What if Model Checking Must Be Truly Symbolic*

Hardi Hungar
OFFIS, Oldenburg
hungar@informatik.uni-oldenburg.de

Orna Grumberg
The Technion, Haifa
orna@cs.technion.ac.il

Werner Damm
University Oldenburg
damm@informatik.uni-oldenburg.de

Abstract

There are many methodologies whose main concern it is to reduce the complexity of a verification problem to be ultimately able to apply model checking. Here we propose to use a model checking like procedure which operates on a small, truly symbolic description of the model. We do so by exploiting systematically the separation between the (small) control part and the (large) data part of systems which often occurs in practice. By expanding the control part, we get an intermediate description of the system which already allows our symbolic model checking procedure to produce meaningful results but which is still small enough to allow model checking to be performed.

1 Introduction

This paper is about a close marriage of two well known verification paradigms: that of *model checking* and *generation of verification conditions*. There is no need for reiterating the success story of model checking in the verification of reactive systems originating with the seminal paper by Clarke, Emerson and Sistla on CTL model checking [7]; indeed it is safe to say that the combination of (so-called) symbolic techniques [6], abstraction [8] and compositional reasoning [15, 18] have rendered this technology to a state where industrial usage is feasible.

But beyond doubt even those combined approaches are inadequate for a complete verification of the majority of designs. In particular, applications with large or complicated data parts will escape them. We will bring in the generation of verification conditions to overcome some of the limitations.

The story of generation of verification conditions dates back to Floyd's seminal paper [13] from 1967. A large body of research has been conducted over the years on sequential program verification for increasingly more complex programming language constructs [1]. More recently, parallel programming languages [2] have also been extensively investigated. However, the inherent complexity of the task and less stringent commercial need for formally verified software systems has impeded industrial applications of this technology. A few exceptions mainly come from the area of secure systems.

The arguments impeding industrial applications of software verfication do not hold if we look at systems closer to the hardware level. For such systems, the incentive to avoid errors is higher. Moreover, many of them combine data and control in a way that enables simplifying or even automating large parts of the verification.

In this paper we will show a method that avoids some of the difficulties with verification condition generation. We will demonstrate how model checking techniques may be used to reduce automatically first-order temporal logic specifications to simpler verification conditions. These conditions concern either purely sequential behavior of subsystems or first-order data properties. Our procedure is very different from what is usually called "symbolic" model checking, which operates on *codes* for the state sets of the system. Here, we represent data and data operations by first-order formulas and substitutions, similar to their respective representations in the specification logic and the system description language. We called this "truly symbolic" in contrast to the coding approach of "symbolic" model checking.

---

*This research has been funded in part by the MWK under No. 210.3 - 70631 - 99 - 14/93. The views and conclusions contained in this document are those of the authors.

The class of applications we aim at include processors where the data path is simply too wide to be reasonably considered finite state, or embedded control applications, where complex interfacing logic is combined with sometimes nontrivial computations on sampled data (e.g. solving differential equations numerically). These applications have in common, that there is a clear separation between the handling of *control* and *data*. I.e.:

- The pipelined execution of a RISC instruction is solely determined by the instruction type, the pipeline stage, and other state information collected in the controller, which together constitute the *control* part of the design; register contents as well as address fields etc. form the *data* part and are evaluated separately and do influence control only sparsely.

- In embedded control applications it is the control part which governs the interaction between the controller and the controlled system (determining e.g. the sampling rate, strobes, etc.); whenever sampled data are latched into the controller, it initiates the data part of the computation, causing a possibly complex but terminating evaluation.

We find the perfect match for our approach when the data part does not affect control at all. In this case, we show that specifications can be *tested* by conventional model checking on the control part of the system. If the test result is negative, not only the control part of the specification, but also the complete specification involving data is not satisfied by the system. A positive result, on the other hand, tells that the control part of the specification is true in the complete system.

Specifications (and systems) which survive this test phase may then be analyzed more thoroughly. For that, we propose a method that generates first-order verification conditions. This phase does not require a complete separation of control from data. The restriction on their interdependence is more relaxed. Therefore, this phase is applicable also to systems for which the test phase is not.

The procedure we apply is based on a first-order extension of local model checking in the style of [22], using the control information present in the system description to investigate only those first-order aspects of the model consistent with the required behavior of its control part. The first-order verification conditions to be generated appear as success conditions of the model checking procedure. A *sufficient criterion* for the generation to be performed completely automatically is that the control part only allows a bounded number of computations on the data. This criterion subsumes e.g. Wolper's data independence property [23], which forbids any computation on data. Sometimes it is possible to transform a system description which does not meet our criterion to one which does. A loop which computes on data may be replaced by a finite (first-order) representation of its effects. This generates a sequential verification condition which can be treated separately.

Our approach differs from others addressing the verification of first-order temporal logic specifications mainly by exploiting the above separation between control and data to achieve a high degree of automation of the verification process. Also, its scope of application certainly goes beyond what can be done in others.

Approaches based on abstraction [8, 14] and, to some extent, [11] try to reduce the state space to a small resp. finite one, where the proof engineer is required to find suitable abstractions for program variables. In our approach, the verifyer's main involvement is in deciding which variables to consider as control. Remaining are of course first-order and sequential verification conditions. But even these may often be discharged automatically, e.g. if each single data loop can be handled by BDD-techniques after it is extracted from the context of the rest of the system.

More similar results involving data/control separation can be found in [17] where another generalization of Wolper's data independence is pursued. Due to the different system description format used there, separation has a different meaning and thus the results are complementary to ours. However, [17] does not even attempt to cope with data computations, and does not include techniques for first-order verification condition generation.

Verification techniques in the style of [20] which underly e.g. procedures of the STEP system [19] are closer to our approach. Indeed, one could certainly integrate a variant of our generation method as one subprocedure of STEP, suited to deal with a specific class of problems.

Although our techniques and results are rather independent from the overall framework, we chose one particular for their demonstration.

Our specification logic is FO-ACTL, a first-order version of ACTL (which resembles CTL, but allows only universal path quantifiers). The programming language might be thought of as being VHDL, stripped to its semantical essence: a flat parallel composition of sequential processes, which are essentially while-programs extended by one communication construct inspired from VHDL's **wait** statement called **step**. A **step** can only be executed jointly by all processes and thus serves as a *synchronization barrier*; whenever the processes synchronize in a **step**, they exchange information through typed in- resp. outports. All local computations (between steps) work only on local variables.

A program is given as a transition system in which the transitions are annotated by the actions performed between states. Such a program stands for a (possibly infinite-state) Kripke structure, whose states represent the current position in the program and the current variable valuation. Halfway to this large Kripke structure, we have the *control-expanded program*, where only control valuations are explicitly coded into the states and operations on the data variables still annotate the transition symbolically, in the same way as in the original program. This is the structure on which our verification procedures operate.

The test whether a specification is consistent with control of the system is performed by *stripping* the control-expanded program from its data annotations (e.g. turning branches governed by data dependent predicates into nondeterministic choice). This process may introduce nonterminating loops which, if data were considered, would always terminate. In the stripped program, these loops get annotated by fairness constraints ensuring their eventual termination. The validity of a similarly stripped formula will then be evaluated using standard (i.e. propositional) model checking. The data/control separation we require in the original program guarantees that this evaluation approximates validity of the specification in the desired way.

The verification condition generation essentially collects data operations on those paths through the control-expanded model which justify the specification. Besides the sufficient criterion mentioned above which guarantees fully automatic verification condition generation, the procedure works in several other cases as well (which do not seem to have a nice characterization).

The paper is organized as follows. Having developed the programming language and its semantics including the control-expanded program and its stripped version in Section 2, Section 3 defines the logic as well as a stripping operator on formulas, reducing them to their control aspects. Section 4 develops the theory to provide the quick test of validity of a FO-ACTL formula, while the generation of verification conditions is described in Section 5.

A fully formal development of our method would require numerous definitions and constructions, which would be impossible to fit into the available space. So we appeal to the reader's intuition whenever a concept is introduced not rigorously but informally or by example.

2 Semantical Foundation

This section introduces the programming language and its semantics. We treat a toy language vaguely similar to VHDL; any other parallel programming language would serve the purpose of this paper. The main novel notion introduced is that of a *control-expanded* program, which makes the distinction between data and control aspects of a program explicit, thus providing the semantic basis of the subsequent sections.

Programs in our toy language consist of a flat parallel composition $P_1 \| \ldots \| P_r$ of sequential processes. We retain from VHDL that processes communicate over *ports*, which in our toy language almost reduce to read-only variables modelling *in*ports resp. write-only variables modelling *out*ports. In contrast to variables, updates of ports are possible only when executing a *step*-statement discussed below.

Process definitions are of the form

process <*process-declarative-part*> **begin** <*sequential-statement*> **end.**

The process declarative part of a process P defines in particular the sets of its *in*- resp. *outports* I_P resp. O_P, and V_P of *P's local variables*. We require ports and variables to be initialized and omit the index P whenever it is understood from the context. Its body is given by a so-called *sequential statement*, which is executed continuously as if enclosed in a *do forever* loop. We allow, like VHDL, standard statements such as *variable assignments*, *if-, case-,* and *while-* statements, and *sequential composition*. Given an assignment

v:=e, we will call v the *sink* of the assignment. In our toy language we have collapsed *signal assignments* and *wait statements* from VHDL in the *step statement* taking the form

$$\textbf{step(in } v_1, \ldots, v_m \textbf{; out } e_1, \ldots, e_n\textbf{)} \ .$$

A step statement is executed iff all processes are willing to do a step; in this case, P's inports $I_P = \{i_1, \ldots, i_m\}$ are copied into the local variables $v_1, \ldots, v_m$, while its outports $O_P = \{o_1, \ldots, o_n\}$ take values determined by expressions $e_1, \ldots, e_n$. For simplicity, we assume that "wiring" of ports is given by equality of port names, hence the collection of all ports are variables shared between all processes, which are updated only in the disciplined style provided by the *step* statement; in VHDL jargon, this restriction would correspond to using only signal assignments with *delta delay.* We also require that for each port p there is at most one process assigning a value to p.

Our language is strongly typed; for the purpose of this paper we simply assume a collection of types with typical element τ. Example types are **bool, bit, integer, real, bitvector, array**, and enumeration types. At latest at verification time we assume, that types are classified in two *modes*, *data* and *control*, with the obvious restriction that the domain D_τ of expressions of type τ is *finite* whenever τ is of *mode control*. This classification of types induces a classification of ports and variables.

As a simple example, consider the program from Fig. 1. Depending on the value of the boolean input `op`, until the next **step** the program either computes `res:=arg*2` or - by executing a terminating loop - `res:=arg^2`. A typical choice of modes is to consider the inport `op` and the corresponding local variable `c` to be of mode *control*.

```
process small
  in op:  bool := f, arg:  nat := 0
  out res:  nat := 0
  var x,y,z:  nat := 0, c:  bool := f
  begin
    step(in c, x; out z);
    if c
      then z:= x+x
      else y:= x; z:= 0;
        while y>0
        do
          y:= y-1; z:= z+x
        od
    fi
  end
```

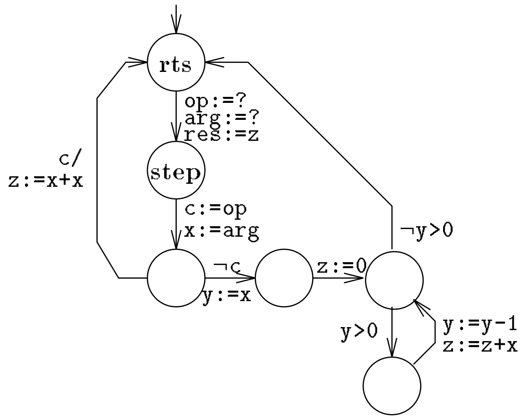

Figure 1: Example program and its flowchart

We use a variant of labeled transition systems as intermediate models for the semantics of our toy language. As a first step, a program is translated into a *flowchart* which represents the flow of control in a graphical format, see again Fig. 1 for an example. States in the flowchart correspond to positions in the program. They are labeled by **rts**, **step** or **none** to indicate whether in that position, the program is willing to engage in a step action, performing a step, or doing neither. To get the second intermediate model, the values of variables and ports of mode *control* get *expanded*: Their values will then be represented explicitly in the states. This results in a structure we call the *control-expanded program*, denoted *CEP*. It is this control-expanded program on which the verification condition generation will operate. Removing the transition labels yields the *stripped CEP* or *SCEP*, which will allow tbe propositional test of specifications. If, instead of removing the transition labels, we expand *all* variables, we get the *fully expanded program*, or *FEP*. The *FEP* is a Kripke structure. Its states include a valuation of all variables and ports, and its transitions are not labeled any more. This Kripke structure is the reference structure for defining the satisfaction relation between first-order temporal logic formulas and processes of our toy language.

For the more formal development, we fix a set of inports I, outports O, and variables V, and abbreviate $V \cup I \cup O$ by Var.

A *labeled symbolic transition system over I, O, V* assumes a classification of each element of Var as either being *expanded* (Var_{exp}) or *symbolic* (Var_{symb}). It is an (ordinary) labeled transition system whose state space consists of *pairs* of so called *control points* from a finite set S and valuations of the expanded variables Var_{exp} collected in the set Γ. Its transitions are labeled by an *enabling condition* on the symbolic variables and a set of assignments to symbolic variables. We use s (resp. γ) as meta variables for control points (resp. valutations of expanded variables). The *initial value* of expanded variables is given by a designated valuation γ_0, while the initial valuation of symbolic variables is given by a set of *initial assignments* A_{init}. The initial control point is designated s_0. The (standard) labeling function of states L assigns to any control point atoms of our logic in the set $\{\mathbf{rts}, \mathbf{step}, \mathbf{none}\}$. Assignments are of the form v:=e s.t. v and all variables occurring in e are *symbolic*. All sinks of assignments occuring in one transition label must be mutually distinct. Moreover we require, that sinks of assignment are local variables, except for transitions originating from control points labeled **rts**, where also assignments to outports are allowed.

Collecting all items into a structure yields an eight-tuple $(S, \Gamma, L, R, Var_{exp}, s_0, \gamma_0, A_{init})$ as constituents of a labeled symbolic transition system M. Flowcharts, *CEP*s and *FEP*s are all instances of symbolic labeled transition systems. So the flowchart in Fig. 1 constitutes an example with $Var_{exp} = \emptyset$ (only that the initial assigmnments "`op := c := f, arg := out := x := z := y := 0`" have been omitted in the picture). In a *CEP*, the expanded variables are those of mode *control*, while in the *FEP*, the set Var_{exp} consists of all variables (i.e. it equals Var and Var_{symb} is empty.

We translate processes of our toy language into flowcharts by induction on the structure of processes. With each statement, we associate a canonically derived flowchart with a unique entry- and exit control-point, which are used in the inductive definition as gluing points. Since the definition is otherwise routine, we only discuss the semantics of the step statement in detail.

The flowchart of **step(in** $v_1, \ldots, v_m$; **out** $e_1, \ldots, e_n$) has three control points s_0, s, s_e labeled **rts**, **step**, **none**, respectively. In s_0 the process is willing to synchronize with its brother processes. If and only if this happens - as modeled in the definition of the product of the transition systems at the end of this section - it will pass to the designated control point s representing the *passage* of the synchronization barrier. The transition from s_0 to s is labeled by *random assignments* for all inports, which *guess* the value produced by some brother process during this synchronization step, as well as a collection of assignemts to its outports with the expressions occuring in the step statement. More formally,

$$tt \; / \; i_1 \; := \; ?, \ldots, \; i_m \; := \; ?, \; o_1 \; := \; e_1, \ldots, \; o_n \; := \; e_n$$

labels the transition connecting s_0 and s. The subsequent postlude transition copies the values received through inports into the local variables specified in the step statement:

$$tt \; / \; v_1 \; := \; i_1, \ldots, \; v_m \; := \; i_m$$

Compound statements are handled trivially by appropriate gluing and possibly introduction of fresh control points, e.g. using fresh s_0, s_e in the semantics of **if** b **then** π_0 **else** π_1 **fi** to relate s_0 with the entry point of π_0 using a transition labeled with b and the entry point of π_1 labeled $\neg b$. The exit points of π_i are linked with the new exit point.

The *flowchart semantics of process P* , $FC[\![P]\!]$, is obtained from the semantics of its *body* by relating its exit point with its entry point and adding as set of initial assignments those canonically induced from P's *process declarative part.*

The semantics of *programs* is given by defining a parallel composition operator on labeled symbolic transition systems capturing VHDL's communication and synchronization semantics. Since synchronization is only required at steps, all transitions except for those relating control-points labeled **rts** with **step** control points can be taken in *any* order, e.g. in an interleaved fashion. Transitions handling the step are taken in *lock step*, replacing random values assigned to inports by those expressions provided by the processes running in parallel. Due to space restrictions, we do not discuss this in detail; the reader might refer to [10] for a full definition of the comparable operator of VHDL.

Let us now turn to the process of *expanding* a labeled symbolic transition system M. Fig. 2 shows an expansion of our example flowgraph.

To the right, the result of expanding the variable `c` and the port `op` in the flowgraph from Fig. 1 is shown. This is the *CEP* belonging to the example program. The picture omits initial assignments and does not contain unreachable states.

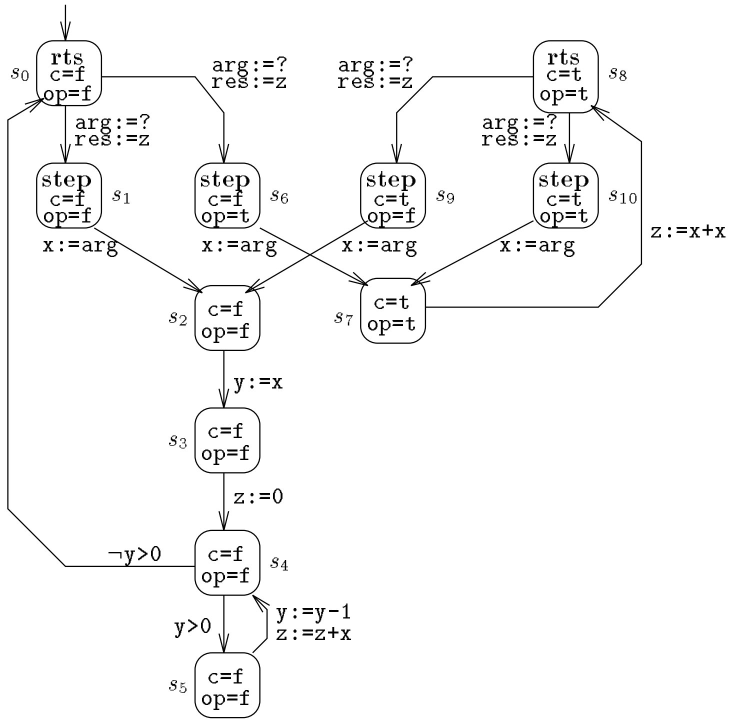

Figure 2: Example *CEP*

Each symbolic variable $v \in Var_{symb}$ can be expanded separately. The expansion of M w.r.t. v is obtained by essentially substituting each occurrence of v in transition labels by its value now represented in the valuation component of states. The only situation deserving special attention arises, whenever v occurs as the sink of an assignment $v := e$ in a transition label. In this case, the assignment is deleted from the transition label. But the query $d = e$ is added to the condition part of the transition leading to a state where v is evaluated to a value d. Expanding the variables and ports of mode *control* in the flowchart $FC[\![P]\!]$ of a program yields its *control-expanded* version, $CEP[\![P]\!]$. Expanding all variables gives the *fully expanded program* $FEP[\![P]\!]$.

By abstracting from data annotations, any labeled symbolic transition structure turns into a classical Kripke structure allowing safe model checking of properties related only to expanded variables and the synchronization atoms, provided the expanded structure is finite. The next section shows that this abstraction, called the *stripped* transition system, enriched by suitable fairness constraints, may in fact be a *precise* abstraction for such formulas under some additional assumptions. The definition of stripping is trivial: for a labeled symbolic transition system M we simply delete all transition labels, thus replacing conditional selection by nondeterminism. We are only interested in the stripped version of the control-expanded program, and will denote this structure by $SCEP[\![P]\!]$.

When the program P is understood from the context, the parameter $[\![P]\!]$ will be omitted and we will simply write *FEP* or *CEP*. For ease of exposition we will assume in the following that all control variables are of type **bool** (instead of an arbitrary finite type).

3 The Logic

The logic FO-ACTL (first-order ACTL) is a *branching-time first-order* temporal logic. It is similar to the propositional temporal logic ACTL (universal CTL) except that it is defined over first-order atomic formulas. Following Emerson [12], a formula in the logic is interpreted over a Kripke structure and an interpretation which is fixed for all states of the Kripke structure.

Similarly to propositional ACTL, FO-ACTL provides only universal path quantifiers. To avoid the invocation of existential path quantifiers via negations, the logic is given in a *positive normal form* in which negations are applied only to atomic formulas. Since only universal path quantifiers are allowed, path quantifiers are left implicit in the syntax. Thus, $\phi\,\mathbf{U}\,\psi$ represents the ACTL formula $\mathbf{A}(\phi\,\mathbf{U}\,\psi)$ and similarly for any other temporal operator.

Definition 1 (FO-ACTL) *Let $\mathcal{L}$ be a first-order language over some signature and let Var be a set of (typed) variables. A formula in our logic is defined inductively as follows:*

1. *Every first-order formula of $\mathcal{L}$ over Var is an atomic formula.*
2. **rts**, **step**, **none** *are atomic formulas.*
3. *If p is an atomic formula then $\neg p$ is a formula.*
4. *If ϕ and ψ are formulas and $x \in Var$ then $\phi \vee \psi$, $\phi \wedge \psi$, $\exists x.\phi$, $\forall x.\phi$ are formulas.*
5. *If ϕ and ψ are formulas then $\mathbf{X}\,\phi$, $\phi\,\mathbf{U}\,\psi$ and $\phi\,\mathbf{W}\,\psi$ are formulas.*

The operator $\mathbf{U}$ is the usual *until*. I.e. $\phi\,\mathbf{U}\,\psi$ requires to eventually reach a state satisfying ψ and not violate ϕ before that event. $\mathbf{W}$ is *weak until* and allows the formula to the left to hold forever.

We use the following abbreviations:

$$\mathbf{F}\,\phi = tt\,\mathbf{U}\,\phi \quad \text{and} \quad \mathbf{G}\,\phi = \phi\,\mathbf{W}\,ff.$$

Let Int be an interpretation for $\mathcal{L}$ over domains D_τ for occurring type τ. The semantics of FO-ACTL formulas is defined with respect to an interpretation Int and a Kripke structure K. For simplicity we denote $T = S \times \Gamma$ and omit the empty set of assignments A_{init} in K. For $t = (s, \gamma)$, with a slight abuse of notation, we use $L(t)$ and $t(v)$ instead of $L(s)$ and $\gamma(v)$. A Kripke structure has now the form $K = (T, L, R, Var, t_0)$. A path in a Kripke structure K is a sequence, $\pi = w_0, w_1, \ldots$, such that for every i, $(w_i, w_{i+1}) \in R$.

$K, Int, t \models \phi$ denotes that the formula ϕ is true in state t of structure K under interpretation Int. If clear from the context, Int is omitted.

We sometimes want to restrict our attention to *fair paths* only, based on some given fairness criterion F that characterizes fair paths. We use $K, t \models_F \phi$ to denote that ϕ holds at t in K with respect to the fair paths only. In particular, the relation $\models_F$ for the temporal operators $\mathbf{X}$, $\mathbf{U}$, and $\mathbf{W}$ is defined with respect to every *fair* path rather than with respect to every path.

In the sequel, we will only consider specifications that do not contain the next-time operator. This operator will be used, however, in the tableau construction in Section 5.

Stripped formulas Given a specification written in FO-ACTL, we extract its propositional part by applying the *strip* operator. The strip operator eliminates all first-order components of the formula, thus results in a propositional ACTL formula. Data-dependent parts of the formula are replaced by *tt*, so the stripped formula will be more often true.

Definition 2 (stripped formula) *Let $Var_c \subseteq Var$ be a set of boolean (control) variables and let ϕ be a FO-ACTL formula, strip(ϕ) with respect to Var_c is defined as follows.*

1. $strip(p(v_1, \ldots, v_k)) = p(v_1, \ldots, v_k)$; $strip(\neg(p(v_1, \ldots, v_k))) = \neg(p(v_1, \ldots, v_k))$ *if* $v_1, \ldots, v_k \in Var_c$.
2. $strip(p(v_1, \ldots, v_k)) = strip(\neg p(v_1, \ldots, v_k)) = tt$ *if some variable* $v_i \notin Var_c$.
3. $strip(l) = l$; $strip(\neg l) = \neg l$ *for* $l \in \{\mathbf{rts}, \mathbf{step}, \mathbf{none}\}$.
4. $strip(\phi \vee \psi) = strip(\phi) \vee strip(\psi)$.
5. $strip(\phi \wedge \psi) = strip(\phi) \wedge strip(\psi)$.
6. $strip(\exists x.\phi) = strip(\phi[tt/x]) \vee strip(\phi[ff/x])$ *for* $x \in Var_c$.
7. $strip(\forall x.\phi) = strip(\phi[tt/x]) \wedge strip(\phi[ff/x])$ *for* $x \in Var_c$.

8. $strip(\exists x.\phi) = strip(\forall x.\phi) = strip(\phi)$, *for* $x \notin Var_c$.

9. $strip(\phi \,\mathbf{U}\, \psi) = strip(\phi)\,\mathbf{U}\,strip(\psi)$.

10. $strip(\phi \,\mathbf{W}\, \psi) = strip(\phi)\,\mathbf{W}\,strip(\psi)$.

Lemma 3 *If ϕ is a FO-ACTL formula then $strip(\phi)$ with respect to Var_c is a propositional ACTL formula over Var_c.*

Example: Consider two specifications for the example in Figure 1, where `op` is a control variable and `arg`, `res` and `x` are data variables. Let $\phi_1 = (\mathbf{F}\,\mathbf{rts})\,\mathbf{W}\,(\mathbf{step} \wedge \neg\mathtt{op})$, then $strip(\phi_1) = \phi_1$.
Consider now the formula $\phi_2 = \forall x.\mathbf{G}\,((\mathbf{step} \wedge \mathtt{arg} = x \wedge \mathtt{op}) \rightarrow \mathbf{F}\,(\mathbf{step} \wedge \mathtt{res} = x * 2))$. Then, $strip(\phi_2) = \mathbf{G}\,((\mathbf{step} \wedge tt \wedge \mathtt{op}) \rightarrow \mathbf{F}\,(\mathbf{step} \wedge tt))$ which is equivalent to $\mathbf{G}\,((\mathbf{step} \wedge \mathtt{op}) \rightarrow \mathbf{F}\,\mathbf{step})$.

4 The Propositional Verification Methodology

In this section, we restrict our concern to programs for which there is a clear separation between data and control. In particular, data cannot influence control variables. For such programs, their verification with respect to first-order temporal specification can take advantage of a preliminary phase in which propositional temporal specifications are proved for the control part of the program.

More precisely, let a *data-dependent condition* be a boolean condition that contains (also) data variables. A program has the *separation property* if no control variable gets assigned a value depending on data, and neither assignments to control variables nor step statements occur in the scope of a data-dependent condition.

The separation property ensures that data do not directly influence control values. But there is a more subtle way in which the validity of a temporal formula not referring to data may be affected: by the termination behavior of data-controlled loops it might be determined whether observable changes to control might happen or not. This influence we eliminate by the assuming - which at least in a hardware context is not unreasonable - that data-controlled loops always terminate. Formally, the assumption enters in the form of fairness constraints.

In more detail, the situation is as follows. Let P have the separation property. Since the transition labels in $CEP[\![P]\!]$ contain no control variables, stripping the CEP from its transition labels eliminates data-dependent conditions only. But the separation property implies that also no control variable changes its value along a transition if the condition labeling it is different from tt. Thus, the stripping does not introduce changes of control which did not happen before. And if the stripping results in an infinite loop that did not occur before, then this must be a *data loop* in which only data variables may change their value. For all these loops, we assume termination and check the stripped formula in the stripped CEP based on this assumption (To complete the verification, we must of course later show that in the fully expanded Kripke structure $FEP[\![P]\!]$ all data loops are indeed terminating). As a result, control properties are not affected by stripping the CEP.

For the verification of formulas which also depend on data, we can conclude the following. If the check of the stripped formula in $SCEP$ (the stripped CEP) returns tt, then we can conclude that the stripped formula is true of $FEP[\![P]\!]$. But if the check returns $f\!f$, then we know that the *original* formula is $f\!f$ on $FEP[\![P]\!]$. As mentioned before, we consider the latter as a significant contribution that enables model checking together with termination proofs to debug any first-order temporal specification.

Our methodology is summarized in the following theorem, where F denotes termination of all data loops. We refer to the well-known notion of a *generalized* Kripke structure [12] to explain the meaning of validity of a temporal logic formula under fairness assumptions.

Theorem 4 *If $FEP[\![P]\!] \models F$ then*

1. $SCEP[\![P]\!] \models_F strip(\phi) \quad\Longrightarrow\quad FEP[\![P]\!] \models strip(\phi)$, *and*

2. $SCEP[\![P]\!] \not\models_F strip(\phi) \quad\Longrightarrow\quad FEP[\![P]\!] \not\models \phi$.

The proof of the theorem could not be included in this paper due to space limitations. The main technical result in the proof states that, if all data loops terminate, then $SCEP[\![P]\!]$ and $FEP[\![P]\!]$ are fair stuttering bisimilar and therefore agree on all propositional ACTL formulas (with no next-time operator).

Example: Consider again the example of Figure 1. Once we verify that the while loop always terminates, we can use $SCEP[\![P]\!]$ to verify propositional ACTL formulas and to refute FO-ACTL formulas. $SCEP[\![P]\!]$ is obtained from the *CEP* of Figure 2 in the appendix by eliminating all transition labels.

For instance, since $SCEP[\![P]\!]$ satisfies $\phi_1 = (\mathbf{F}\,\mathbf{rts})\,\mathbf{W}\,(\mathbf{step} \wedge \neg\mathtt{op})$ under loop termination assumption, we can conclude that this formula is true also in $FEP[\![P]\!]$ (recall that $strip(\phi_1) = \phi_1$).
Consider the FO-ACTL formula $\phi_2 = \forall x.\mathbf{G}\,((\mathbf{step} \wedge \mathtt{arg} = x \wedge \mathtt{op}) \rightarrow \mathbf{F}\,(\mathbf{step} \wedge \mathtt{res} = x * 2))$. Since the formula $strip(\phi_2) = \mathbf{G}\,((\mathbf{step} \wedge \mathtt{op}) \rightarrow \mathbf{F}\,\mathbf{step})$ is true of $SCEP[\![P]\!]$ we can conclude that $strip(\phi_2)$ is true also in $FEP[\![P]\!]$. Note that we cannot conclude that ϕ_2 is true in $FEP[\![P]\!]$. For that we must use the method developed in Section 5.
Consider also the FO-ACTL formula $\phi_3 = \mathbf{G}\,\mathbf{F}\,(\mathbf{step} \wedge \neg\mathtt{op} \wedge y = 0 \wedge z = \mathtt{arg}^2)$. Then $strip(\phi_3) = \mathbf{G}\,\mathbf{F}\,(\mathbf{step} \wedge \neg\mathtt{op} \wedge tt)$. Recall that our formulas have an implicit universal path quantifier accompanied with any temporal operator. Thus, $strip(\phi_3)$ means that for every path, $(\mathbf{step} \wedge \neg\mathtt{op})$ is true for infinitely many states on that path. This does not hold, for instance, on the path $s_0, s_6, s_7, s_8, s_{10}, s_7, \ldots$ in Fig. 2. Hence, $strip(\phi_3)$ is false in $SCEP[\![P]\!]$ and as a result we can conclude that ϕ_3 is false in $FEP[\![P]\!]$.

5 Verification Condition Generation

To handle specifications including data, we propose to verify the temporal aspects relative to first-order verification conditions. As we did before, we start by expanding control variables to get the *CEP*. The key idea then is to use an approach close to what is usually called *local model checking*. Local model checking searches for a sufficient reason for the specification to be satisfied. It has the advantage over iterative model checking that it may turn out that some parts of the program behavior are irrelevant to the specification considered. Here, it may be the case that control information alone can tell that a loop, which can not be handled in general, does not affect validity of the specification. In such and further cases, local model checking will be successful without expanding every data domain. This is essential if some of the data domains are infinite or too large or complex to be completely expanded.

A tableau system for FO-ACTL Local model checking consists in constructing a tableau proving the validity of the formula in question for the start state - or, in the negative case showing the nonexistence of such a tableau. A tableau is essentially a proof tree. Ignoring data for the moment, the root of the tableau is the sequent $s_0 \vdash \phi$, where s_0 is the initial state of the system and ϕ is the formula in question. The successors of each node provide sufficient reason for the validity of that node. Rules are available for each form of node which fix possible successor sets. If the expansion of a tableau is stopped at some point, a success criterion tells whether the tableau constitutes a complete proof for the sequent at its root.

Our tableau system serves as the basic formalism to derive first-order temporal properties involving data, by providing a well-defined method to generate pure first-order conditions from the system and a specification. Below we present our rules for tableaux construction. They differ in two respects from the usual rules for CTL. One is notational: Usually, the different possibilities for proving a sequent (i.e. the different possibilities for successor sets of one vertex) are given in different rules which could be applied alternatively. Our format comprises them in one schema, the alternatives being separated by " | ". Elements in one successor set are separated by " , ". But the rules also reflect that we deal with a first-order model: The state component of a sequent consists of a control state (an element of S) *and* a condition on a variable valuation, given in the form of a first-order formula.

Or Rule

$$\frac{s, p \vdash \phi \vee \psi}{s, p \vdash \phi \mid s, p \vdash \psi}$$

And Rule

$$\frac{s, p \vdash \phi \wedge \psi}{s, p \vdash \phi \;,\; s, p \vdash \psi}$$

Exists Rule

$$\frac{s, p \vdash \exists x.\phi}{s, p \vdash \phi[y/x]}\ ,\ y \notin \mathit{free}(p)$$

Forall Rule

$$\frac{s, p \vdash \forall x.\phi}{s, p \vdash \phi[y/x]}\ ,\ y \notin \mathit{free}(p)$$

Until Rule

$$\frac{s, p \vdash \phi\,\mathbf{U}\,\psi}{s, p \vdash \psi \vee (\phi \wedge \mathbf{X}\,(\phi\,\mathbf{U}\,\psi))}$$

Unless Rule

$$\frac{s, p \vdash \phi\,\mathbf{W}\,\psi}{s, p \vdash \psi \vee (\phi \wedge \mathbf{X}\,(\phi\,\mathbf{W}\,\psi))}$$

Next Rule

$$\frac{s, p \vdash \mathbf{X}\,\phi}{s_1, p_1 \vdash \phi\ ,\ \ldots,\ s_n, p_n \vdash \phi}\ ,\ s \longrightarrow \{s_1, \ldots, s_n\}$$

where $p \wedge c_i \rightarrow \mathrm{subst}(p_i, A_i)$ for $s \xrightarrow{c_i/A_i} s_i,\ i = 1, \ldots, n$.

Case Split Rule

$$\frac{s, p \vdash \phi}{s, p_1 \vdash \phi\ ,\ s, p_2 \vdash \phi}\ ,\ p \rightarrow p_1 \vee p_2$$

$\mathrm{subst}(p_i, A_i)$ in the rule dealing with "$\mathbf{X}$" means the parallel substitution of e for v in p_i for each assignment $v := e \in A_i$. The rules above are chosen to be as simple rules as possible. For convenient application, usually several of them would be combined. For instance, a more useful rule to deal with "$\vee$" like

$$\frac{s, p \vdash \phi \vee \psi}{s, p_1 \vdash \phi \mid s, p_2 \vdash \psi}\ ,\ p \rightarrow p_1 \vee p_2$$

is derived from our rule set by combining the Case Split and the Or Rule.

The reader may have noted that $\exists$ and $\forall$ as well as $\mathbf{U}$ and $\mathbf{W}$ are treated in the same way by the rules. The difference between the operators is captured by (global) success conditions, see below.

A *tableau* is a finite tree of sequents $s, p \vdash \phi$ where the set of successors of each internal node are instances of one of the alternative successor sets according to the rules. The nodes on the path from the root of the tableau to a given node are called its predecessors.

To each tableau we associate a first-order formula which specifies whether the tableau is *successful*. This *success formula* is computed bottom-up. The success formulas of leaves are as follows.

- $p \rightarrow \bigvee_i p_i$ for leaves $s, p \vdash \phi\,\mathbf{W}\,\psi$, where $p_i,\ i = 1, \ldots, n$ are the first-order conditions in predecessors of the form $s, p' \vdash \phi\,\mathbf{W}\,\psi$,
- $p \rightarrow r_s$ for leaves $s, p \vdash r$ with a first-order formula r (see below for the computation of r_s), and
- $p \rightarrow \mathit{ff}$ for other leaves $s, p \vdash \phi$.

For a first-order formula r and a state s, replace the atoms **rts** and **step** as well as control variables in r by their truth values in the state s to obtain the formula r_s.

At inner nodes, the success formula is computed by conjuncting the success formulas of the subtableaux following it. If case split is applied, the appropriate implication is added. At quantifier steps, the respective quantifier is applied.

A tableau is *successful* in a data domain, if its success formula is valid in the domain. A sequent is *provable* if it has a successful tableau. A formula ϕ is provable if $s_0, \mathit{tt} \vdash \phi$ is a provable sequent.

Theorem 5 (Soundness) *The tableau system is sound. I.e., if a sequent $s, p \vdash \phi$ is provable, then all copies of s in the full model where the data variable valuation satisfies p have property ϕ. If a formula is provable, it is valid in the system.*

The tableau system does not provide us with a decision method, though. One reason is that of course the validity of success formulas can not be decided in general. Another one concerns the treatment of the

$\mathbf{U}$ operator. To achieve a stronger form of completeness than we do, we would have to allow a successful recurrence of $\mathbf{U}$-formulas in the style of the recurrence condition for minimal fixpoints of [4, 3]. This, however, would introduce a new dimension of undecidability, because successful $\mathbf{U}$-recurrence would have to involve a well-foundedness condition. We do not strive for completeness in general, though. We do achieve completeness and even decidability relative to first-order questions for a certain class of interesting cases, as indicated by the results below.

The construction of a generic tableau Roughly spoken, systematic tableau construction will provide a proof or a refutation (up to first-order verification conditions) if all "nontrivial cycles" are "broken by control". This is a property of system and formula combined. A "nontrivial cycle" occurs when a data variable value at one position in the program may result by applying a function other than identity to the value the same variable had at that same location at an earlier stage of the execution of a program.[1] Such cycles may cause unbounded expansion of the tableau during construction. A cycle like that one is "broken by control", if one can tell from control information that there is a bound on the number of iterations through this cycle which are necessary to decide the validity of the formula. As an extreme case, the path through the program which introduces the cyclic dependency might not be executable at all without violating an essential control condition in the formula, giving zero as a bound.

A formalization of this informal concept will take several steps. First of these is the construction of a *generic tableau* which comprises in some sense all tableaux which can be constructed for a given formula ϕ. It represents, essentially, the control part of each first-order tableau. Thus, it can later be used to detect cycles broken by control. The rules for the generic tableau are derived from the above rules essentially by removing all first-order aspects.

$$\frac{s \vdash \phi \vee \psi}{s \vdash \phi \mid s \vdash \psi} \qquad \frac{s \vdash \phi \wedge \psi}{s \vdash \phi \,,\, s \vdash \psi}$$

$$\frac{s \vdash \exists x.\phi}{s \vdash \phi[\mathit{ff}/x] \mid s \vdash \phi[\mathit{tt}/x]}\ ,\ x \in V_c \qquad \frac{s \vdash \forall x.\phi}{s \vdash \phi[\mathit{ff}/x] \,,\, s \vdash \phi[\mathit{tt}/x]}\ ,\ x \in V_c \qquad \frac{s \vdash \exists/\forall x.\phi}{s \vdash \phi}\ ,\ x \notin V_c$$

$$\frac{s \vdash \phi \,\mathbf{U}\,/\,\mathbf{W}\, \psi}{s \vdash \psi \mid s \vdash \phi \,,\, s \vdash \mathbf{X}\,(\phi \,\mathbf{U}\,/\,\mathbf{W}\, \psi)} \qquad \frac{s \vdash \mathbf{X}\,\phi}{s_1 \vdash \phi \,, \ldots ,\, s_n \vdash \phi}\ ,\ s \longrightarrow \{s_1, \ldots, s_n\}$$

With these rules, we construct the generic tableau for a given *CEP* and a temporal formula by the following deterministic procedure. Starting with $s_0 \vdash \phi$, the appropriate rule gets applied. But different from the first-order tableau, no choice is made between alternative successors. Instead, all alternatives are pursued. The expansion of the generic tableau stops if the temporal formula is reduced to a pure first-order formula (*first-order leaf*) or if a node recurs, i.e. at a node which has a predecessor labeled by the same sequent (*recurring leaf* resp. *recurrence node*). Since there is a finite number of states and subformulas, the process is bound to terminate.

Next, irrelevant branches are removed. This starts at non-recurring leaves. $\mathbf{X}$-leaves can be replaced by $s \vdash \mathit{tt}$. Also, some of the first-order leaves $s \vdash p$ can be evaluated. To do this, first the formula p_s is constructed. Then, the control information present in p_s is used to determine whether by propositional reasoning and trivial first-order identities like $(\exists x.\mathit{ff}) \rightarrow \mathit{ff}$ the formula can be reduced to tt or ff.

Then, tt and ff are propagated upwards in the tableau. A successor set gets replaced by ff (resp. tt) if one (resp. all) of its components becomes ff (resp. tt). If one of the alternative successor sets of a node becomes tt, the node itself is replaced by tt, and if all alternatives become ff, it is replaced by ff. The resulting, reduced structure is called the *generic tableau* for ϕ.

[1] More general, the value need not be computed from the previous value alone, but also other variables might influence the result.

Observation 6 *For every system and formula, there is one (unique) generic tableau.*

Let us return to our example program from Fig. 1, and take $\phi_1 = (\mathbf{F}\,\mathbf{rts})\,\mathbf{W}\,(\mathbf{step} \wedge \neg\mathtt{op})$ as a specification. Fig. 3 shows the first steps of the construction of the generic tableau (indicating the evaluation of first-order leaves in boxes) and the final result, after removing irrelevant branches. The generic tableau contains one pair of a recurring leaf and recurrence node. These are marked with "$\bullet$". Note that other recurrences (e.g. of sequences involving $\mathbf{F}\,\mathbf{rts}$) occurring during its construction have been eliminated by the reduction process.

$$\dfrac{s_0 \vdash (\mathbf{F}\,\mathbf{rts})\,\mathbf{W}\,(\mathbf{step} \wedge \neg\mathtt{op})}{\dfrac{s_0 \vdash \mathbf{F}\,\mathbf{rts}}{\begin{array}{c} s_0 \vdash \mathbf{rts} \\ \boxed{\mathit{tt}} \end{array} \mid \begin{array}{c} s_0 \vdash \mathbf{X}\,\mathbf{F}\,\mathbf{rts} \\ \vdots \end{array}} \quad , \quad \begin{array}{c} s_0 \vdash \mathbf{X}\,((\mathbf{F}\,\mathbf{rts})\,\mathbf{W}\,(\mathbf{step} \wedge \neg\mathtt{op})) \\ \vdots \end{array} \mid \begin{array}{c} s_0 \vdash \mathbf{step} \wedge op = \mathtt{f} \\ \boxed{\mathit{ff}} \end{array}}$$

$$\dfrac{s_0 \vdash (\mathbf{F}\,\mathbf{rts})\,\mathbf{W}\,(\mathbf{step} \wedge \neg\mathtt{op})}{s_0 \vdash \mathit{tt} \;,\; \dfrac{s_0 \vdash \mathbf{X}\,\phi_1}{s_1 \vdash \mathit{tt} \;,\; \dfrac{s_6 \vdash \phi_1}{s_6 \vdash \mathit{tt} \;,\; \dfrac{s_6 \vdash \mathbf{X}\,\phi_1}{\dfrac{\bullet\, s_7 \vdash \phi_1}{s_7 \vdash \mathit{tt} \;,\; \dfrac{s_7 \vdash \mathbf{X}\,\phi_1}{\dfrac{s_8 \vdash \phi_1}{s_8 \vdash \mathit{tt} \;,\; \dfrac{s_8 \vdash \mathbf{X}\,\phi_1}{\dfrac{s_9 \vdash \phi_1}{s_9 \vdash \mathit{tt}} \;,\; \dfrac{s_{10} \vdash \phi_1}{s_{10} \vdash \mathit{tt} \;,\; \dfrac{s_{10} \vdash \mathbf{X}\,\phi_1}{\bullet\, s_7 \vdash \phi_1} \mid s_{10} \vdash \mathit{ff}}} \mid s_8 \vdash \mathit{ff}}} \mid s_7 \vdash \mathit{ff}}} \mid s_6 \vdash \mathit{ff}}} \mid s_0 \vdash \mathit{ff}}$$

Figure 3: Constructing the generic tableau

If the program has the separation property, the construction of the generic tableau can profit from the results of the test computation according to Theorem 4. They enable early detection of irrelevant or always successful branches.

Instantiating the generic tableau The relevance of the generic tableau construction relies on the fact that every successful tableau can be put in a form that it is an *instance* of the generic one. Instances are built by adding first-order formulas to the state components of sequents and perhaps by unfolding the generic tableau at its recurring leaves.

To be more precise, a first-order tableau T with root $s, p \vdash \phi$ is an instance of a subtableau (to get an inductive condition) of the generic tableau starting at node n if:

- n has the form $s \vdash \phi$, and
- if n is not a leaf, the rule applied to n is matched by an appropriate rule combination in T, and subtableaux starting at end nodes of the rule combination are instances of the corresponding end nodes of the generic rule ("Matching" requires choosing among the alternatives present in the generic tableau, and we allow the matching combination to contain applications of Case Split). And
- if n is a recurring leaf and T is not a leaf itself, it is an instance of the subtableau starting at the recurrence node. And
- if $n = s \vdash \mathit{tt}$ where this is the result of a reduction, T is an instance of the subtableau reduced to n.

The restrictions imposed on a tableau to be an instance of the generic tableau are rather modest. They require complete case distinction for control values, and that branches which are always successful (and have been reduced to tt in the generic tableau construction) have to be chosen. So we have:

Observation 7 *If a formula is provable at all, it is also proved by an instance of its generic tableau.*

Now we give a procedure which tries systematically to construct an instance of the generic tableau. It will not terminate in general. The procedure operates on the generic tableau. It computes a first-order formula, called *instantiating formula*, for each node of the generic tableau. These formulas can subsequently be used to generate an instance.

First-order leaves $s \vdash p$ are instantiated with p_s. Recurring $\mathbf{U}$ -leaves are initialized with *ff* and recurring $\mathbf{W}$ -leaves with *tt*. For inner nodes, the instantiating formulas are computed from those for their successor nodes. Disjunction is used for $\vee$, conjunction for $\wedge$, existential quantification for $\exists$, and universal quantification for $\forall$. For a $\mathbf{X}$ -node with successor formulas $p_1, \ldots, p_n$, the conjunction over $c_i \rightarrow \text{subst}(p_i, A_i)$ is taken. Inner $\mathbf{U}$ - and $\mathbf{W}$ -nodes get instantiated with their successor formulas. But if such a node is a recurrence node, the process of computing the instantiating formula is iterated after instantiating the corresponding recurring leaves with the formula computed for the recurrence node. The iteration stops if a fixpoint is reached for a recurrence node. Propositional and control reasoning is applied to detect a fixpoint.

Although this process does not literally generate an instance of the generic tableau, it performs all necessary computations. Due to lack of space we can not show the formal construction of the instance. One point to note is that the iteration steps at $\mathbf{U}$ -nodes during the computation process correspond to unfoldings in the construction. Most importantly, we can prove that the result of a terminating instantiation provides us with a first-order characterization of the correctness of the program.

Theorem 8 *If the instantiation process terminates for a specification ϕ, the success formula of the generated instance characterizes validity of ϕ. I.e. the success formula is valid in a data domain iff under this interpretation the specification ϕ is valid (for the system).*

In our example in Fig. 3, data do not matter at all. A successful tableau can be derived directly from the generic tableau. One only has to restore branches which have been reduced to the form $s \vdash tt$. As an example for a nontrivial, but still terminating instantiation process the reader may consider the specification $\forall x.\mathbf{G}\,((\mathbf{step} \wedge \mathtt{arg} = x \wedge \mathtt{op} = \mathtt{t}) \rightarrow \mathbf{F}\,(\mathbf{step} \wedge \mathtt{res} = x * 2))$. We have to leave the development of this example to the reader.

The formulas computed for recurrence nodes form chains of monotonically weaker ($\mathbf{U}$) resp. stronger ($\mathbf{W}$) approximations of the strongest resp. weakest fixpoint formula. For infinite data domains, this process need not come to an end, or the end, if reached, need not be detected. Below we will formalize the notion of "cycles broken by control" by a criterion sufficient for the termination of the instantiation.

Termination of the instantiation The termination criterion is based on an annotation of the generic tableau with variable sets. Basically, one just takes the sets of free variables of the instantiating formulas which would be computed by the process sketched above. But it is not necessary to compute the formulas themselves. Instead, one can operate on the finite domain of sets of variables involved (namely, the data variables of the program and the bound variables of the formula) where termination is guaranteed.

The case of next nodes may serve as an example of how these sets are computed. If x annotates the ith successor node of a next node in the CEP, and $x := e \in A_i$, all variables in e annotate the next node. Additionally we take the variables from c_i.

On the completed annotation sets, we draw edges indicating for each variable which other annotations caused its introduction. E.g. if x annotates the ith successor node of a next node, and e gets assigned to x along the edge, all variables in e have an edge pointing to x. Edges always go from inner node annotations to their successor annotations and from recurring leaves to recurrence nodes. Edges originating at next nodes which arise from some $x := e$ where e contains a function application get marked. Let us call the generic tableau *cycle-free* if there is no cycle in the resulting graph contains a marked edge.

Theorem 9 *The instantiation of the generic tableau of a formula terminates if the tableau is cycle-free.*

Critical points for termination of the instantiation are the fixpoint computations at recurrence nodes. During a fixpint computation, only substitutions and boolean operations are applied. If the generic tableau is cycle-free, only a finite number of terms will occur in those computations. Since only finitely many propositionally nonequivalent formulas can be constructed with finitely many terms, fixpoints will be reached and detected.

The condition on the annotations of the generic tableau can be viewed as describing a set of specifications having a finite reason in every data domain. It gives rise to a proof procedure which subsumes properly everything which can be gained by *data independence* reasoning [23]. A program is said to be data independent if, intuitively, its behavior does not depend on the identity of input values (changes to input values lead to similar changes of output values).

Any program which meets appropriate syntactic criteria on its data ports[2] will have only cycle-free generic tableaux, regardless of the formula. On the other hand, there are programs with cycle-free tableaux which perform a control-bounded number of computations and also tests on their data and which are thus not data independent.

This becomes clear if we draw a *value flow graph* of the CEP, similar to the graph on the annotations of the generic tableau. I.e. we annotate each state with the full set of data variables and draw edges and marked edges between variables annotating successive nodes according to the value flow. Transferring the notion of cycle-freeness to value flow graphs, we get a class of programs which will have only cycle-free tableaux.

Proposition 10 *If the value flow graph of a program is cycle-free, then each generic tableau built on its CEP is cycle-free.*

The proposition is implied by the observation that cycles in the generic tableau come from cycles in the *CEP*. This criterion is not necessary, but close to. It should be kept in mind, though, that the automatic instantiation process works in far more cases than just for programs having cycle-free value flow graphs. To decide specific properties, it is not necessary that *each* generic tableau is cycle-free.

Elaborations of the method The basic proof procedure described above, which is already quite powerful and has the advantage of being completely automatic, can be improved in several ways. For instance, it may be adapted to make use of the first-order theory of the data domain. Also, the user might be allowed to propose invariants or other guidance.

6 Conclusion

We envision the techniques described in this paper to be integrated into current design verification environments, providing interfaces to standard design languages. Given a system in one of those languages, the designer would provide formal specifications in FO-ACTL. Based on design knowledge and the properties to be checked, the designer would then debug the system by model checking stripped versions of the specifications in stripped control-expanded versions of the system. Note that the selection of the expanded set of variables will typically depend on the formula to be verified. In this phase, the full range of techniques for "classical" symbolic model checking will come into play. Only after surviving this debugging phase, *truly symbolic model checking* enters the stage.

Truly symbolic modelchecking will unfold the *CEP* in the verification process; data loops touched in this unfolding process have to be contracted using guidance on the source language level by the designer to a single transition labeled by the effect of the loop on the data variables and a condition guaranteeing termination. The verification of the purely sequential loop against such a total correctness formula is a classical task handled by a dedicated prover component, which will also have to handle termination proofs for loops claimed to be terminating by the introduction of fairness assumptions during the debugging phase. Given the contraction of loops, the techniques described in Section 5 will automatically generate verification conditions reducing the correctness of the FO-ACTL formula to be checked to a pure first order formula.

The scenario described above will be realized on the basis of the FORMAT verification tools [9], using *symbolic timing diagrams* [21] as graphical representations of FO-ACTL specifications, within a new industrial project aiming at safety critical embedded control applications.

---

[2] These are: No computations on data variables, no tests depending on them.

References

[1] Apt, K.R. *Ten years of Hoare's logic: A survey – part I*, TOPLAS **3** (1981), 431–483.

[2] Apt, K.R. and Olderog, E.-R. *Verification of sequential and concurrent programs*, Springer, New York (1991).

[3] Bradfield, J.C. *Verifying temporal properties of systems*, Birkhäuser, Boston (1992).

[4] Bradfield, J.C. and Stirling, C.P. *Verifying temporal properties of processes*, CONCUR '90, LNCS 458 (1990), 115-125.

[5] Brown,M.C., Clarke, E.M. and Grumberg, O. *Characterizing finite Kripke structures in propositional temporal logic*, TCS **59** (1988), 115–131.

[6] Burch, J.R., Clarke, E.M., McMillan, K.L. and Dill D.L. *Sequential circuit verification using symbolic model checking* DAC '90, 46–51.

[7] Clarke, E.M., Emerson, E.A. and Sistla, A.P. *Automatic verification of finite state concurrent systems using temporal logics*, POPL '83, 117–126.

[8] Clarke, E.M., Grumberg, O. and Long, D.E. *Model checking and abstraction*, POPL '92, 343–354.

[9] Damm, W., Döhmen, G., Helbig, J., Herrmann, R., Josko, B., Kelb, P., Korf, F. and Schlör, R. *Correct system level design with VHDL*, Tech. Rep., Oldenburg (1994), 54p.

[10] Damm, W., Josko, B. and Schlör, R. *Specification and verification of VHDL-based system-level hardware designs*, in Börger (ed.) Specification and Validation Methods, Oxford Univ. Press, 331–410 (to appear).

[11] Dingel, J. and Filkorn, T. *Model checking for infinite state systems using data abstraction, assumption-commitment style reasoning and theorem proving*, CAV '95, to appear.

[12] Emerson, E.A. *Temporal and modal logic*, in: Handbook of Theor. Comp. Sc., B, North Holland (1990), 997–1072.

[13] Floyd, R.W. *Assigning meanings to programs*, Proc. AMS Symp. Applied Math. 19 (1967), 19–31.

[14] Graf, S. *Verification of a distributed cache memory by using abstractions*, CAV '94, LNCS 818 (1994), 207–219.

[15] Grumberg, O. and Long, D.E. *Model checking and modular verification*, TOPLAS **16** (1994), 843–871.

[16] Herrmann, R. and Pargmann, H. *Compiling VHDL data types into BDDs*, EURO-VHDL '94, 578–583.

[17] Hojati, R. and Brayton, R.K. *Automatic datapath abstraction in hardware systems*, CAV '95, to appear.

[18] Josko, B. *Verifying the correctness of AADL modules using model checking,* in: Stepwise refinement of distributed systems: models, formalisms, correctness, LNCS 430 (1990), 386–400.

[19] Manna, Z. *Beyong model checking*, CAV '94, LNCS 818 (1994), 220–221.

[20] Manna, Z. and Pnueli, A. *The temporal logics of reactive and concurrent systems. Specification.* Springer, New York 1992.

[21] Schlör, R. and Damm, W. *Specification and verification of system-level hardware designs using timing diagrams*, EDAC '93, 518–524.

[22] Stirling, C. and Walker, D. *Local model checking in the modal mu-calculus*, TAPSOFT '89, LNCS 351, 369–383.

[23] Wolper, P. *Expressing interesting properties of programs in propositional temporal logic*, POPL '86, 184–193.