An Interactive Verification Tool for Synchronous/Reactive Systems

Manuel Gesell  
TU Kaiserslautern  
gesell@cs.uni-kl.de

Klaus Schneider  
TU Kaiserslautern  
schneider@cs.uni-kl.de

Abstract

Model checking is so far the dominant verification technique for synchronous/reactive systems. However, it is well-known that model checking suffers from an enormous complexity so that only systems of moderate size can be verified this way. Interactive verification methods based on theorem provers are therefore an interesting alternative to verify large systems at different levels of abstraction. However, state-of-the-art theorem provers were designed for general proof problems, so that their use for the interactive verification of synchronous/reactive systems is often too inconvenient for the typical user.

In this paper, we therefore describe the implementation and the main data structures of an interactive tool for the verification of synchronous systems. Due to its specialized application domain, the tool does not aim to replace state-of-the-art theorem provers. Instead, we want to develop a specialized and convenient interactive theorem prover for the verification of synchronous systems. Moreover, since the presented verification tool is tightly integrated with a tool for hardware and software synthesis, many software libraries can be shared between synthesis and verification. In the long term, synthesis can therefore also benefit from verification in that optimizations are possible that are not used today due to too pessimistic estimations.

1. Introduction

Model checking is without doubt one of the success stories of modern computer science [GV08]. However, despite the many improvements made during the last two decades, model checking still suffers from the state space explosion, which means that in the worst case, the runtime for model checking grows exponentially with the size of the system to be verified. For this reason, model checking – even in its restricted versions of bounded and SAT-based model checking – will in general not be applicable to large systems. For this reason, interactive theorem proving has been considered as an interesting alternative since the beginning of formal verification [Gup92, KG99]. Most of the proposed theorem provers are based on higher order logic like HOL [Gor86], Isabelle [Pau94], and PVS [ORS92] and they usually follow the LCF style [GMW79] of theorem provers, i.e., a functional programming language like ML is used as a meta language. Formulas, proof goals, and theorems are implemented as data types in ML, and proof rules are implemented as ML functions mapping existing theorems to new theorems. Since most functional languages offer interactive sessions, one can directly use these for interactive verification.
LCF style theorem provers [GMW79] are safe in that they use only a very small set of deduction rules. For example, HOL [Gor86] relies on five axioms and eight basic deduction rules that are directly implemented on the data structure of theorems. These axioms and basic proof rules refer to the polymorphic $\lambda$-calculus that has been chosen as a foundation of HOL's logic. All other data types like natural numbers, lists, etc. and operators on these types like conjunction and addition are implemented as abbreviations of expressions of the $\lambda$-calculus. For these derived types, hundreds of more convenient proof rules have been implemented on top of the primitive rules.

While this approach makes the theorem provers trustworthy, there is also a price to pay: A huge amount of interaction is often required for a proof; especially adding new data types to specify a proof goal is a complicated process. First of all, one has to implement that data type in terms of existing ones. Additionally, a proof of its nonemptiness and the implementation of useful proof rules for it is required. After many years of experience, we can say that the amount of work that is required for such embeddings is still very large even though there is some limited form of automation [Mel89]. A promising approach is therefore the reduction of interaction by adapting the theorem provers to specific problems. Model checking can thereby be integrated as a particular proof rule [SH99, Gor00]. Although there was some early progress for register-transfer level hardware circuits, e.g., [KSK93], and general reactive systems as e.g., the STeP prover [BBC00], there was little progress on automation for hardware descriptions at higher abstraction levels [Gor95].

Since the system to be verified is typically given in a system description language, one has to embed that language in the theorem prover as well to formulate the verification problem. Several authors considered the embedding of different languages like VHDL [Ree95], ELLA [Bou92], or Verilog [Gor95, Gor98]. However, often only small fragments of the languages were embedded and the verification was never supported by convenient proof rules. Embedding a non-trivial system description language with suitable proof rules turns out to require many years of work. Thus, it is important that we first make sure that we have the right system representation together with a convenient set of proof rules before these are embedded in an existing theorem prover.

In our Averest system (see http://www.averest.org), we use the Esterel-like synchronous language Quartz [Sch09] to describe synchronous/reactive systems [BCE03, BG92]. Using an embedding of the Quartz language in the theorem prover HOL [Sch01, Sch02], we have verified a translation of Quartz programs to synchronous guarded actions [Sch01, Sch02, SBS06, Sch09] that are used as a simple intermediate representation in our Averest system. We are moreover able to formally verify these descriptions by means of model checking, and we can synthesize these systems as hardware or software systems.

As we outlined in [GS12a], difficult problems arise when source code descriptions should be directly used for interactive verification. For this reason, we suggested in [GS12b] to work on the intermediate representation of synchronous guarded actions that allows more flexible proof goal decompositions. We also presented in [GS12b] a preliminary set of proof rules that we used to verify some first examples. However, we still believe that our current proof rules are in a preliminary stage, and therefore it does not yet make sense to start the large effort to embed these in a LCF style theorem prover.

In this paper, we therefore suggest to implement a new kind of interactive verification tool that is tightly integrated with a framework for HW/SW co-design. This allows us not only to reuse the data structures and transformations that are already available in such a framework, but also to use results of formal verification to optimize HW/SW co-design. In particular, we focus in this paper on the implementation and the basic data structures of our interactive theorem prover.
The outline of the paper is as follows: We discuss some basics on synchronous languages and our Averest framework in Section 2. Some basic data structures of our prover will be explained in Section 3. Section 4 illustrates our approach by means of some case studies. Finally, we summarize our work in Section 5.

2. Preliminaries

2.1. Synchronous Programs

Synchronous programming languages [BCE+03] have been developed to describe reactive systems. The computation of a synchronous program is partitioned into macro steps that correspond to interactions between the reactive system and its environment. In each macro step, the reactive system reads new inputs and computes new outputs for the current state as well as a new internal state for the next macro step. Each macro step consists of a finite number of micro steps that are atomic actions like assignments, assumptions or assertions. The syntactic representation of macro steps depends on the particular language. In imperative languages like Esterel [BG92] and Quartz [Sch09], there is a statement called pause that separates one macro step from the next one. The control flow may rest at some of these pause statements and will resume the execution of the micro steps from these locations in the next macro step. The synchronous model of computation further demands that all micro steps are executed in zero-time (i.e. within the same variable environment), and all updates to variables are made synchronously for the next macro step. The execution of micro steps within a macro step must therefore be ordered by their data dependencies since it is required that all variables have a unique value per macro step. Thus, the unique assignment to a variable must be done before the variable is read, and thus, there must be no causality/dependency cycles at runtime.

2.2. Compilation to Synchronous Guarded Actions

Synchronous languages can be compiled to synchronous guarded actions that are a convenient intermediate representation for compilers [Sch09]. Synchronous guarded actions have a very simple structure: In general, a guarded action \( \gamma \Rightarrow \alpha \) consists of a condition \( \gamma \) and an atomic action \( \alpha \) which is in our case either an immediate assignment \( x = \tau \), a delayed assignment \( \text{next}(x) = \tau \), an assumption \( \text{assume}(\varphi) \) or an assertion \( \text{assert}(\varphi) \). The synchronous model of computation then demands that in each macro step, all enabled guarded actions have to be executed and that their execution does not invalidate once enabled guards (so that causality will be respected).

In our previous work, we have developed algorithms to translate synchronous programs to an equivalent set of guarded actions [BS09, Sch09, SBS06]. In addition to the generated guarded actions, the behavior of a system moreover includes for every variable an implicit guarded action which is called the reaction to absence: It defines the value of a variable in case that no action has determined its value in the current macro step (obviously, this is the case iff the guards of all immediate assignments in the current step, and the guards of delayed assignments in the previous step of a variable are false).
2.3. Averest

As a long term project, our group developed a tool called Averest (see http://www.averest.org) for verification and HW/SW-codesign of synchronous Quartz programs. The design flow used in Averest is shown in Figure 1. It contains a compiler that computes for a given synchronous Quartz program an equivalent set of synchronous guarded actions that are stored in AIF files. Since it is possible to separately compile Quartz modules [BS09, Sch09, SBS06] and to link them afterwards, we have to distinguish between modules and linked systems. Modules can be called in other Quartz modules so that one is able to create libraries for later reuse.

There are several transformations available to modify a generated AIF system description. For example, the reduction of compound data types like tuples and arrays to scalar types, reduction to boolean types for hardware synthesis, the aggregation of all guarded actions on one variable into a single guarded action (so that equations are generated), dead code elimination, the generation of an extended finite state machine, and many more are available.

After suitable transformations, AIF systems can be converted to various target languages. For example, there are code generators for software synthesis (producing C, Java or SystemC) or hardware synthesis (producing VHDL and Verilog files). Moreover, a simulator and a code generator for the well-known model checker SMV are also available in the Averest framework.

In this paper, we describe the AIFProver which adds interactive verification rules to the Averest framework. These rules are implemented on top of AIF systems and can therefore directly benefit from the already available data structures for types, expressions, guarded actions etc., as well as from the already available transformations like reduction to boolean types (which allows the use of SAT solvers).

2.4. Example

The source code of a simple Quartz module called ABRO with two specifications is shown in Figure 2. The module has three boolean inputs a, b, r (indicated by ?) and a boolean output o (indicated\(^1\) by !). The module waits on the input signals a and b in parallel, and immediately

\(^1\)Note however that ! means boolean negation outside the interface.
module ABRO(event ?a, ?b, ?r, !o) {
    loop
        abort {
            {wa: await(a); || wb: await(b);}
            emit(o);
            wr: await(r);
        } when(r);
    } satisfies {
        s0: assert A G (o -> a | b);
        s1: assert A G (o -> ! next(o));
    }
}

Figure 2: Quartz Module ABRO

emits output o as soon as the last one of a and b occurred. This behavior is restarted if r occurs. Specification s0 asserts that either a or b must hold, when o is emitted. Property s1 asserts that o cannot hold at two successive points of time.

The ABRO module is compiled to the AIF file shown in Figure 3. Note that the compiler takes care of duplicate use of sub-expressions by abbreviating these into local variables, but to increase readability, we expanded these abbreviations in Figure 3. As can be seen, the guarded actions are separated into control flow and data flow. Control flow actions are the assignments to control flow locations and the data flow consists of the assignments to local and output variables. Note that a new control flow location w0 has been automatically added which is often called the boot location and all control flow variables are false at the initial point of time.

As already mentioned, it is possible to translate synchronous guarded actions into an equation system by aggregating all guarded actions on the same variable. This is useful for theorem proving since one can then use the equation system as rewrite system to prove the specification. The equation system for the ABRO example is shown in Figure 4.

Another important transformation is the computation of an extended finite state machine (EFSM). This description can often be useful for interactive verification in that one can decompose the proof goal with respect to the reachable control states. The EFSM of the ABRO module is shown in Figure 5. Obviously, we can associate every state of the EFSM by the set of active control flow locations in that state. For example, states 1, 3, and 4 represent the three different possibilities for the control flow being in the parallel statement, i.e., either the control flow is in both threads, or one thread already terminated and we are waiting for the other one to terminate.

3. Implementation of AIFProver

In the previous section, we described the basics of the Averest framework. The synthesis framework and the AIFProver share data structures for types, expressions, specifications, and guarded actions as well as available transformations on sets of guarded actions. This does not only avoid reimplementations, it also immediately assures that the prover and the synthesis framework will always remain consistent in case the language would be extended. For this reason, we only have to implement proof goals, theorems and proof rules in the AIFProver. Since Averest has been
**system ABRO:**

**interface:**
- a, b, r: input event bool
- o: output event bool

**locals:**
- w0, wa, wb, wr: label bool

**guarded actions:**

**control flow:**
- True => next(w0) = True
- !w0 => next(wa) = True
- !w0 => next(wb) = True
- !r & wa & !a | r & (wr | wa | wb) => next(wa) = True
- !r & wb & !b | r & (wr | wa | wb) => next(wb) = True
- !r & (wr | a & wa & b & wb | !wa & b & wb | !wb & a & wa) => next(wr) = True

**data flow:**
- !r & (a & wa & b & wb | !wa & b & wb | !wb & a & wa) => o = True

**specifications:**
- s0: A G o --- > a | b
- s1: A G o --- > ! next(o)

---

**Figure 3:** AIF Module ABRO

---

**Figure 4:** Equation System for ABRO

---

implemented in Microsoft’s new programming language F#, we also implemented AIFProver in F#, but due to the .NET framework, any other programming language like C# could have been used as well. The choice of F# was however also made to directly use F# sessions for interactive verification following the spirit of LCF style theorem provers.

Proof rules are therefore F# functions that map existing theorems to new theorems. Also, we implement tactics which decompose desired proof goals into a list of subgoals. Starting with a proof goal, a proof tree is generated by applying a tactic to a leaf of the current proof tree. In our implementation, only the leaves of the current proof tree are stored, and if all leaves were finally proved by a decision procedure, the original root becomes a theorem (i.e. a proved proof goal). Proof rules and tactics correspond to each other and are used in forward proofs (where one derives theorems from axioms by rules) and backward proofs (where proof goals are subsequently decomposed into trivial subgoals), respectively.

We do not list particular proof rules or tactics in this paper, and refer to [GS12b] for a set of proof rules. Instead, we want to discuss their implementation, which was not obvious. The rules listed in
[GS12b] operate at the level of macro steps and have to introduce assumptions and assertions when a proof goal is decomposed into subgoals in order to implement an assume-guarantee deduction system [HQR00]. For example, consider Figure 6: It illustrates the introduction of assumptions and assertions when splitting a sequence $P_1; P_2$ into subgoals. To this end, the corresponding proof rule is given an intermediate specification $\varphi$ that is added by the rule application as assertion for $P_1$ and as assumption for $P_2$. For this reason, there are assumptions and assertions that are added by rule applications and others that were part of the original proof goal.

Proof rules and tactics often refer to states of the EFSM, which turned out to be a good level of abstraction to determine useful decompositions of the system. Thus, the F# type for a proof goal is defined as follows:

```fsharp
type ASM = QName * SpecExpr
type proofGoal = {
    id : string
    sasm : ASM list
    uasm : ASM list
    labels : Set<QName>
    prevStep : GrdAction list
    system : AIFSystem
}
```

The proof goal has an identifier `id` to address different proof goals. Moreover, it contains system defined (sasm) and user defined (uasm) assumptions that can be used for the proof. The EFSM state is encoded by the set of names of the control flow locations that hold in that state. Moreover, we store the set of delayed assignments that were executed in the previous step (and will therefore affect the current state). The core of the proof goal is an AIF system itself! Note that guarded actions do not only consist of assignments, but also of assumptions and assertions, and the proof goal is to prove the assertions believing in the assumptions.
We have to distinguish between the initial step and other steps of a program to deal with the semantics of past temporal operators and the initialization of variables. Moreover, we want to distinguish between proof goals that consider closed systems, and others where an arbitrary environment may be added. The latter is required for a modular verification and differs from closed system verification in that the reaction to absence is left out, and instead variables may have arbitrary values in these cases. Hence, we introduce the following discriminated union:

```fsharp
type ProofGoal = InitClosedGoal of proofGoal | GenClosedGoal of proofGoal | InitModularGoal of proofGoal | GenModularGoal of proofGoal
```

Proved tasks will be stored as theorems, with the type `Thm` that consists of a similar discriminated union:

```fsharp
type Thm = InitClosedThm of proofGoal | GenClosedThm of proofGoal | InitModularThm of proofGoal | GenModularThm of proofGoal
```

An entire proof is represented by the data type `aifproof`. It contains an unmodified copy of the originally considered AIF file (`proofSystem`), a list of all still unproved (`ProofGoals`) and proved (`provedTHM`) subgoals as well as a global list of assumptions (`proofASM`).

```fsharp
type aifproof = {
    proofSystem : AIFSystem;
    ProofGoals : (ProofGoal * ProofGoal list) list;
    proofASM : ASM list;
    provedTHM : Thm list
}
```

Finally, rules and tactics are implemented by the following F# types:

```fsharp
type Rule = Thm list -> Thm
type Tactic = ProofGoal list -> ProofGoal list
```

### 4. Case Studies

To evaluate our approach, we verified some classic sequential algorithms like computing *Fibonacci numbers*, the *extended Euclidean algorithm*, and sorting algorithms like *Bubblesort*. These proofs are analogous to classic proofs using the Hoare calculus and are found in many textbooks. It was however satisfactory to see that we can reproduce these results also with our rules.

We also proved the equivalence of two descriptions of a simple microprocessor. One description is a definition of its instruction set and the other one was a simple non-pipelined hardware implementation\(^2\). The CPUs process 16 bit instruction words on 8 bit data words. Each CPU has eight

\(^2\)See the example section on [http://www.averest.org/examples/Architecture/Abacus/cpus](http://www.averest.org/examples/Architecture/Abacus/cpus)
registers. The proof of the equivalence was very simple, no model-checker was required to this end. Instead, only simple term rewriting was sufficient. During the verification process, we identified several errors in the implementation. First of all, signed and unsigned arithmetic operations were swapped. Then, we identified that the instructions BNZ and BEZ used a wrong register for the comparison. After removing the bugs, the verification process was easily done and could also be used to verify the same processors with other word sizes without changes.

```
ShowCorrectness (fsi.CommandLineArgs.[0]);;
//property s0:
AutoTac();;

//property s1:
RWSWIA o;;
UseVarDefs [wa;wb];;
UseBootFlagDef w0;;
RW_NEXT()
AutoTac()
```

Figure 7: Proof for ABRO

4.1. Verifying the ABRO Example

In this section, a proof for the ABRO module shown in Section 2 is presented (see Figure 7). The proof is initiated by calling function ShowCorrectness(path). This function reads the AIF file and generates an initial proof task for each specification.

The first proof task is simple and is proved by rewriting the specification with the definitions of the variables using one of the most important tactics called AutoTac, which rewrite with immediate assignments, replaces case statements, splits conjunctions in the conclusion, shifts implications in the conclusion to the assumptions and tries to prove ProofGoals before case splits are performed. The proof of specification s1 uses the definition of variable o to rewrite the specification (RWSWIA). After this, the definitions of the variables wa and wb are inserted in the assumptions. Then, the fact that the boot flag w0 holds if another label holds is used. Finally, tactic RW_NEXT is used to shift the next operator inwards to variables so that the obtained goal is proved by AutoTac.

5. Conclusion

We described the implementation of an interactive tool for the verification of synchronous systems that is tightly integrated with the Averest synthesis framework. This allows an efficient reuse of existing code and a lightweight implementation of proof rules so that we can experiment with different sets of proof rules. We have implemented a preliminary set of proof rules [GS12b] and use these rules to verify some benchmark examples. Proofs can be stored as F# scripts so that one can try to rerun the proofs if minor changes were made in the systems or specifications. The next steps are (1) the implementation of proof rules for temporal logic properties and (2) proof rules for modular verification. Other specialized interactive verification systems are the STeP prover [BBC+00] and the recently developed Rodin3 platform [Abr10].

---

3http://handbook.event-b.org
References


