semantics makes use of an initial state.

The semantics of a BIP model is shown first. It is defined by a state transition inference rule. This rule defines the set of possible state transitions for a BIP model BM: $[\![BM]\!]_{BIP}$. A state transition is possible if there is an interaction – the list of active ports – such that there is in each component either a possible state transition labeled with the port or the component is not involved in the interaction. The latter is denoted by port 0 in the interaction. Furthermore, in order to do a transition in an atomic component the appropriate guard functions must evaluate to true. To derive the succeeding states the update functions are performed on the variable valuations of the involved atomic components.

The set of reachable states is defined very similar to the semantics. The first item formalizes that the initial state is always reachable. The second item says that if there is a possible state transition from a reachable state from the semantics definition, this succeeding state is also reachable. The definition is inductive meaning that we take the smallest fixpoint of the set defined by the rules.

Figure 4 shows the definition of reachable states from an initial state s_0 comprising a state transition rule for a BIP model in Coq. We have formalized an independent state transition rule for each number of atomic components in a BIP model. This is due to the fact that we are not able to deal with the l_i, g_i, f_i ,p_i, l'_i variables in a convenient way. Each of them stands for a couple of different variables (destinguished by i) in the definition in Figure 3. In Coq, we mention them explicitly.

Note, that neither the D-Finder tool nor our certifying deadlock-freedom methodology works on the full BIP language. Most notably we have omitted priorities of interactions – which could be added relatively easy – and hierachical composition of components.

Semantics of a BIP model

$$\frac{(p_1, ..., p_n) \in interactions \quad \forall i.((l_i, g_i, f_i, p_i, l'_i) \in TE_i \wedge g_i(x_i) \wedge x'_i = f_i(x_i)) \vee (l_i = l'_i \wedge p_i = 0 \wedge x'_i = x_i)}{(((l_1, x_1), ..., (l_n, x_n)), (p_1, ..., p_n), ((l'_1, x'_1), ..., (l'_n, x'_n))) \in [\![BM]\!]_{BIP}}$$

Reachable States of a BIP model

$$s_0 \in ReachableStates_{BM}(s_0)$$

$$\frac{((l_1, x_1), ..., (l_n, x_n)) \in ReachableStates_{BM}(s_0) \quad (p_1, ..., p_n) \in interactions \quad \forall i.((l_i, g_i, f_i, p_i, l'_i) \in TE_i \wedge g_i(x_i) \wedge x'_i = f_i(x_i)) \vee (l_i = l'_i \wedge p_i = 0 \wedge x'_i = x_i)}{((l'_1, x'_1), ..., (l'_n, x'_n)) \in ReachableStates_{BM}(s_0)}$$

Figure 3: Semantics of BIP models

reachablestates
(s_0: ((L × X) × (L × X)))
(TE$_1$: set (L × (X → bool) × (X → X) × P × L))
...
(TE$_n$: set (L × (X → bool) × (X → X) × P × L))
(interactions: set (P × ... × P)):
(((L × X) × ... × (L × X))) → bool :=

1. reachablestates s_0 T$_1$ ... T$_n$ interactions init
2. ∀ l$_1$ ... l$_n$ x$_1$ ... x$_n$.
   reachablestates s_0 T$_1$ ... T$_n$ interactions ((l$_1$,x$_1$),...,(l$_n$,x$_n$)) ⟶
   ∀ g$_1$ f$_1$ l'$_1$ x'$_1$ p$_1$ ... g$_n$ f$_n$ l'$_n$ x'$_n$ p$_n$.
   (p$_1$,...,p$_n$) ∈ interactions ⟶
   ((l$_1$,g$_1$,f$_1$,p$_1$,l'$_1$) ∈ TE$_1$ ∧ g$_1$ (x$_1$) ∧ x'$_1$ = f$_1$ (x$_1$)) ∨
   (l$_1$ = l'$_1$ ∧ p$_1$ = 0 ∧ x'$_1$ = x$_1$) ⟶
   ...
   ((l$_n$,g$_n$,f$_n$,p$_n$,l'$_n$) ∈ TE$_n$ ∧ g$_n$ (x$_n$) ∧ x'$_n$ = f$_n$ (x$_n$)) ∨
   (l$_n$ = l'$_n$ ∧ p$_n$ = 0 ∧ x'$_n$ = x$_n$) ⟶
   reachablestates s_0 TE$_1$ ... TE$_n$ interactions ((l'$_1$,x'$_1$),...,(l'$_n$,x'$_n$))

Figure 4: Semantics of BIP in Coq

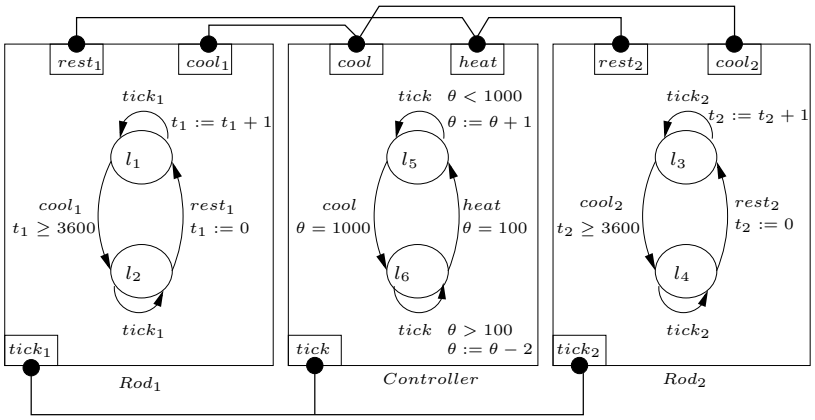

Figure 5: Temperature Control System

An Example

Figure 5 shows a temperature control system [BBNS08, ACH+95] modeled in BIP. It controls the cooling of a reactor by moving two independent control rods. The goal is to keep the temperature between $\theta = 100$ and $\theta = 1000$. When the temperature reaches the maximum value one of the rods has to be used for cooling. The BIP model comprises three atomic components one for each rod and one for the controller. Each contains a state transition system. Transitions are labeled with guard conditions, variable valuation updates, and a port. The components interact via ports thereby realizing cooling, heating, and time elapsing interactions.

3 Verifying Invariants

As described in Section 1.2 the task of showing that an invariant holds for all reachable states of a BIP model is an important part of our methodology to prove deadlock-freedom of a BIP model. In this section we examine a technique that addresses this task.

3.1 The Nature of State Predicates

State predicates defined on BIP models are predicates taking a BIP state (s) and returning a boolean value. Invariants are given as state predicates defined on BIP models. Typical predicates used as invariants on our BIP models have the following form:

$$\bigwedge CI_1(s) \bigwedge ... \bigwedge CI_n(s) \bigwedge II_1(s) \bigwedge ... \bigwedge II_m(s)$$

The CI predicates encapsulate invariants of the different atomic components. The II predicates encapsulate invariant properties of interactions between atomic components. Both, CI and II predicates are themselves made up of disjunctions of properties.

Thus, in our case state predicates are in general made up of multiple predicates – in the scheme below denoted $a_{l\ k}$ – which are already or can easily be grouped into a conjunctive normalform:

$$\bigwedge(a_{1\ 1}(s) \vee ... \vee a_{1\ i}(s)) \bigwedge ... \bigwedge(a_{n\ 1}(s) \vee ... \vee a_{n\ j}(s))$$

Each predicate $a_{l\ k}$ typically encapsulates a statement over a control location or an (in)equation about a variable's value.

If we want to automatically verify that a state predicate in conjunctive normal form as shown above holds for given states in the Coq theorem prover we have to do the following:

- We have to split the conjunctions up into n different subgoals:

$$\begin{array}{l}\mathsf{subgoal\ 1} : a_{1\ 1}(s) \vee ... \vee a_{1\ i}(s) \\ \quad ... \\ \mathsf{subgoal\ n} : a_{n\ 1}(s) \vee ... \vee a_{n\ j}(s)\end{array}$$

- We have to investigate for each subgoal whether an $a_{l\ k}$ element holds.

Each $a_{l\ k}$ itself can be made up of predicates that are grouped into a conjunctive normal form. Thus, verifying a state poperty containing predicates of this form requires to split the subgoals they appear in, into further subsubgoals. This scheme may be applied recursively. Proving whether an $a_{l\ k}$ predicate holds should be easier and especially less time consuming than proving the original state predicate. It can often be done automatically by pre-defined tactics.

3.2 Inductive Invariants

This section describes a method to verify inductive invariants that are presented in conjunctive normal form as shown above. We examine a method to prove that such an invariant holds for a given BIP model. While invariants can be inductive, the set of reachable states of a BIP model is defined using induction, too.

Definition 3.1 ((Inductive) Invariants) *A state predicate ϕ is an inductive invariant of a BIP model BM with initial state s_0 iff*
$s = s_0 \vee ((s_1, s) \in [\![BM]\!]_{BIP} \wedge \phi(s_1)) \longrightarrow \phi(s)$
It is an invariant if there is an inductive invariant ϕ_0 such that
$\phi_0 \longrightarrow \phi$

Thus, for each invariant of a system there is always an inductive invariant that is at least as strong. Furthermore, we like to mention the following well known properties of invariants [BBNS08]:

Proposition 3.1 *Let ϕ_1, ϕ_2 be two invariants of the same BIP model. Then $\phi_1 \wedge \phi_2$ and $\phi_1 \vee \phi_2$ are also invariants of this BIP model.*

This proposition allows the verification of the CI and II predicates from an invariant of the form

$$\bigwedge CI_1(s) \bigwedge ... \bigwedge CI_n(s) \bigwedge II_1(s) \bigwedge ... \bigwedge II_m(s)$$

independantly from each other as long as the CI and II predicates are themselves inductive.

3.2.1 Inductive Invariant Verification Algorithm

The fact that an inductive invariant holds for all reachable states of BIP models can be verified by induction. Thereby we use the induction principle that comes together with the definition of reachable states: A state is reachable iff

1. it is the initial state,
2. or it is the successor of a reachable state.

Using the Coq definition of reachable states such an induction principle is automatically generated by Coq and can be applied within its environment.

Suppose we want to verify that the inductive invariant ϕ holds on all reachable states of a BIP model BM, this means:

$$\forall\ \mathsf{s}.\ \mathit{ReachableStates}_{BM}(\mathsf{s}) \longrightarrow \phi(\mathsf{s})$$

An induction on s using the induction principle from the definition of $\mathit{ReachableStates}_{BM}$ as sketched above turns this lemma into the following subgoals:

1. Initial Case:

$$\phi\ (s_{0\ BM})$$

2. Induction Step:

$$\begin{array}{l}\forall\ s\ s'.\\ \quad \phi\ (s) \longrightarrow\\ \quad (s, s') \in [\![BM]\!]_{BIP} \longrightarrow\\ \quad \phi\ (s')\end{array}$$

The initial case can be verified by using the technique from Section 3.1.

3.2.2 A General Algorithm for the Induction Step

In the step case we make a case distinction on ϕ. Assuming ϕ has the form:

$$\bigwedge(a_{1\ 1}(s) \vee ... \vee a_{1\ i}(s)) \bigwedge ... \bigwedge(a_{n\ 1}(s) \vee ... \vee a_{n\ j}(s))$$

we split the verificatation of the step into different subgoals. Each of these subgoal assume a different possible combination of the disjunct elements $a_{l\ k}$ of ϕ. I.e. each subgoal has the following form:

$$\begin{array}{l}\forall\ s\ s'.\\ (a_{1\ k}(s) \wedge ... \wedge (a_{n\ k'}(s)) \longrightarrow\\ (s, s') \in [\![BM]\!]_{BIP}\\ \qquad\qquad \longrightarrow\\ \phi(s')\end{array}$$

Due to the fact that the number of possible combinations grows roughly exponential with the size of the invariant the number of generated subgoals grows exponentially, too.

To verify such a subgoal we have to make a case distinction on possible state transitions ($(s, s') \in [\![BM]\!]_{BIP}$). This implies the way they modify the state via update functions. Thus, each of the subgoals sketched above is transformed into the following form:

$$\begin{array}{l}\forall\ l_1\ x_1 ... l_n\ x_n.\\ (a_{1\ k}((l_1, x_1), ..., (l_n, x_n)) \wedge ... \wedge (a_{n\ k'}((l_1, x_1), ..., (l_n, x_n))\\ \qquad\qquad \longrightarrow\\ \phi((l'_1, f_{g\ 1}(x_1)), ..., (l'_n, f_{h\ n}(x_n)))\end{array}$$

If a component is not involved in the interaction that lead to the succeeding state the update function is the identity function and the location stays the same.

Matching the appropriate transition rules requires the unfolding of the involved components transition relations and the process of trying to match the unfolded rules against the original states. For larger transition systems this has a notable effect on the theorem prover's time performance.

Each of these subgoals should now be small enough to be solved by Coq with a standard tactic. In the cases examined so far the $a_{l\ k}$ predicates often encapsulate facts on inequalities between systems variables. In such a case Coq's omega tactic has turned out to be adequate.

3.2.3 A Refined Induction Step Verification

To verify that an invariant ϕ holds for a set of reachable states in the Coq theorem prover it is usually more time efficient to split the verification of invariants that have a syntactically large representation into smaller ones. Thus, it is a good choice to perform the following tasks:

- If ϕ consists of a conjunction of invariants CI_i or II_j as sketched above, we verify each of them independently, since smaller invariants are easier automatically verified (see the above mentioned increase of complexity).
- However, one major drawback is the fact that an invariant may not be inductive. In this case we have to find some predicate IP such that $IP \wedge CI_i$ or $IP \wedge II_j$ is inductive respectively.
- The resulting inductive invariants are verified using the algorithm sketched above. Each case in the sketched case distinction should now contain at most two $a_{l\ k}$ predicates: one from the CI_i or II_j predicate, the other one from the IP predicate.

Note, that unlike in model checking the presented technique does not only work for concrete instantiations of states. If we have a concrete state instantiation we can check whether an invariant holds directly by unfolding it and applying some theorem prover tactic. We usually regard states that are universally quantified as sketched above. This procurement allows us to reason about possibly infinite sets of states.

4 Putting it all Together

The different parts of our methodology have to be implemented and put together as suggested in Section 1.1.

In Section 3.1 we proposed the fact that the time the theorem prover needs to decide whether an invariant holds for a given system increases exponentially with the size of the invariant. This is confirmed by our experimental results. Thus, checking whether an invariant holds or not seems to be a bottleneck of our approach. Furthermore, the size of the systems in consideration have an effect on proving the induction step correct. This is due to the fact that we have to determine possible state transitions leading to a succeeding state. The larger these systems are, the more state transition rules have to be considered.

At the moment we are able to prove generated invariants correct for several case studies. Proving some of the invariants from the example in Section 2 correct requires their strengthening to make them inductive. We had to add the following properties to the invariants $t_1 >= 0$, $t_2 >= 0$, and the fact that if we are at location l_6, θ will always be even. D-Finder generates for this example three component and seven interaction invariants (see [BBNS08] for a more detailed description of their generation). Nevertheless, the second part, the proof that the invariants imply deadlock-freedom fails for this example, since it indeed contains a deadlock. The verification of the invariants of this example can be done in Coq within a few minutes on standard machines.

5 Conclusion and Future Work

In this paper we did present a methodology for certifying deadlock-freedom of BIP models using a higher-order theorem prover. The use of a higher-order theorem prover allows us to verify deadlock-freedom with respect to a human readable notion of deadlock-freeness. We did present a Coq formalization of BIP semantics. Furthermore, we identified the main tasks that need to be done during certificate checking and proposed first solutions to conduct these tasks.

Possible directions for future work comprise but are not limited to the following topics:

- Refine and improve the algorithm for inductive invariant verification.
- Establish a technique to make non-inductive invariants inductive by adding additional constraints.
- Implement the other tasks as shown in our meta-proof (cp. Figure 2).
- Improve the Coq formalization of the BIP semantics in a way that it allows the recursive composition of components into larger components.
- Investigate the feasibility of other theorem provers than Coq for the tasks proposed in this paper.

References

[ACH+95] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P. H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138(1):3–34, 1995. 2

[App01] A.W. Appel. Foundational proof-carrying code. *16th Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 247–258, 2001. 1.3

[BBNS08] S. Bensalem, M. Bozga, H. Nguyen, and J. Sifakis. Compositional deadlock detection and verification for component-based systems. In *2nd International Workshop on Verification and Evaluation of Computer and Communication Systems, July 2-3, Leeds, UK*, 2008. (document), 1, 2, 2, 3.2, 4

[BBS06] A. Basu, M. Bozga, and J. Sifakis. Modeling heterogeneous real-time components in bip. In *SEFM*, pages 3–12, 2006. (document), 1, 2

[BK95] M. Blum and S. Kannan. Designing programs that check their work. *Journal of the ACM (JACM)*, 42(1):269–291, 1995. 1.3

[Ble08] J. O. Blech. *Certifying System Translations Using Higher Order Theorem Provers*. PhD thesis, 2008. submitted in April 2008. 1

[BPH07] J. O. Blech and A. Poetzsch-Heffter. A certifying code generation phase. In *Proceedings of the 6th Workshop on Compiler Optimization meets Compiler Verification (COCV 2007), Braga, Portugal*, ENTCS, March 2007. 1, 1.3

[BSPH07] J. O. Blech, I. Schaefer, and A. Poetzsch-Heffter. Translation validation of system abstractions. In *7th Workshop on Runtime Verification (RV'07), Vancouver, Canada*, volume 4839 of *LNCS*. Springer-Verlag, March 2007. 1.3

[HJM+02] T. A. Henzinger, R. Jhala, R. Majumdar, G. C. Necula, G. Sutre, and W. Weimer. Temporal-safety proofs for systems code. In *CAV '02: Proceedings of the 14th International Conference on Computer Aided Verification*, pages 526–538, London, UK, 2002. Springer-Verlag. 1.3

[Nam01] K. S. Namjoshi. Certifying model checkers. In *CAV '01: Proceedings of the 13th International Conference on Computer Aided Verification*, pages 2–13, London, UK, 2001. Springer-Verlag. 1.3

[Nam03] K. S. Namjoshi. Lifting temporal proofs through abstractions. In *VMCAI 2003: Proceedings of the 4th International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 174–188, London, UK, 2003. Springer-Verlag. 1.3

[Nec97] G. C. Necula. Proof-carrying code. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 106–119, New York, NY, USA, 1997. ACM. 1.3

[Nec00] G. C. Necula. Translation validation for an optimizing compiler. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 83–95, 2000. 1.3

[PSS98] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. *Lecture Notes in Computer Science*, 1384:151+, 1998. 1.3

[TC02] Li Tan and R. Cleaveland. Evidence-based model checking. In *CAV '02: Proceedings of the 14th International Conference on Computer Aided Verification*, pages 455–470, London, UK, 2002. Springer-Verlag. 1.3

[The07] The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version 8.1*, 2007. http://coq.inria.fr. 1

[TL08] J-B. Tristan and X. Leroy. Formal Verification of Translation Validators: A Case Study on Instruction Scheduling Optimizations. In *POPL '08: Conference record of the 35rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, New York, NY, USA, 2008. ACM Press. 1.3

[WAS03] D. Wu, A.W. Appel, and A. Stump. Foundational proof checkers with small witnesses. *Proceedings of the 5th ACM SIGPLAN international conference on Principles and practice of declaritive programming*, pages 264–274, 2003. 1.3

[ZPG+05] L. Zuck, A. Pnueli, B. Goldberg, C. Barrett, Y. Fang, and Y. Hu. Translation and Run-Time Validation of Loop Transformations. *Formal Methods in System Design*, 27(3):335–360, 2005. 1.3