Certifying Deadlock-freedom for BIP Models

Jan Olaf Blech and Michaël Périn
Verimag Laboratory, Université de Grenoble, France

Abstract

The BIP framework provides a methodology supported by a tool chain for developing software for embedded systems. The design of a BIP system follows the decomposition in behavior, interaction and priority. The first step comprises the division of desired behavior of a system into components. In a second step interactions and their priorities are added between the components. Finally, machine code is generated from the BIP model. While adding interactions it is possible to overconstrain a system resulting in potential deadlocks. The tool chain crucially depends on an automatic tool, D-Finder, which checks for deadlock-freedom.

This paper reports on guaranteeing the correctness of the verdict of D-Finder. We address the problem of formally proving deadlock-freedom of an embedded system in a way that is comprehensible for third party users and other tools. We propose the automatic generation of certificates for each BIP model declared safe by D-Finder. These certificates comprise a proof of deadlock-freedom of the BIP model which can be checked by an independent checker. We use the Coq theorem prover as certificate checker. Thus, bringing the high level of confidence of a formal proof to the deadlock analysis results.

With the help of certificates one gets a deadlock-freedom guarantee of BIP models without having to trust or even take a look at the deadlock checking tool. The proof of deadlock-freedom fundamentally relies on the computation of invariant properties of the considered BIP model which is carried out by D-Finder and serves as basis for certificate generation. Encapsulating these invariants into certificates and checking them is the most important subtask of our methodology for guaranteeing deadlock-freedom.

1 Introduction

In many areas the development of embedded systems is shifting to software development methodologies working at high levels of abstraction known as Model Driven Engineering. The goal is to develop a model of a system, conduct analyses, perform simplifications and refinement at a high abstraction level and finally automatically generate an implementation. The approach crucially depends on the correctness of the tools used in the development tool chain.

In this paper we present our work on automatic generation and checking of certificates in the form of machine checkable proofs ensuring deadlock-freedom, to increase the reliability of the BIP tool chain used in development of software for embedded systems.

BIP [BBS06] is a software framework designed for building embedded systems consisting of heterogeneous components. It is characterized by three modeling layers: *behavior* of components encoded as transition systems extended with variables, *interactions* between components realized via communication ports and *priority* rules which reduce non-determinism between interactions (BIP stands for Behaviors + Interactions + Priorities). Apart from code generation the BIP tool chain comprises static analyses tools for checking properties like deadlock-freedom. Deadlock-freedom can be difficult to establish and is especially crucial for systems with complex component interaction. Therefore, BIP models are analyzed using the D-Finder [BBSN08] tool to discover potential deadlocks. It does either regard a BIP model as deadlock-free or provide counterexample(s) comprising configurations in which a BIP model might encounter a deadlock. D-Finder consists of a static analyzer discovering invariant properties of components and a model-checker guaranteeing the unreachability of potential deadlock configurations. Among other application areas the BIP language has been successfully applied in the synthesis of robot controllers [BGL+08, BGI+09].

We propose a certifying approach to guarantee deadlock-absence in BIP models. This means that we generate for each successful run indicating deadlock-freedom of D-Finder a certificate – a proof of deadlock-freedom. This proof comprises as a main ingredient a proof that invariants provided by D-Finder for the BIP model hold. Thus, each successful usage of the D-Finder tool for a particular system is verified independently after it has been performed. Using this certificate, the BIP model, an easily human understandable formalization of deadlock-freedom, and the Coq [The07] theorem prover serving as certificate checker, developers and users can ensure themselves of the

deadlock-freedom without having to trust D-Finder, its algorithms and implementation.

For checking certificates we rely on the proof-checker of the higher-order theorem prover Coq. Certificates are proof scripts for Coq. The Coq proof-checker is realized as a type-checking algorithm based on very few typing rules which restricts possible deductions to well-formed mathematical reasoning. Hence, the proof is correct if each proof step respects the typing constraints (see [BD04] for more details).

Coq features the ability to formalize semantics of BIP models and the notion of deadlock-freedom in a human readable way. Our Coq representations of BIP models directly reflect all modeling decisions without containing further abstractions. Thus, it can serve as basis to verify various transformations and analysis results.

Using our methodology the only parts that have to be trusted to guarantee deadlock-freedom of a BIP model are the proof checker of the Coq theorem prover, our notion of deadlock-freedom and semantics definition. Using certifying techniques has – among others – the following advantages over non certifying verification techniques: *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. *Privacy* there is no need to give access to the D-Finder tool and its algorithms to guarantee deadlock-freedom. Hence, the know-how of the tool designers does not have to be revealed.

Compared to algorithm verification the drawbacks of our methodology comprise the fact that we do not have a proof of completeness: there might be BIP models which D-Finder regards as deadlock-free, but we can not prove it.

1.1 Our Approach

An excerpt of the BIP methodology is shown in Figure 1. BIP models are developed using analysis tools including D-Finder. Finally we generate executable code. In principle each development process from a BIP model to a deployable embedded system using the sketched tool chain could be proved correct – including all transformations of BIP models. The correctness proof could be distributed within a certificate. In this paper, however, we restrict ourselves to generating and checking certificates for the analysis results of D-Finder.

Our methodology for generating certificates for D-Finder results guaranteeing the absence of deadlocks is depicted in Figure 2. BIP models are passed to D-Finder, the certificate generation (denoted CertGen) and the Coq theorem prover. The D-Finder tool computes invariants and uses them to decide whether a system is deadlock free or not. 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.

1.2 Proving Deadlock-freedom

Generating and checking D-Finder certificates in a theorem prover means to establish and conduct a formal proof of deadlock-freedom.

We break the task of verifying deadlock-freedom for a given BIP model BM down into different subtasks as shown in Figure 4. The proofs for these subtasks are then composed to prove the top line. 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:

$$Enabled_{BM}(s) \stackrel{def}{=} \exists s'.(s, s') \in [\![BM]\!]_{BIP} \wedge s \neq s'$$

The $[\![BM]\!]_{BIP}$ denotes the set of possible state transitions of the BIP model BM thereby defining its semantics. Furthermore, we use the definition shown in Figure 3 of reachable states of a BIP system BM with s_0 as initial state of BM. It is inductively defined demanding that the initial state is reachable and each state that can be reached from it via transitive state transitions.

The task of verifying deadlock-freedom is performed by using the refinement shown in Figure 4:

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

2. Instead of a direct proof, we can conduct the proof shown in the second line. Furthermore, 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. DIS_{BM} characterizes the states of a BIP model where no further transition is possible. It is provided by D-Finder but can be computed in an easy way using the definition of deadlock-freedom. Thus, we may verify that the second line holds for a BIP model BM. To show that the first line is indeed implied – guaranteeing the more human readable notion of correctness – we have to show that $\forall s.\neg DIS_{BM}(s) \longrightarrow Enabled_{BM}(s)$ holds.

3. The third line introduces as a transitive step invariants provided by D-Finder: II (interaction invariant) and CI (component invariant) which characterize the behavior of a BIP model. These invariants are part of the

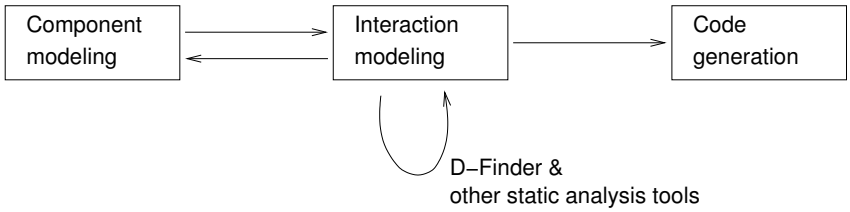

Figure 1. BIP Tool Chain (Excerpt)

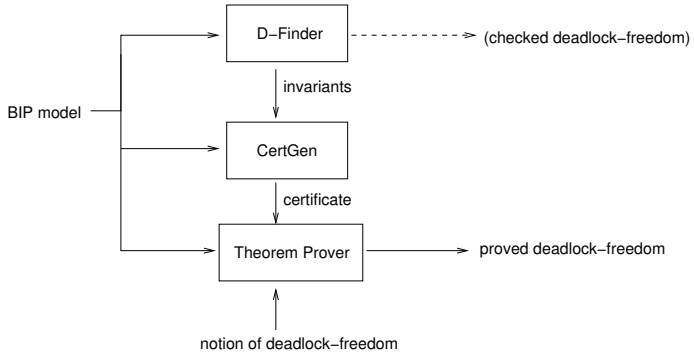

Figure 2. Generating Certificates for D-Finder

$$\left.\begin{array}{l} ReachableStates_{BM}(s) \stackrel{def}{=} \\ \quad s = s_0 \\ \vee\ \exists s'.ReachableStates_{BM}(s') \wedge (s', s) \in [\![BM]\!]_{BIP} \end{array}\right\} \textit{smallest fixpoint}$$

Figure 3. Reachable States Definition

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

   $\uparrow \quad (\forall s.\neg DIS_{BM}(s) \longrightarrow Enabled_{BM}(s))$

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

   $\uparrow$ *transitivity*

3. $(a)\ \forall s.ReachableStates_{BM}(s) \longrightarrow II(s) \wedge CI(s) \quad$ *and* $\quad (b)\ \forall s.II(s) \wedge CI(s) \longrightarrow \neg DIS_{BM}(s)$

Figure 4. Verifying Deadlock-freedom: The Meta-Proof

certificate. To use this line in our proofs we have to show that it also implies the first line.

The decomposition of the proof follows the way D-Finder computes its results. The correctness of that decomposition is verified within our certificates and is not a hard task. Most tasks in this proof scheme are relatively easy from a technical point of view. D-Finder uses a standard SMT-solver to handle the verification of the property $(3b) \quad \forall s. II(s) \wedge CI(s) \longrightarrow \neg DIS_{BM}(s)$ itself. On the other hand D-Finder computes the invariants II and CI using sophisticated algorithms in a way that the property $(3a)$ should hold. However, no *a posteriori* verification is conducted. Thus, the automatic derivation of a proof script to prove that
$(3a) \quad \forall s. ReachableStates_{BM}(s) \longrightarrow II(s) \wedge CI(s)$
really holds in a time efficient way is a challenging subtasks of our methodology. It captures the correctness of the main task of the D-Finder tool: finding invariants. The work presented in the rest of this paper concentrates on this task.

1.3 Related Work

To the authors' knowledge certifying deadlock-freedom with higher-order theorem provers has not been studied before.

Most related to our work is Proof-Carrying Code (PCC) [Nec97], a method to guarantee that executable code fulfills a policy on access and resource management. Executable code is given together with a certificate that is checked before execution. The checker consists of 23000 lines of C code. Foundational PCC [WAS03] uses a small set of axioms and a simpler proof-checker (803 lines of C code). Thus, the size of the trusted computing base is reduced. However, the formalization of the statement to be proved is expressed in pure λ-calculus which is difficult for humans to read.

Furthermore, the translation validation approach [PSS98, ZPFG03] generating certificates for compilation correctness was influential to us. In translation validation the compiler is regarded as a black box with at most minor instrumentation. For each compiler run, source and target program are passed to a separate checking unit comprising an analyzer generating proofs. These proofs are checked with a proof checker. A translation validation approach and implementation for the GNU C compiler is described in [Nec00]. A translation validation checker (called validator) has been formally verified in [TL08]. Compilers generating proof scripts for Isabelle and Coq as certificates have been studied for code generation [BG08, BPH07].

The generation of proofs to certify the verdict of verification tools was first introduced in [Nam01] for a model checker. The certificates are given in an ad'hoc proof-system that is not supported by a proof-checker. Informative certificates (support sets) are used in [TC02]. They contain information about a model-checker's computations. However, support sets can only be checked using a dedicated special purpose tool. In [HJM+02], the BLAST model-checker for C code is extended keeping track of justifications from some simplification steps performed. A complete proof of correctness is not the goal in this work.

In this paper we focus on achieving three important properties of certificates and their checking/proving: 1) human readable specifications, 2) production of complete proofs in a format supported by 3) a trustable proof/certificate-checker (Coq is accepted by some certification authorities).

1.4 Overview

The remainder of this paper is structured as follows: Section 2 presents our formalization of BIP models in Coq. In Section 3 we discuss how to verify invariants appearing in our certificates. Section 4 introduces the certificate generation mechanism. An evaluation is given in Section 5. We draw a conclusion and present our directions of future work in Section 6.

2 Formalizing BIP Models in Coq

In this section we describe the formalization of BIP models in the language of the Coq theorem prover. BIP models are composed of atomic components [BBS06] [BBSN08] that can be composed into larger components. Components are state transition systems. They communicate via ports with each other. An atomic component B_i can be represented by a tuple (L_i, P_i, T_i, V_i) such that

- V_i is a set of variables,
- $L_i = \{l_i^0, l_i^1, l_i^2, ..., l_i^k\}$ is a set of control locations,
- P_i is a set of ports,
- $T_i \subseteq L_i \times (X_i \rightarrow bool) \times (X_i \rightarrow X_i) \times P_i \times L_i$ is a set of transitions, each one comprising a location, a guard function $g : X_i \rightarrow bool$, an update function $f : X_i \rightarrow X_i$, a port, and a succeeding location. The X_i denote variable valuations: mappings from variables V_i to their values.

The atomic components of a BIP model are connected via ports. They communicate via interactions. Thus, a composed component is defined as a tuple $((B_1, ..., B_n), Interactions)$ comprising the atomic components and a fixed set of possible interactions. A single interaction is a tuple $(p_1, \ldots, p_n)$ where p_i is a port of the atomic component B_i or *skip* if B_i is not involved in this

interaction. For consistency reasons (c.p. definition of enabled states $Enabled_{BM}$ in Section 1.2) we require that at least one component is participating in an interaction. Thus, their is no interaction comprising only *skip* values. The state of an atomic component B_i is a tuple (l_i, x_i) comprising a location and variables' valuations. The state of a BIP model is the product of the state of its atomic components: $(L_1 \times X_1) \times \ldots \times (L_n \times X_n)$.

Figure 5 shows the definition of the reachable states of a BIP model for a fixed initial state s_0. The first rule says that the initial state is reachable. The second inference rule captures the entire semantics of BIP. A state transition from a given reachable state is possible if there is an interaction 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. Furthermore, in order to do a transition of 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. Not shown in the definition are priorities of interactions – which can be added relatively easy. The presented definition is the Coq realization of the reachable states definition from Section 1.2. Most importantly it contains the definition of the state transition relation $[\![BM]\!]_{BIP}$.

An Example Figure 6 shows a temperature control system [BBSN08, 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 can be labeled with guard conditions, variable valuation updates, and a port. The components interact via ports thereby realizing cooling, heating, and time elapsing interactions. To give a look and feel for the generated Coq formalization of the example system, the encoding of the set of state transitions in the atomic *Controller* component are shown in Figure 7: a set of tuples, each consisting of a location, a guard function, an update function, a port, and a succeeding location, concatenated by `::` is defined. `fun X => E` is the Coq syntax for a function taking an argument `X` and returning some expression `E`.

Currently neither the D-Finder tool nor our certifying deadlock-freedom methodology works on the full BIP language. Most notably we have omitted hierarchical composition of components.

3 Proving Inductive 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.

Inductive Invariant Verification Invariants of BIP models are given as state predicates that take a BIP state s and return a boolean value. A state predicate ϕ is an invariant of a BIP model iff $\forall s.\ ReachableStates(s) \Rightarrow \phi(s)$. This is proved by performing an induction resulting in the following subgoals that need to be proved:

- $\phi(s_0)$ holds,
- $\forall s, s'.\ \phi(s) \wedge (s, s') \in [\![BM]\!]_{BIP} \longrightarrow \phi(s')$ holds

Typical predicates used as invariants on our BIP models have the following form:

$$\phi \stackrel{def}{=} \bigwedge CI_1(s) \bigwedge \ldots \bigwedge CI_n(s) \bigwedge II_1(s) \bigwedge \ldots \bigwedge II_m(s)$$

The CI and II predicates are made up of disjunctions of basic properties. The CI predicates are component invariants: one for each atomic component. The first of the three component invariants of the example from Figure 6 is the following fact (at_i states that we are at location i):

$$CI_1 \stackrel{def}{=} (at_{l1} \wedge 0 \leq t1) \vee (at_{l2} \wedge 3600 \leq t1)$$

The II predicates capture invariant properties of interactions thereby putting some constraints on the entire system. In the example from Figure 6 the synchronization through interactions makes some component locations incompatible (for instance assuming an initial state of the form $(l_1, \_, l_5, \_, l_3, \_)$, any state $(l_1, \_, l_6, \_, l_3, \_)$ is not reachable). Instead of considering the whole conjunction ϕ, we prove each CI_i, II_i independently, since smaller invariants are easier automatically verified. Note, that component invariants only contain information on the behavior of the corresponding components. For verifying interaction invariants all components involved have to be regarded. Nevertheless, for each component, these invariants only contain information indicating whether an interaction is triggered or not. Thus, for each component involved in an interaction they can contain less information than in the corresponding component invariants. This keeps interaction invariants small and makes the verification feasible. Invariants are verified using a case distinction to cope with the disjunction of basic properties appearing in the invariant. Specialized Coq tactics are applied to verify the different cases.

$$\frac{}{ReachableStates(s_0)} \textit{ where } s_0 \in (L_1 \times X_1) \times \ldots \times (L_n \times X_n)$$

$$\frac{\begin{array}{c} ReachableStates((l_1, x_1), ..., (l_n, x_n)) \quad (p_1, \ldots, p_n) \in Interactions \\ \forall i \in \{1..n\}.\ (l_i, g_i, f_i, p_i, l_i') \in T_i \wedge (g_i(x_i) \wedge x_i' = f_i(x_i)) \vee (l_i = l_i' \wedge p_i = skip \wedge x_i' = x_i) \end{array}}{ReachableStates((l_1', x_1'), ..., (l_n', x_n'))}$$

Figure 5. Definition of Reachable States in Coq

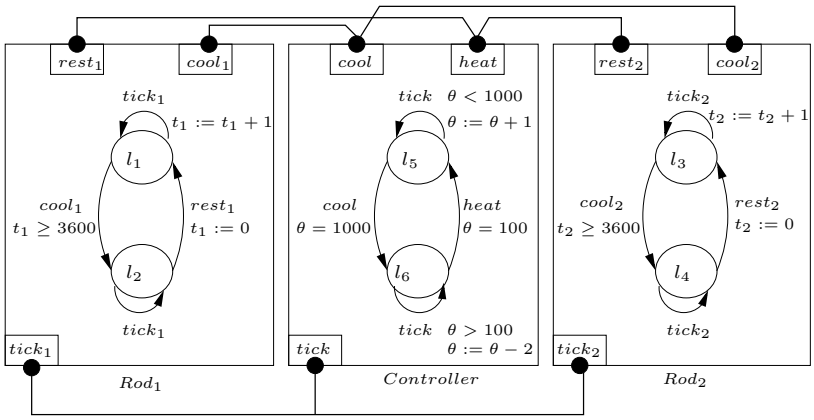

Figure 6. Temperature Control System

```
Definition Controller  : list (L * (X -> Prop) * (X -> X) * P * L) :=
  (L5, fun X => X(theta)=1000, fun X => X, cool, L6)::
  (L5, fun X => X(theta)<1000, fun X => fun v => if v==theta then X(theta)+1 else X(v), tick, L5)::
  (L6, fun X => X(theta)=100 , fun X => X, heat, L5)::
  (L6, fun X => X(theta)>100 , fun X => fun v => if v==theta then X(theta)-2 else X(v), tick, L6)::
  nil.
```

Figure 7. Coq Representation of an Atomic Component

A more detailed version of this algorithm is described in a technical report [BP08].

4 Generating the Certificates

The certificates stating deadlock-freedom of a BIP model are generated for each BIP-model individually. They are provided in a textual representation and formulated in the description and proof language of the Coq theorem prover. The specification part of this language has expressional features similar to those of functional programming languages plus logical formulae. Proof scripts consist of tactic applications. Tactic applications transform proof goals into other proof goals, split proof goals into several subgoals, or solve a prove goal. They can be combined in various ways resulting in new tactics. Most notably are sequential composition and the application of tactics to all subgoals of another tactic.

A certificate, written in the Coq specification and proving language, comprises three parts:

1. A Coq representation of the BIP model,
2. A lemma stating an invariant definition accompanied by a proof script. The proof script algorithmically encapsulates the proof that the invariant does indeed hold,
3. A lemma stating correctness of deadlock-freedom accompanied by a proof script. This proof script references the lemma stating the invariants from above.

Most interesting to this paper is the second item. The proof scripts for it are automatically generated by a small program as shown in Figure 2. It takes the BIP model (more precisely the number of atomic components, their transition rules, possible interactions between components) and the definition of the invariant to be proved correct as arguments. These arguments come with various name and type definitions appearing in the BIP model which are used in the script generation, too. The following parts are generated by the certificate generator denoted CertGen in Figure 2:

1. First we state the actual property to be proved in a lemma: Figure 8 shows the Coq statement to prove for establishing that the property CI_1 stated in Section 2 is indeed an invariant. The definition of the invariant is provided by D-Finder and can be used with minor syntactical modifications for our purposes.
2. The proof starts with an induction on the definition of reachable states. The base case is resolved by a special tactic without any system specific adaptation of the proof script.
3. Based on the structure of the invariant the proof script is generated in a way that it splits the invariant into independently verifiable subgoals.
4. Based on the interactions defined in the BIP model, each subgoal is split again into further subgoals realizing a case distinction on the possible interactions between components.
5. Most crucial in the generation of the proof script is another splitting of subgoals into additional subgoals. This realizes a case distinction on possible transitions. In principle we have to examine every combination of transitions from each atomic component to fulfill the semantics state transition rule (cp. Figure 5 of Section 2). The proof script is generated such that we exploit the previous case distinctions to resolve contradictions in the assumptions as soon as possible to prevent an exponential blowup of cases.
6. Each remaining case represents a very distinct execution step of a BIP model. Showing that the invariant holds afterwords is done by some specialized proof tactics which depend on the class of guards used in the BIP model. In the examples we encountered, guards are arithmetic (in-)equalities and the subgoals are discarded by calls to the Coq inequality and quantifier elimination solver, called omega.

An excerpt of the generated proof script realizing the fifth generation step for the example BIP model from Section 2 is shown in Figure 9. We present it to show that the case distinctions follow the shape of the BIP model and to highlight the regularity of proof scripts. Three blocks for the main part realizing case distinctions for the three components can be distinguished. The `...` contains a number of definition names occurring in component and invariant definitions. Inside the three blocks the case distinctions – triggered by the innermost `destruct` tactic calls – on possible transitions inside the three components takes place: the handling of the four possible transitions can be observed. Each component in a tuple from a transition rule defintion is assigned a name using the tactic `injection`. Since we are dealing with one transition at a time, we reuse the names in the handling of different transitions. At the end of each block, contradictions that have occurred so far are resolved via the `try congruence` tactic.

Note, that there is no need for humans to read the proof script to validate a certificate. To be convinced, a sceptical human evaluator who trusts the Coq proof checker, only has to check the formalization of the BIP semantics, the formalization of deadlock-freedom and the automatic translation of BIP models into their Coq representation. If something declared by D-Finder as invariant of a BIP model turns out not to be invariant, the proof fails. Our certification process

```
Lemma CI1_invariant:
  forall s,
   ReachableStates s -> (at_l1 s /\ 0 <= val_t1 s) \/ (at_l2 s /\ 3600 <= val_t1 s).
```

Figure 8. Generated Proof Script: Stating the Lemma

```
  cbv delta[...]  in I;
  try injection I as  Iport3  Iport2  Iport1 ;
  (destruct C1 as [(C1a,(C1b,x1behavior))|(C1a',(C1b',x1behavior))];
  [
     unfold set_In in C1a;
     unfold  Rod1transrel in C1a; cbv delta[...]  in C1a;
     unfold In in C1a;
     destruct C1a as  [ C1a1 |  [ C1a1 |  [ C1a1 |  [ C1a1 | False ]  ]  ]  ] ;
     [
       injection C1a1 as  succlabel1  port1 f1behavior g1behavior prevlabel1 |
       injection C1a1 as  succlabel1  port1 f1behavior g1behavior prevlabel1 |
       injection C1a1 as  succlabel1  port1 f1behavior g1behavior prevlabel1 |
       injection C1a1 as  succlabel1  port1 f1behavior g1behavior prevlabel1 |
       try contradiction ]
  |
  idtac]);  try congruence;
  (destruct C2 as [(C2a,(C2b,x2behavior))|(C2a',(C2b',x2behavior))];
  [
     unfold set_In in C2a;
     unfold  Rod2transrel in C2a; cbv delta[...]  in C2a;
     unfold In in C2a;
     destruct C2a as  [ C2a1 |  [ C2a1 |  [ C2a1 |  [ C2a1 | False ]  ]  ]  ] ;
     [
       injection C2a1 as  succlabel2  port2 f2behavior g2behavior prevlabel2 |
       injection C2a1 as  succlabel2  port2 f2behavior g2behavior prevlabel2 |
       injection C2a1 as  succlabel2  port2 f2behavior g2behavior prevlabel2 |
       injection C2a1 as  succlabel2  port2 f2behavior g2behavior prevlabel2 |
       try contradiction ]
  |
  idtac]);  try congruence;
  (destruct C3 as [(C3a,(C3b,x3behavior))|(C3a',(C3b',x3behavior))];
  [
     unfold set_In in C3a;
     unfold  Controllertransrel in C3a; cbv delta[...]  in C3a;
     unfold In in C3a;
     destruct C3a as  [ C3a1 |  [ C3a1 |  [ C3a1 |  [ C3a1 | False ]  ]  ]  ] ;
     [
       injection C3a1 as  succlabel3  port3 f3behavior g3behavior prevlabel3 |
       injection C3a1 as  succlabel3  port3 f3behavior g3behavior prevlabel3 |
       injection C3a1 as  succlabel3  port3 f3behavior g3behavior prevlabel3 |
       injection C3a1 as  succlabel3  port3 f3behavior g3behavior prevlabel3 |
       try contradiction ]
  |
  idtac]);  try congruence;
```

Figure 9. Generated Proof Script: Case Distinction on Transitions

revealed some weaknesses of D-Finder such as over simplifications due to external simplifiers used in D-Finder.

5 Evaluation

We have implemented a generator for Coq representations of BIP models in Java using the abstract syntax tree representation provided by the BIP tool chain. The proof script generation is implemented in Ocaml.

Most component and interaction invariants are inductive and can be verified by the sketched principle. However, some approximations are performed by D-Finder that can result in non-inductive invariants. In such cases, we manually strengthen them by adding further constraints thereby making them inductive.

At the moment, we are able to prove the correctness of generated invariants for several case studies including the example in Section 2 and classical textbook examples for potentially deadlock prone systems like different variations of the dining philosophers. Proving some of the invariants from the example correct requires their strengthening to make them inductive. We had to add the following properties to the invariants: $t_1 \geq 0$, $t_2 \geq 0$, and the fact that if we are at location l_6, θ is even. The missing properties where discovered by manually analyzing the feedback of the Coq theorem-prover. D-Finder generates for the example three components and seven interaction invariants (see [BBSN08] for a more detailed description of their generation). The time to verify them takes a few minutes using the proof scripts from our certificates. However, the second part, the proof that the invariants imply deadlock-freedom fails for this example, since it indeed contains a deadlock. Up till now, our work did reveal that some of the generated invariants are not as strong as expected. This leads to the detection of a larger number of potential deadlocks, but is no safety critical error since no BIP model is asserted deadlock-free that does contain deadlocks.

6 Conclusion and Perspectives

In this paper we extended the BIP tool chain such that it generates certificates proving deadlock-freedom of BIP models. We identified the process of proving invariants provided by D-Finder as the main task of certifying deadlock-freedom. We implemented a first solution to conduct this task that consists in automatically generating proof scripts establishing the validity of the verdict provided by D-Finder. We use the Coq higher-order theorem prover as certificate checker. In addition to its high-reliability, it enables a human readable formalization of the notion of deadlock-freedom and BIP semantics.

We implemented a certificate generator that produces proof scripts proving invariants on BIP models which is the most challenging task in certifying D-Finder computations. Furthermore, we implemented the translation of BIP models into Coq representations.

As central contribution of this paper, we showed that it is possible to increase the confidence in a verification tool which is part of a tool chain for the synthesis of embedded systems by generating independently checkable certificates. Directions for future work comprise but are not limited to the following topics:

- Extension of D-Finder, our semantics formalization, and the proof-generator such that they are able to work on hierarchical components and use typed ports between them. Furthermore, the possibility to enrich BIP expressions with C++ code demands the development of Coq tactics to deal with the most commonly used features as a long term goal.
- Implement the other tasks as shown in our meta-proof (cp. Figure 4). For efficient large scale certificate generation and checking this might involve the extension of Yices, a SMT-solver, to produce certificates that are compatible with and can be used inside Coq certificates. This work could be reused in other certificate generating scenarios involving SMT-solvers.
- Non-inductiveness of invariants obstructs their automatic verification. The development of automated techniques to deal with this problem such as strengthening of invariants by adding additional constraints is also a subject we want to address in the future.
- To achieve a fully certifying tool chain including code generation for BIP models we identified the following tasks:
  1. Proving the conformance of the BIP engine to the formal BIP semantics.
  2. Establishing a certifying code generation phase for BIP. This could be a very long term goal. It does not only involve to deal with optimizations appearing during code generation, but also handling language features like multi threading for which to the authors' knowledge up till now no practicably applicable methodology exists.
  3. Proving the race-freedom between components. Race conditions are eliminated using priorities in BIP.

Furthermore, we want to investigate, if we can take advantage of the structure of a BIP model comprising subsystems to improve the structure and checking of certificates.

Acknowledgements Many thanks to Saddek Bensalem, Marius Bozga, Thanh-Hung Nguyen, and Joseph Sifakis for helpful discussions.

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.

[BBSN08] S. Bensalem, M. Bozga, J. Sifakis, and T-H. Nguyen. Compositional Verification for Component-based Systems and Application. ATVA 2008 6th International Symposium on Automated Technology for Verification and Analysis, October 20-23, 2008, Seoul, South Korea.

[BBS06] A. Basu, M. Bozga, and J. Sifakis. Modeling heterogeneous real-time components in BIP. In *SEFM*, pages 3–12, 2006.

[BD04] Y. Bertot and P. Castéran. Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions. Springer-Verlag, 2004.

[BG08] J. O. Blech and B. Grégoire. Certifying code generation with coq. In *Proceedings of the 7th Workshop on Compiler Optimization meets Compiler Verification (COCV 2008), Budapest, Hungary*, ENTCS. April 2008.

[BGI+09] S. Bensalem, M. Gallien, F. Ingrand, I. Kahloul, T-H. Nguyen. Toward a More Dependable Software Architecture for Autonomous Robots. IEEE Robotics and Automation Magazine. *to appear*

[BGL+08] A. Basu, M. Gallien, C. Lesire, T-H. Nguyen, Saddek Bensalem, Felix Ingrand and Joseph Sifakis. Incremental Component-Based Construction and Verfication of a Robotic System. ECAI 2008 The 18th European Conference on Artificial Intelligence, Patras, Greece, July 21 - 25, 2008.

[BP08] J. O. Blech and M. Périn. Towards certifying deadlock-freedom for BIP models. Technical Report TR-2008-1, Verimag, September 2008.

[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.

[HJM+02] T. A. Henzinger, R. Jhala, R. Majumdar, G. C. Necula, G. Sutre, and W. Weimer. Temporal-safety proofs for systems code. *Proc of CAV '02*, 2002. Springer-Verlag.

[Nam01] K. S. Namjoshi. Certifying model checkers. *Proc of CAV '01*, 2001. Springer-Verlag.

[Nec97] G. C. Necula. Proof-carrying code. In *POPL '97*, pages 106–119, New York, NY, USA, 1997. ACM.

[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.

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

[TC02] L. Tan and R. Cleaveland. Evidence-based model checking. *Proc of CAV '02*, London, UK, 2002. Springer-Verlag.

[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 35th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, New York, NY, USA, 2008. ACM Press.

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

[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*, 2003.

[ZPFG03] L. Zuck, A. Pnueli, Y. Fang, and B. Goldberg. VOC: A methodology for the translation validation of optimizingcompilers. *Journal of Universal Computer Science*, 9(3):223–247, March 2003.