Efficient Bit-Level Model Reductions for Automated Hardware Verification

Sergey Tverdyshev*
Saarland University, Germany
deru@wjpserver.cs.uni-sb.de

Eyad Alkassar†
Saarland University, Germany
eyad@wjpserver.cs.uni-sb.de

Abstract

Transition systems which do not perform domain-specific operations on their state variables can be efficiently reduced. We present two different algorithms which automatically *eliminate domain-specific operations and reduce the domains of occurring variables from infinite to small domains. Our work extends earlier techniques which are applicable solely to combinatorial properties to temporal properties of transition systems. We have implemented our algorithm as a proof method in the Isabelle/HOL theorem prover and applied it to bit-level hardware designs. To demonstrate the efficiency of our technique, we fully automatically verify a liveness property of a pipelined processor and correctness of a memory management unit.*

1 Introduction

Hardware design companies often work on models described at the Register Transfer Level (RTL). Ideal verification techniques for hardware should target these models with a high-degree of automation. Due to the details and the complexity of these models, this ideal is not yet reality.

On the one side, algorithmic techniques (e.g. model checking, SAT, and SMT solvers) verify combinatorial and temporal properties automatically but they suffer from the infamous state explosion problem. To overcome that problem, many automatic abstraction techniques have been developed [10, 29]. Still, these techniques apply to term-level models which are already abstractions of actual implementations. For instance, term-level models of processors only implement a subset of the instruction set architecture (ISA) and data paths, memories, processor elements, such as decoders and ALU's are left uninterpreted. Moreover, there is often no obvious connection between the implementations and their abstractions.

On the other side, deductive methods can handle very large and detailed designs but require significant human efforts from expert users. For instance, the ACL2 [12] and PVS [21] theorem provers have been successfully used to verify complex out-of-order pipelined machines described at low-levels of abstraction [3, 26].

We present a fully automatic technique to verify large systems, which are characterized by domain-independency. Domain-independency requires the correctness of the system not to rely on domain-specific properties. This often holds for correctness criteria of caches and processors.

Our techniques are implemented in IHaVeIt [27] (**I**sabelle **Ha**rdware **Ve**rification **I**nfras**t**ructure), an environment based on the Isabelle/HOL theorem prover [23], the model checkers NuSMV [5] and SMV [18], as well as different SAT solvers. This environment also provides a tool to generate Verilog descriptions that are then synthesized on FPGA platforms.

The main contributions of our approach are:

- We present a unique combination of transformation and domain reduction techniques used so far only for combinatorial problems. These techniques are for the first time extended for proving temporal properties stated over transition systems. We prove our domain reduction to be sound and complete.
- We present an alternative reduction technique which syntactically determines the number of memory cells required to preserve a given temporal property over a transition system.
- Our algorithms are implemented as decision procedure for the interactive theorem prover Isabelle/HOL and their efficiency is demonstrated by applying them to non-trivial designs: correctness of a pipelined machine and a memory-management unit.

Since our algorithms works solely on the symbolic representation of the transition systems it can be understood as a preprocessor. Thus, further reduction techniques can

*This author was supported by the German Federal Ministry of Education and Research (BMBF) in the Verisoft project under grant 01 IS C38

†This author was supported by the German Research Foundation (DFG) within the program 'Performance Guarantees for Computer Systems'.

1530-1311/08 $25.00 © 2008 IEEE
DOI 10.1109/TIME.2008.11

be applied. Furthermore all hardware constructs verified in our environment could be directly synthesized.

The paper is organized as follows. In the next Section we discuss the related work. Section 2 motivates the basic ideas of the algorithms behind IHaVeIt. In Section 3 presents how we model transition systems. Section 4 describes the proposed reductions and transformations in detail. The applications of these techniques to a pipelined machine and a memory management unit are reported in Section 5. Section 6 concludes the paper and gives future research directions.

Related Work

Algorithms The principle of our data abstraction algorithm is related to symmetry reduction [10] and abstraction of data insensitive models [15, 22]. In contrast to our algorithm symmetry reduction works entirely on symbolic level rather than on the explicit state transition graph.

Lazic et al. [15] gave a semantic definition and a rigorous analysis of domain independent systems. Their domain reduction algorithm targets safety properties which can be expressed as reachability properties of concurrent programs. In contrast we can verify arbitrary CTL* properties.

Paruthi et al. [22] presented a domain reduction algorithm that distinguishes three kinds of variables: control, data, and mixed ones. While their reduction can only be applied to data variables, we extend to arbitrary variables.

Manolios [17] proposed a memory abstraction algorithm that is implemented in a bounded model checker. The algorithm basically consists of two parts: (1) unrolling the given transition relation and property to verify for a given number of steps, i.e. transformation to a SAT problem (2) compute the number of required memory cells and reduce memory domains to preserve validity of the the unrolled formula. This approach works for bounded model checking of properties of the form AG or AF. Furthermore it allows domain-specific operations on addresses. Compared to his work, our approach is not restricted to bounded model checking and can be applied on any CTL* properties.

Function elimination is related to the classical "freeing" technique and is widely used in hardware verification. For example it is applied in [9], in which the authors construct a verification model by eliminating memory arrays and retain only the memory interface signals. However, to our knowledge IHaVeIt is the first tool applying function elimination on transition systems and temporal properties rather than on combinatorial ones.

Applications to processor verification The complexity of the pipelined machine verified in this paper is comparable to the recent results obtained by Manolios and Srinivasan using ACL2 and UCLID [16]. Their approach relies on showing specific refinement relations between a bit-level model, intermediate term-level abstractions and the ISA. They verify liveness properties on term-level abstractions. In contrast, we directly target the bit-level design and only one general refinement relation has to be proven.

Jacobi [11] used the PVS theorem prover, the PVS built-in model checker, and SMV to verify complex models, such as floating point units. He can verify a slightly more complex architecture, but with substantial human interaction: the abstraction is built manually.

Lahiri et al. achieved a fairly high-degree of automation to verify safety properties on abstractions of out-of-order architectures, but liveness verification still requires intensive human efforts [14]. Velev [28] verified liveness properties of pipelined machine automatically, but his approach relies on a liveness criterion specialized for processors.

The pipelined machine verified in this paper corresponds to the processor verified by Kröning using PVS [13]. The MMU that we verified is Dalinger's optimization of the model presented in his PhD thesis [6]. It was interactively verified using PVS. Using our reduction techniques, no human interaction is needed.

2 Motivating the algorithm

Data Abstraction for Temporal Systems Suppose a model description is given by a transition system, where the transition relation is defined by a predicate over the current and the next state. This predicate is described by means of equalities, and does not make use of any domain specific operations, as e.g., integer addition. Such a transition system will only input, output, propagate and compare values of variables with each other and hence all computations are *domain independent*. For example consider the following transition system $\delta(s, s') \equiv s'.x = \text{if } (s.a = s.b) \text{ then } s.x \text{ else } s.y$. Assume we want to verify the temporal property $G(s.x = s.y)$, which states that $(s.x = s.y)$ holds in each step during a run.

For this particular example assigning a domain of size three to x and y and a domain of size two to a and b is sufficient for proving or disproving the temporal property. This is the basic idea of our data abstraction algorithm, which we prove to be *sound* and *complete*, i.e. a temporal property holds on the transition system with reduced domains, if and only if it holds on the original one (see Section 4.2). Its correctness relies on the following basic observations:

- during a run the value of x either stays the same or it takes the value of y
- the values of variables x, y are neither compared, nor assigned to the variables a or b
- the flow of data does neither depend on particular values of x and y, nor on those of a and b, only the values of occurring equalities are important
- to maintain the values of the equalities appearing in the transition predicate and in the property during a run, it

suffices only to consider some finite domains for the occurring variables.

Function and Memory Elimination For modeling interesting systems, usage of function symbols, which most importantly include memory accesses, is essential. Allowing function symbols and memories as part of the model description is however a challenging task, since reading from a function or a memory are domain-specific operations, which violate the domain-independency of our model. Often the correctness properties do not depend on the concrete interpretation of the functions or content of the memory. In those cases, the classical solution would be to treat any reading from a function or a memory as a non-deterministic input. This well-known technique is widely used in hardware verification [4] for combinatorial problems. We extend it for the first time to transition systems. This extension will lead to a significant loss in completeness of the method, which we overcome with a new memory reduction algorithm.

First we describe how to transfer the classical solution to transition systems. Every function application or read from memory is replaced by a fresh variable. Additionally some constraints preserving functional consistency have to be added for the current state. For example if the two function applications $f(a)$ and $f(b)$ are replaced by the fresh variables v_{fa} and v_{fb}, one has to ensure that their values are equal whenever a and b are equal: $a = b \implies v_{fa} = v_{fb}$.

Note that the added constraint only maintain function consistency in one state, but not over time. In the following example the transition system assigns x in each step the value of the application $f(3)$: $\delta(s, s') \equiv s'.x = f(3)$ Obviously, then the value of x will stay constant over time: $f(3) = 10 \wedge x = 10 \implies G(x = 10)$. After transformation this is no longer true, since the newly introduced variable v_{f3} is not further constrained, and its interpretation may vary over time:

$$\delta(s, s') \equiv s'.x = v_{f3}$$
$$v_{f3} = 10 \implies G(x = 10)$$

In short we see that function elimination is *sound* but not *complete*, i.e. not all properties which hold on the original model can be proven on the one after function elimination (see Section 4.1). Lack of completeness is due to: (i) the concrete function interpretation is drawn (ii) functions and memories do not stay stable over time, e.g. memory updates are lost.

Memory Abstraction The first listed drawback of the previous approach is endurable because the user determines which functions are left uninterpreted and which not. Yet, loosing temporal functional consistency turns out to be far more serious. Handling properties whose correctness depend on distinguishing between accesses to different memory locations at different times becomes impossible. A natural example where such a requirement is needed is the verification of a cache system. To overcome the above described problem, we propose in Section 4.3 an alternative algorithm, which does not abstract the memory away but only reduces the number of modelled memory cells. The basic observation is similar to that of the domain-reduction algorithm: the number of memory addresses needed to be modelled for proving the property can be deduced syntactically from the description of the transition system and the property. Even though this reduction may generate larger domains, it preserves the expected memory-semantics.

3 Background: Propositional logic and Kripke structures

3.1 Logic of Equality with Function Symbols (EFS)

The Logic of Equality (EFS) is propositional logic, augmented with terms and equality over terms. For clarity, we consider variables to range over integer domains. A term t is an integer constant, a variable, a memory, *if-then-else* construct (ITE), a function application or a memory read or write access. Formulae are equalities between two terms, Boolean variables, negations or conjunctions of formulae. Let Var, $BVar$, $MVar$ and Fun denote the sets of variable, Boolean variable, memory and function names. Formally, the set of valid terms and formulae is defined as follows:

$$\begin{array}{ll} t ::= & c \in \mathbb{N} \mid x \in Var \mid ITE(\psi, t_1, t_2) \mid \\ & m \in MVar \mid read(m, t) \mid write(m, t_1, t_2) \mid \\ & f(t_1, \ldots, t_n) \text{ where } f \in Fun \\ \psi ::= & b \in BVar \mid \psi_1 \wedge \psi_2 \mid \neg\psi \mid \ t_1 = t_2 \end{array}$$

An interpretation is a pair (I_v, I_f) consisting of a mapping I_v from variable or memory names to boolean, integer values or memories, which are functions of type $\mathbb{N} \to \mathbb{N}$, and a mapping I_f from function names to functions. An interpretation (I_v, I_f) *satisfies* a formula ψ, written $(I_v, I_f) \vDash \psi$, if ψ evaluates to true under I_v and I_f. A recursive definition of $\vDash$ is given in Table 1.

For a given function interpretation I_f, a formula ψ defines a predicate ψ^{I_f} over the domain of its variables. Formally, this predicate is defined as follows:

$$\psi^{I_f}(v_1, \ldots, v_n) \triangleq$$
$$(I_v, I_f) \vDash \psi \wedge I_v(x_1) = v_1 \wedge \cdots \wedge I_v(x_n) = v_n$$

3.2 Kripke Structures in EFS

Transition Systems are modelled as Kripke structures. A Kripke structure K over a set of atomic propositions K.AP is defined as a 4-tuple $K = (S, S_0, R, L)$, where

- S is a finite set of states,

term t	**evaluation under** $I = (I_v, I_f)$
c	c
x, b, m	$I_v(x), I_v(b), I_v(m)$
$ITE(\psi, t_1, t_2)$	if $I \vDash \psi$ then $I(t_1)$ else $I(t_2)$
$f(t_1, \ldots, t_n)$	$I_f(f)(I(t_1), \ldots, I(t_n))$
$read(m, t)$	$I_v(m)(I(t)))$
$write(m, t_1, t_2)$	$I_v(m)[I(t_1)] := I(t_2)$
formula ψ	**evaluation under** $I = (I_v, I_f)$
$\neg\psi$	$\neg(I \vDash \psi)$
$\psi_1 \wedge \psi_2$	$I \vDash \psi_1 \wedge I \vDash \psi_2$
$t_1 = t_2$	$I(t_1) = I(t_2)$

Table 1. Evaluation of terms and formula

- $S_0 \subseteq S$ denotes the set of initial states,
- $R \subseteq S \times S$ is a total transition relation: $\forall s \in S. \exists s'. (s, s') \in R$
- $L : S \to 2^{AP}$ is a labeling function, which maps each state to the set of propositions that hold in that state.

A relation $B \subseteq S \times S'$ is a *simulation* of two Kripke structures K and K', if for all elements $(s, s') \in B$:

- $L(s) \cap K'.AP = L'(s')$
- $\forall s_1. (s, s_1) \in R \implies \exists s'_1. (s', s'_1) \in R' \wedge B(s_1, s'_1)$

We say K' *simulates* K, written as $K' \succ K$ if there is a simulation relation B, such that for each initial state s_0 in K, an initial state s'_0 in K' exists with $B(s_0, s'_0)$.

A relation $B \subseteq S \times S'$ is a *bisimulation* of two Kripke structures K and K' if for all elements $(s, s') \in B$:

- $L(s) = L(s')$
- $\forall s_1. (s, s_1) \in R \implies \exists s'_1. (s', s'_1) \in R' \wedge B(s_1, s'_1)$
- $\forall s'_1. (s', s'_1) \in R' \implies \exists s_1. (s, s_1) \in R \wedge B(s_1, s'_1)$

Two Kripke structures K and K' are called *bisimular*, written $K \approx K'$, if a bisimulation B of K and K' exists and if the initial states are also within the bisimulation relation.

It is well-known that simulation preserves ACTL* properties and that bisimulation preserves CTL* properties [8, pp. 171–178].

Kripke structures can be represented in EFS by specifying context functions. The context of a Kripke structure K is a set of EFS formulae unambiguously describing K. The initial states, the transition relation, and the labeling function are considered as predicates over state variables s or pairs (s, s') of a state and a next state variable. For a given function interpretation I_f, we can describe a Kripke structure by the following predicates: $\psi^{I_f}_{S_0}(s)$ for the initial states, $\psi^{I_f}_{R}(s, s_1)$ for the transition relation and for each atomic proposition $ap \in K.AP$ we define a predicate $\psi^{I_f}_{ap}(s)$. For instance, from formula ψ_{S_0} we can construct predicate $\psi^{I_f}_{S_0}$ through the following equivalence: $(I_v, I_f) \vDash \psi_{S_0} \Leftrightarrow I_v(s) \in S_0$.

Formally, a context of a Kripke structure K parametrized by a function interpretation I_f, noted $co(K^{I_f})$, is given by the triple: $co(K^{I_f}) = (\psi^{I_f}_{S_0}, \psi^{I_f}_{R}, \{\psi^{I_f}_{ap} \mid ap \in K.AP\})$.

4 The algorithms behind IHaVeIt

4.1 Function Elimination

Often, the actual definition of a function is irrelevant for the truth value of a formula, i.e. this formula is valid for all interpretations of the function. Same holds for *read* and *write* operations to the memories. In the following we will not distinguish between memory and function accesses. A simple function elimination algorithm is based on the Burch and Dill approach [4]. We extend this technique to Kripke structures.

Application to formulae

Ackermann [1] proposed a methodology for eliminating function symbols, where each function application $f(x)$ is replaced by a new variable f_x. Furthermore he adds restrictions to impose functional consistency: if the arguments of the original function applications are equal, the new variables are equal too. Obviously, this transformation preserves formulae satisfiability (and validity).

Burch and Dill's approach to function elimination relies on the usage of ITE constructs [4]. For each syntactically different occurrence of a function application a new ITE is added. For example, the three function applications $f(x)$, $f(y)$ and $f(z)$ are replaced by f_x, $ITE(x = y, f_x, f_y)$ and $ITE(y = z, ITE(x = y, f_x, f_y), f_z)$.

We denote the above outlined ITE-transformation of a formula ψ to one without function symbols by ψ^*.

Lemma 1 *Given a formula ψ and an interpretation (I_v, I_f). Let I^*_v be an interpretation for the fresh variables. For the ITE-transformed formula the following holds:*

$$\forall I^*_v : I_v \cup I^*_v \vDash \psi^* \implies (I_v, I_f) \vDash \psi$$

Obviously the ITE-transformation does only change and depend on the equations of a formula, not on its structure. Hence, we can apply it also to a set of formulae $\{\psi_1, \ldots, \psi_n\}^*$ where function names of all formulae are substituted simultaneously. Moreover, for each ψ^*_i of the resulting set the property of Lemma 1 holds.

Application to Kripke Structures

We apply function elimination to a Kripke K, by first translating it to a context, then we apply ITE-transformation on the context and we subsequently translate it back to a Kripke structure. The resulting Kripke structure is denoted by K^*. It contains no function symbols but new domain variables, which represent the result of the corresponding function applications. Since memories only occur in read and write operations, and because these operations are also eliminated we do not longer need to model the memories in the state space. The temporal behavior of these domain variables is not specified and hence they can change non-deterministically and functional consistency over time is lost. In particular, memory semantics is not preserved.

Note that the resulting Kripke structure K^* contains more transitions than a "union" of all Kripke structures K^{I_f} for all possible interpretations I_f.

It is easy to prove that K^* simulates K:

Theorem 1 $K^* \succ K$

4.2 Data abstraction

A well known property of equality logic *without* function symbols, the small model property, states that the domain of variables in a formula can be efficiently reduced by maintaining satisfiability and validity [24]. Our data abstraction algorithm exploits the mentioned property and extends it for shrinking the domains of the state variables of Kripke structures.

An important feature of the presented algorithm is that it does not modify the structure of the formulas describing the Kripke structure nor the formula to be verified. Only the variable domains are reduced and occurring constants are substituted.

In this section we introduce data abstraction for the model without function symbols, i.e. we assume that all present function symbols, including memory accesses have been eliminated by the technique presented in the previous Section. We prove soundness and completeness of the domain-reduced abstraction.

The basic observation of our modelled systems is that they do not distinguish among different elements of variable domains. Therefore such systems are often labeled as data *symmetric* or data *independent*. The idea is to exploit this property through analyzing data propagation in the model and determine those variables which are only compared to each other, without applying domain-specific operations. For those variables we are not interested in their concrete values, but only how they determine the evaluations of the equations in the corresponding predicates for the initial state set, the transition relation and the atomic propositions.

For example, given a formula $v_1 = ITE(c, v_2, v_3)$, we deduce that variables v_1, v_2, and v_3 could be compared with each other during a run. Equality and inequality between them could be maintained by small domains. For determining how small these domains indeed could be chosen, we need a formal definition of variables being compared to each other during runs of the system.

Definition 1 *Given a term or formula we define the set of related terminal nodes (e.g. variables, and constants) recursively:*

$$\begin{aligned} rnodes(t_1 = t_2) &= rnodes(t_1) \cup rnodes(t_2) \\ rnodes(ITE(\psi_b, t_1, t_2)) &= rnodes(t_1) \cup rnodes(t_2) \\ rnodes(v) &= \{v\} \\ rnodes(c) &= \{c\} \end{aligned}$$

Definition 2 *For a given formula ψ we define a related relation on subformulas as:*

$$R_\psi = \{(t_1, t_2) \mid \exists \psi' \sqsubseteq \psi.\ \{t_1, t_2\} \subset rnodes(\psi')\}$$

Now we simply define the set of variables compared with each other as the transitive closure of R_ψ: the equivalence relation E_ψ. In the following we use the notation $[e]_{E_\psi}$ to denote the equivalence class containing the node e, where a node is either a formula or term. By writing $e_1 \sim_{E_\psi} e_2$ we denote that the two nodes e_1 and e_2 belong to the same equivalence class.

For a given Kripke structure K the data abstraction algorithm works as follows:

1. Compute the context $co(K)$ of K. Initially add all pairs of state and corresponding next state variables $s.x$ and $s'.x$ to the same equivalence class.

2. Compute the equivalence relation E_ψ of compared variables for each context formula ψ. Merge intersecting equivalence classes. The result denoted by E_K is again an equivalence relation.

3. Update the context:

- Assign to each variable x a new abstract domain of size which equals the cardinality of the corresponding equivalence class: $\mid [x]_{E_K} \mid$.
- Replace every constant occuring in a context formula by a value from the new abstracted domain, according to the following rule: two originally different constants will be replaced by two different abstract values: $\forall c_1.\ \forall c_2.\ c_1 \sim_{E_K} c_2 \implies c_1 = c_2 \equiv c_1' = c_2'$.

4. Construct the new, abstract Kripke structure K^{abs} from the updated context.

We claim that the size of the new domains is big enough to preserves the values of all equalities during any run of the model. Basically, the proof depends on a bisimulation relation between concrete and abstract variable values, which

obey the following rule: If two variables (variables refer here to field names of a state variable s) are in the same equivalence class, their abstracted values must be equal if and only if their original ones were.

Lemma 2 *Given a context formula ψ and its abstraction ψ^{abs} we define a mapping between values of the old and new domain as follows: whenever two variables are related to each other by the same equation in the formula, we require their new values to be the same if and only if their old values were. Same should hold for comparison between variable values and constants.*

$$A_{E_\psi}(s, s_{abs}) \equiv$$
$$s.x_i \sim_{E_\psi} s.x_j \Rightarrow (s_{abs}.x_i = s_{abs}.x_j \Leftrightarrow s.x_i = s.x_j) \wedge$$
$$s.x_i \sim_{E_\psi} c \Rightarrow (s_{abs}.x_i = c_1 \Leftrightarrow s.x_i = c)$$

Then for the abstracted formula it holds:

$$A_{E_\psi}(s, s_{abs}) \implies \psi(s) \equiv \psi^{abs}(s_{abs})$$

Proof: Simple induction over the structure of equations.

Since the formulae in the contexts range over the state and the next state variable, $A_{E_K}(s, s_{abs})$ relates transitions to abstracted transitions. However, for bisimulation we are interested in a relation between states, not between transitions. The sought relation $B_{E_K}(s, s_{abs})$ can be obtained from A_{E_K}, by just ignoring all rules related to next state variables.

Lemma 3 $K^{abs} \approx K$

Proof sketch: We choose B_{E_K} as bisimulation. From Lemma 2 we deduce $\psi_{ap}(s) = \psi^{abs}_{ap}(s_{abs})$ and $\psi_{S_0}(s) = \psi^{abs}_{S_0}(s_{abs})$. Next we have to simulate each transition (s, s') of K by some (s_{abs}, s'_{abs}) in K^{abs}. Lemma 2 implies the existence of a transition (s, s') with $B_{E_K}(s, s_{abs})$ if $A_{E_K}((s, s'), (s_{abs}, s'_{abs}))$ holds. Finding an s'_{abs} satisfying the last predicate, can easily be done since the domains are sufficiently large. Proving the opposite simulation direction is even simpler.

4.3 Alternative to function elimination: Reducing the number of memory cells

In this section we present an alternative approach to function elimination, which reduces the domains of memories rather than eliminating them. Additionally we then can treat uninterpreted functions as memories, which will not be updated over time and apply the same reduction technique to those. This algorithm allows to overcome the drawback introduced by function elimination, namely loosing memory semantic, and hence completeness. Furthermore the algorithm even supports comparison of complete memories. However, its application may result in an abstracted model with a larger state space.

Note that our model does not distinguish between address and data types and hence reading and writing from and to the memory are no domain specific operations for us. It suggests itself to reduce the number of modelled memory cells by simply applying our domain reduction algorithm to the memory domain, i.e. the address space.

Though, there are two main reasons why this naive approach is futile:

- Domain and codomain of a memory have to be treated separately.
- Some sort of *temporal aliasing*: syntactically equivalent memory addresses could refer to different memory locations at different times.

For example consider a transition system where the memory m is never updated: $\delta(s, s') \equiv s.m = s'.m$, and the following property, where F stands for the temporal operator *finally*: $((F(m(a) = 2)) \wedge (F(m(a) = 5))) \implies G(m(a) = 2 \vee m(a) = 5)$. In case the original domain size of m is larger than two, the formula is obviously false because a could address more than two memory locations. Hence, the memory domain in this example can not be reduced to less than three memory cells.

Manolios [17] proposed simply to unroll the definition of the transition system together with the property for a given number of steps. A memory access in each step becomes syntactically distinguishable and obviously the model will not use more memory locations during the execution of the given number of steps. This enables him to bounded model check the property for a reduced domain. We generalize this idea to general model checking by syntactically analyzing how many memory cells are distinguishable by the transition system and the property, rather than analyzing how many are used.

The key to memory domain reduction is to collect all terms used as addresses in the symbolic description of the Kripke structure *and* the property. Note that we have to treat syntactically equal address terms which appear under different temporal operators in the property as two different memory locations. That is due to the fact, that terms can be evaluated to different values at different time points.

First we extend the definition of *related* equivalence classes to memory symbols.

$$\begin{aligned} rnodes(read(m, t)) &= \{read(m, t)\} \\ rnodes(write(m, t_1, t_2)) &= \{rnodes(m)\} \\ rnodes(m) &= \{m\} \end{aligned}$$

The number of needed memory cells can then be computed as follows:

- Compute the equivalence relation E_K

- Collect all terms which are used as memory addresses in the Kripke structure and the property to a multiset T_{addr}
- For each address term t determine the maximum domain it can be assigned to:

$$maxdom_t = \max\{|[e]_{E_K}| \mid e \in rnodes(t)\}$$

  Select the maximum domain over all address terms:

$$maxdom_{T_{addr}} = \max\{maxdom_t \mid t \in T_{addr}\}$$

- Next compare the maximal domain some address term can be assigned to, with the number of memory accesses. Select the larger of these two numbers:

$$maxmemory = max\{maxdom_{T_{addr}}, \mid T_{addr} \mid\}$$

- Finally add the number of complete memory comparisons, to model all not directly accessed addresses and reduce the memory domain size to that value:

$$maxmemory + (|[m]_{E_K}| - 1)$$

Theorem 2 *For a given Kripke structure K and a CTL* formula ψ compute the abstracted version K' and ψ' according to the above described algorithm. Then it holds:*

$$K \models \psi \equiv K' \models \psi'$$

Note that the algorithm exploits the following CTL* property: it can't remember contents of memory-cells of the past, except through assignments (which are equalities). But for each additional assignment our algorithm will increase the number of memory cells by one.

5 Applications

Our algorithms are part of the IHaVeIt environment, which allows the description of bit-level hardware models and their translation to the Hardware Description Language (HDL) Verilog [25] for synthesis on FPGA platforms. A typical user work flow is to define the model in the Isabelle/HOL, state the property and call IHaVeIt as the proof method. The user can mark the functions those definitions are irrelevant for the property (e.g., ALU) and IHaVeIt treat them as uninterpreted. In contrast to the most of related work the last step is done formally, i.e. this abstraction is *justified* by Isabelle/HOL+IHaVeIt. The efficiency of our algorithms is demonstrated in the following subsections by the automatic verification of two different bit-level hardware designs: liveness of a pipelined machine and correctness of a memory management unit.

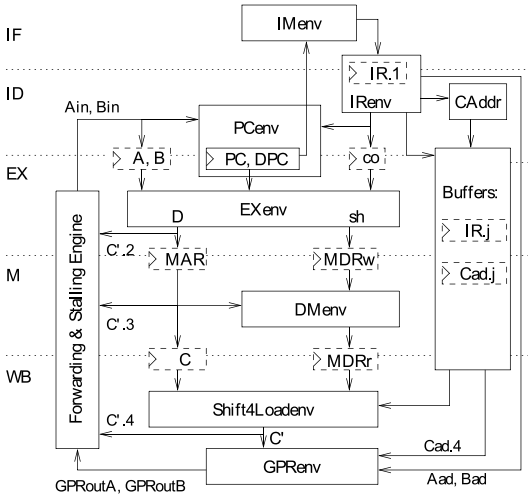

Figure 1. Data paths of the DLX processor

5.1 DLX Like-Processors

We consider a five-stage pipelined machine, the high-level architecture of which is given in Figure 1 [20]. The five stages of the pipeline are instruction fetch (`IF`), instruction decode (`ID`), execution (`EX`), memory write/load accesses (`M`), and result write back (`WB`). The machine implements 36 instructions (e.g., ALU, shifts, moves, load, store and branch instructions) and the length of instructions is set to 32 bits. The data path width can be set to *any integer value* and the verification time of our proofs is independent of this value. The processor comprises 32 general purpose registers (GPR), 31 internal registers (such as PC, DPC, instruction register), and the memory interface has type $2^{30} \rightarrow 2^{32}$. The model includes a delayed PC, as well as a three-stage forwarding and stalling mechanism. The model description is 2778 line of code and contains *bit-level* description of all parts, e.g. ALU, shifters, incrementors. Our DLX machine is similar to the most complex benchmark used in recent related works (e.g., Manolios et al. [17]) but it is much more detailed.

The processor liveness property requires that an instruction is *eventually* fetched and it *eventually* leaves the processor. We prove this property by showing liveness of the processor stalling logic. We assume that the instruction and data memories (memory busy signals) possess liveness property. For the five-stage processor we have five stall signals. They control when the stage should be stalled, i.e. no instructions in upper stages can proceed. We formulate this theorem [13] using LTL:

$$(S_{MA}, I_{MA}, \delta_{MA}) \models_{LTL}$$
$$GF(\neg IMbusy) \wedge GF(\neg DMbusy) \longrightarrow$$
$$\forall 1 \leq i \leq 5.\, GF(\neg stall_i)$$

This theorem states that if `IMbusy` and `DMbusy` signals are finitely often true, then every stage will be stalled for finite number of cycles. The liveness property was verified completely automatically by usage of IHaVeIt.

We also check a version of the DLX processor with a bug in the stalling engine. The Table 2 shows verification time for the correct machine (dlx5) and the buggy machine (dlx5b). The results are obtained by applying NuSMV with and without preprocessing by IHaVeIt. The preprocessing makes application of the model checker feasible at all. The similar strategy we use with SMV, which was run with counterexample-based abstraction turned on. Here we get several time speed-up over plain usage of SMV.

5.2 Memory Management Unit

A Memory management unit (MMU) is a hardware support for virtual memory. A virtual memory consists of two parts: a main memory (e.g., RAM) and a swap memory (e.g., hard drive disk). An MMU is usually placed in the processor just before the memory interface. A processor with an MMU runs either in user mode or in system mode. In system mode, an MMU is irrelevant to memory accesses and all memory accesses are redirected directly to the main memory. In user mode, address translation has to be done. After address translation, either the memory access is redirected to some main memory location or a page fault exception is generated. In the last case, the processor enters the system mode and executes the page fault handler. (A page fault handler exchanges data between main and swap memory). For more details on virtual memory and MMU's we refer the reader to [7].

Our MMU model is an optimization of the MMU presented in [7]. The optimized MMU has a translation lookaside buffer (TLB), that caches recently used translations to speed-up following translations. The MMU implementation itself is not really huge but in order to verify it the interfaces to/from the TLB, the processor, and the memory have to be modelled. These interfaces increase the model state space drastically, e.g. memory has type $2^{29} \rightarrow 2^{64}$.

The MMU correctness can be split according to the access type, i.e. read/write, translated/untranslated, with/without exception. The following assumptions must hold: (i) $proc\_if\_correct$ – the processor has a correct behavior (e.g., it doesn't start a new request while a previous one is still running). (ii) $mem\_if\_correct$ – the memory interface is correct (e.g., the memory is alive and its content is stable). We describe the correctness of a read access [6] without exception directly in LTL as follows:

$$
\begin{array}{c}
(S_{MMU}, I_{MMU}, \delta_{MMU}) \models_{LTL} \\
proc\_if\_correct \wedge\ mem\_if\_correct \longrightarrow \\
G\,(read\_req\_start \wedge\ (X \neg mmu\_out\_excp) \longrightarrow \\
\neg end\_req\, U\ (end\_req \wedge mmu\_out\_data\_correct))
\end{array}
$$

This lemma states that if there is a read request from the processor and MMU doesn't generate an exception then (i) eventually the end of the request is signalled and the output data is the content of the memory cell on the translated address, and (ii) in between there was no request end, i.e. no request will be lost by MMU. The Table 2 shows verification time for the read and write accesses. We experimented with two different formulation of the properties: weak formulation would require some additional user work to reuse it in the verification of a processor; the strong ones is the properties given in [6]. With respect to the plain application of NuSMV we get several order magnitude speed-up. We could not apply the SMV because our LTL formulae contain past temporal operators which are not supported by SMV.

6 Summary and Future Work

We have presented two domain reduction algorithms for Kripke structures: one makes use of function elimination and another, which is complete, reduces the number of modelled memory addresses. They combine and extend earlier techniques which are applicable solely to combinatorial properties to temporal properties of Kripke structure. These algorithms are implemented in an environment named IHaVeIt, which is integrated in Isabelle/HOL and uses NuSMV, SMV, and SAT solvers as decision procedures for temporal properties of bit-level hardware designs. Our experimental evaluation demonstrates significant speed-up of the verification time over the current state-of-the-art model checkers.

The main difference to most previous approaches is that we target bit-level models directly, without introducing informal abstractions. Moreover the algorithms operate solely on symbolic level (preprocessing) and can be combined with any further reduction techniques. The implemented environment is also more general than the previous work in the sense that it allows direct application of the model checking technique on the huge and detailed system descriptions. This allows us to verify effectively not only specific pipelined machines, but any kind of hardware designs.

We have applied our tool to the verification of bit-level synthesizable hardware designs, such as pipelined processors or memory management units. Our processor example is comparable in its complexity to related works (e.g., [16, 28]). There are many interesting ways to improve the tool, e.g. combination of data abstraction with counter example guided predicate abstraction and exploiting more sophisticated domain reductions. We are currently applying IHaVeIt to the verification of safety and liveness properties of complex out-of-order processors like the VAMP [7], timed-triggered bus interfaces [2], and cache systems [19].

The IHaVeIt implementation, examples, and documentation are available online:

	NuSMV	NuSMV + IHaVeIt
dlx5	$>$ 2days	23 h
dlx5b	$>$ 2days	1 h 09 m
mmu(read, weak)	1 m 24.27 s	0.43 s
mmu(write, weak)	1 m 24.27 s	0.43 s
mmu(read, strong)	$>$ 2days	1.61 s
mmu(write, strong)	$>$ 2days	2.14 s
	SMV	SMV + IHaVeIt
dlx5	2.63 s	0.35s
dlx5b	0.46 s	0.27s

Table 2. Comparing verification time with and without preprocessing by IHaVeIt

`http://www-wjp.cs.uni-sb.de/ihaveit/`

References

[1] W. Ackermann. Solvable cases of the decision problem. In Studies in Logic and the Foundations of Mathematics. North-Holland, Amsterdam, 1954.

[2] E. Alkassar, P. Böhm, and S. Knapp. Formal correctness of a gate-level automotive bus controller implementation. In *6th IFIP Working Conference on Distributed and Parallel Embedded Systems (DIPES08), to appear*. Springer Science and Business Media, 2008.

[3] S. Beyer, C. Jacobi, D. Kröning, D. Leinenbach, and W. Paul. Putting it all together - formal verification of the VAMP. *STTT Journal, Special Issue on Recent Advances in Hardware Verification*, 2005.

[4] J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In *CAV '94*, pages 68–80, London, UK, 1994. Springer.

[5] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An open source tool for symbolic model checking. In *CAV '02*, pages 359–364. Springer, 2002.

[6] I. Dalinger. *Formal Verification of a Processor with Memory Management Units*. PhD thesis, Saarland University, Saarbrücken, 2006.

[7] I. Dalinger, M. Hillebrand, and W. Paul. On the verification of memory management mechanisms. In D. Borrione and W. Paul, editors, *CHARME 2005*, LNCS, pages 301–316. Springer, 2005.

[8] J. Edmund M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.

[9] M. K. Ganai, A. Gupta, and P. Ashar. Verification of embedded memory systems using efficient memory modeling. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 1096–1101, Washington, DC, USA, 2005. IEEE Computer Society.

[10] C. N. Ip and D. L. Dill. Better verification through symmetry. *Form. Methods Syst. Des.*, 9(1-2):41–75, 1996.

[11] C. Jacobi. Formal verification of complex out-of-order pipelines by combining model-checking and theorem-proving. In *Computer Aided Verification (CAV 02)*, volume 2404 of *LNCS*, pages 309–323. Springer, 2002.

[12] M. Kaufmann, J. S. Moore, and P. Manolios. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[13] D. Kröning. *Formal Verification of Pipelined Microprocessors*. PhD thesis, Saarland University, Saarbrücken, 2001.

[14] S. K. Lahiri, S. A. Seshia, and R. E. Bryant. Modeling and verification of out-of-order microprocessors in UCLID. In *FMCAD '02*, pages 142–159, London, UK, 2002. Springer.

[15] R. Lazic, T. C. Newcomb, and B. Roscoe. Polymorphic systems with arrays, 2-counter machines and multiset rewriting. *Electr. Notes Theor. Comput. Sci.*, 138(3):61–86, 2005.

[16] P. Manolios and S. K. Srinivasan. A framework for verifying bit-level pipelined machines based on automated deduction and decision procedures. *Journal of Automated Reasoning*, 37(1–2):93–116, 2006.

[17] P. Manolios, S. K. Srinivasan, and D. Vroon. Automatic memory reductions for RTL model verification. In *ICCAD '06: Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*, pages 786–793, New York, NY, USA, 2006. ACM Press.

[18] K. L. McMillan. The SMV language. Technical report, Berkeley Labs, 1999.

[19] C. Müller. Verification of a simple cache system,, Master Thesis, Saarland University, Saarbrücken, 2007.

[20] S. Müller and W. Paul. *Computer Architecture: Complexity and Correctness*. Springer, 2000.

[21] S. Owre, N. Shankar, and J. Rushby. PVS: A prototype verification system. In *CADE'92*, pages 748–752, 1992.

[22] V. Paruthi, N. Mansouri, and R. Vemuri. Automatic data path abstraction for verification of large scale designs. In *ICCD '98: Proceedings of the International Conference on Computer Design*, page 192, Washington, DC, USA, 1998. IEEE Computer Society.

[23] L. C. Paulson. *Isabelle: a generic theorem prover*, volume 828 of *LNCS*. Springer, New York, NY, USA, 1994.

[24] A. Pnueli, Y. Rodeh, O. Strichmann, and M. Siegel. The small model property: how small can it be? *Inf. Comput.*, 178(1):279–293, 2002.

[25] V. Sagdeo. *The Complete Verilog Book*. Springer, 1998.

[26] J. Sawada and W. A. Hunt. Processor verification with precise exceptions and speculative execution. In *Proc. 10th International Computer Aided Verification Conference*, pages 135–146, 1998.

[27] S. Tverdyshev. Combination of Isabelle/HOL with automatic tools. In B. Gramlich, editor, *FroCoS 2005*, volume 3717 of *Lecture Notes in Computer Science*, pages 302–309. Springer, 2005.

[28] M. N. Velev. Automatic formal verification of liveness for pipelined processors with multicycle functional units. In *CHARME*, pages 97–113, 2005.

[29] M. N. Velev and R. E. Bryant. TLSim and EVC: a term-level symbolic simulator and an efficient decision procedure for the logic of equality with uninterpreted functions and memories. *Int. J. Embedded Systems*, 1(1/2):134–149, 2005.