EPR-Based Bounded Model Checking at Word Level [⋆]

Moshe Emmer[1], Zurab Khasidashvili[1], Konstantin Korovin[2], Christoph Sticksel[2], and Andrei Voronkov[2]

[1] Intel Israel Design Center, Haifa 31015, Israel
{memmer,zurabk}@iil.intel.com
[2] The University of Manchester, School of Computer Science, UK
{korovin|sticksel}@cs.man.ac.uk, andrei@voronkov.com

Abstract. We propose a word level, bounded model checking (BMC) algorithm based on translation into the effectively propositional fragment (EPR) of first-order logic. This approach to BMC allows for succinct representation of unrolled transition systems and facilitates reasoning at a higher level of abstraction. We show that the proposed approach can be scaled to industrial hardware model checking problems involving memories and bit-vectors. Another contribution of this work is in generating challenging benchmarks for first-order theorem provers based on the proposed encoding of real-life hardware verification problems into EPR. We report experimental results for these problems for several provers known to be strong in EPR problem solving. A number of these benchmarks have already been released to the TPTP library.

1 Introduction

SAT-based Bounded Model Checking (BMC) [4] is currently the most widespread formal verification method in the hardware industry used for bug finding. Despite the rapid development of SMT [21] and first-order Theorem Proving (TP) [26] techniques for model-checking at word level, their positive impact has been mainly seen on software verification. So far, hardware verification has benefited far less from word-level verification, and applying SMT and TP techniques to hardware verification remains a difficult challenge to the verification community. This is mainly due to the fact that most of the hardware descriptions are written at very low-level, e.g., without explicit usage of arithmetic operations. Nevertheless, there are natural word-level components in hardware designs, in particular memories and bit-vectors, which are challenging for the bit-level verification due to the size of their bit-level representations. Efficient reasoning, at word level, with bit-vectors and arrays is an active research area [28, 8, 1, 14, 24, 6, 7].

This paper focuses on an encoding of the BMC problem with memories and bit-vectors into first-order logic (FOL) and in particular into the Effectively PRopositional (EPR) fragment. The EPR fragment, also called the Bernays-Schönfinkel-Ramsey fragment, consists of first-order formulas with no occurrences of function symbols other than constants and which when written in prenex normal form have the quantifier prefix $\exists^*\forall^*$. Skolemization applied to EPR formulas can introduce only constant function

[⋆] This work is partially supported by EPSRC, the Royal Society and a grant from Intel.

symbols, this can be used to show decidability of the EPR fragment. There are a number of efficient solvers [3, 10, 19, 25] for this fragment as demonstrated at the annual CASC TP competition [27]. Several important verification problems have been encoded into EPR [23, 17, 13, 2], benefiting from succinct representations possible in this fragment. Solvers are becoming increasingly scalable to industrial size problems and therefore it is promising to develop efficient encodings of Model Checking (MC) [9] into EPR.

The first encoding of BMC into EPR was proposed in [23], covering entire linear temporal logic. The transition relation and the initial and final states are specified there via Boolean constraints, and when encoding the unrolled transition relation, the state variables are treated as predicates over states (or over time). This enables a succinct representation of the unrolled system. The main contribution of that paper is theoretical, and no experimental results comparing the method with SAT-based BMC were reported.

Another, completely orthogonal, way to encode the MC problem into EPR was studied in [17, 13]. These works explore encodings of hardware MC problems at word level into EPR. In particular in the so-called *relational encoding* approach, bit-vectors are modeled as unary predicates over bits (or bit-indexes), addresses are modeled as terms, and memories are modeled as binary predicates over addresses and bits. Appropriate axiomatization of bits and bit-ranges allows for a sound and complete encoding of hardware MC problems with bit-vector and memory operations into EPR. These papers report initial results of experimental evaluation of EPR-based first-order verification compared with SMT-based verification on equivalence checking problems at Intel.

In this paper we present an encoding scheme which allows us to retain the strengths of both previous approaches. An ad-hoc combination of the previous two approaches would yield an encoding which in most cases will generate problems outside of the EPR fragment. One of the main issues here is that memory addresses on the one hand occur as arguments in memory predicates and therefore should be treated as terms, and on the other hand addresses depend on the state of the transition system and therefore are functions of state. Presence of non-constant functions in the encoding brings the resulting specification of the transition system outside EPR. Even very small non-EPR problems originating from toy model-checking examples are very hard for the strongest theorem provers. Among the main contributions of this paper are techniques allowing one to keep the translated verification problems within the EPR fragment. In particular, i) we introduce *address unrolling* to eliminate address functions and ii) we use *inlining* to eliminate definitions which after Skolemization result in non-EPR formulas. We evaluate our EPR-based encoding on model checking problems obtained from industrial hardware designs used at Intel. We show that in many cases EPR-based BMC can reach higher unrolling bounds than traditional SAT-based BMC.

The rest of the paper is structured as follows. In the next section we present a generic translation scheme that, given a specification of transition relation, initial state constraints and final state constraints in first-order logic, produces a description of the unrolled system up to a bound k that faithfully models transition paths of length k from the initial to the final states. While FOL is closed under the translation, EPR is not. The encoding that we describe below in Section 3 can be seen as a result of this translation applied to the specification of hardware at word level in EPR as described in [13]. This encoding brings the specification of unrolled system outside the EPR fragment.

Therefore, in Section 4 we describe basic transformations allowing the resulting unrolled system to be formalized within EPR. A number of further optimizations that help generating CNFs that are much simpler to solve are described in Section 5. We remark on incremental solving in Section 6. Experimental results are reported in Section 7. Conclusions appear in Section 8.

2 Translation

Let Σ be a signature consisting of constants, function and predicate symbols. We consider constants as function symbols of arity 0. We assume Σ is partitioned into Σ_c, Σ_s and $\Sigma_{s'}$ where Σ_c consists of symbols whose interpretation does not depend on a state, Σ_s consists of *current-state* symbols and $\Sigma_{s'}$ consists of *next-state* symbols. We assume that for every current-state symbol p in Σ_s there is a corresponding next-state symbol p' in $\Sigma_{s'}$ with the same arity as p and vice versa.

A transition system can be symbolically represented by three closed FOL formulas in, $trans$, and fin, respectively expressing facts about initial states, encoding the transition relation and expressing facts about final states. We assume that in and fin are formulas in $\Sigma_c \cup \Sigma_s$ and $trans$ is a formula in Σ. In order to adapt such a representation for bounded model checking in the EPR fragment we define the following transformation on formulas. First, we replace each current-state and next-state function or predicate symbol p of arity n in Σ_s and $\Sigma_{s'}$, respectively, with a *transient symbol* of arity $n+1$, which has an extra argument for representing transitions over states. Let Σ_t be the signature consisting of all transient symbols corresponding to symbols in Σ_s and by $\overline{\Sigma}$ the signature $\Sigma_c \cup \Sigma_t$.

Let $\mathtt{S}, \mathtt{S'}$ be two fresh variables. Let us define a translation $\mathcal{T}$ of Σ terms to $\overline{\Sigma}$ terms and Σ formulas to $\overline{\Sigma}$ formulas by induction as follows. Let $r_1, \ldots, r_n$ denote terms, let $t_i = \mathcal{T}(r_i)$ for all $i = 1, \ldots, n$, and let $\mathbf{r}, \mathbf{t}$ denote the sequences $r_1, \ldots, r_n$ and $t_1, \ldots, t_n$, respectively. Then:

– For any n-ary function or predicate symbol p define:

$$\mathcal{T}(p(\mathbf{r})) \stackrel{\text{def}}{=} \begin{cases} p(\mathbf{t}), & \text{if } p \in \Sigma_c; \\ p_t(\mathtt{S}, \mathbf{t}), & \text{if } p \in \Sigma_s; \\ p_t(\mathtt{S'}, \mathbf{t}), & \text{if } p \in \Sigma_{s'}. \end{cases}$$

– $\mathcal{T}(F_1 \wedge F_2) \stackrel{\text{def}}{=} \mathcal{T}(F_1) \wedge \mathcal{T}(F_2)$, and similarly for other connectives in place of $\wedge$.

– $\mathcal{T}(\forall x\, F) \stackrel{\text{def}}{=} \forall x\, \mathcal{T}(F)$ and similarly for $\exists$ in place of $\forall$. Recall, we assume that the variables $\mathtt{S}, \mathtt{S'}$ are fresh, this implies that $\mathtt{S}, \mathtt{S'}$ are distinct from x.

For every closed formula F, the only free variables of $\mathcal{T}(F)$ are $\mathtt{S}$ and $\mathtt{S'}$. Moreover, if F uses no next-state symbols, as it is the case for the formulas in and fin, then $\mathcal{T}(F)$ does not contain $\mathtt{S'}$. Let us denote the formulas $\mathcal{T}(in)$, $\mathcal{T}(trans)$ and $\mathcal{T}(fin)$ respectively as $In(\mathtt{S})$, $Trans(\mathtt{S}, \mathtt{S'})$ and $Fin(\mathtt{S})$, parametrized by their free variables.

Let n be a non-negative integer. We define the *n-step unrolling of the transition system* as follows. Take new constants $s_0, \ldots, s_n$ and a new binary predicate $next$. The n-step unrolling of the transition system is defined as the set of formulas

$$In(s_0); Fin(s_n); \forall \mathtt{S}, \mathtt{S'}\, (next(\mathtt{S}, \mathtt{S'}) \rightarrow Trans(\mathtt{S}, \mathtt{S'}));$$
$$next(s_0, s_1); next(s_1, s_2); \ldots next(s_{n-1}, s_n).$$

Theorem 1. *There exists an n-step computation of the transition system leading from a state satisfying in to a state satisfying fin if and only if the n-step unrolling of the transition system is satisfiable.*

Note that the n-step unrolling of the system contains only one copy of the transition relation $Trans$. This explains the name we have chosen for our encoding of BMC into FOL: *BMC1*. It stands for *BMC with one copy of the transition relation*. Unlike in SAT-based BMC [4], there is no need to create a new copy of the transition relation for each unrolling bound.

Unfortunately, this translation is not EPR preserving if the original system contains constants, which become transient functions of states after the translation. Such transient functions are essential in memory specifications representing, e.g., addresses which change during transitions. In later sections we show how to restore the EPR representation.

3 Encoding Hardware Specifications into FOL

Let us show how to encode a hardware verification problem into EPR, using a simple yet realistic word-level hardware design shown in Fig. 1. This example contains typical word-level components: a memory, bit-vectors and addresses.

The memory `mem` has 32 rows and 64 columns, each cell containing one bit. When both the write enable signal `wren` and the clock signal `clock` are true, bits 0 to 63 (written as $[63:0]$ in hardware notation) of the bit-vector `wrdata`$[63:0]$ are written into `mem` at the address given in the bit-vector `wraddr`$[5:0]$. In order to prevent read and write from happening simultaneously, only if `clock` is false and the read enable signal `rden` is true, the value of the memory at address `rdaddr`$[5:0]$ is read into the bit-vector `rddata`$[63:0]$.

The circuit also contains a cache line in the 64 bit bit-vector `cacheline`$[63:0]$, the component `sel` that compares two bit-vectors bit-wise and a multiplexing device `mux` which selects one of its inputs depending on the output value of `sel`. The final output of the circuit is either the bit-wise negated bit-vector `rddata`$[63:0]$ if `wraddr`$[63:0]$ and `rdaddr`$[63:0]$ are equal, or the bit-vector `cacheline`$[63:0]$ otherwise.

3.1 Encoding of Bit-Vectors and Memories

With any bit-vector we associate a binary predicate. For example, atom $\mathtt{wrdata}(\mathtt{S}, \mathtt{B})$ denotes the Boolean value of bit `B` in the write data vector `wrdata` in state `S`. Similarly, with a memory `mem` we associate a ternary predicate `mem`, where an atom $\mathtt{mem}(\mathtt{S}, \mathtt{A}, \mathtt{B})$ denotes the Boolean value of `mem` in row `A` and column `B`, in state `S`.

In our encoding, there are bit-vectors that in addition to this predicate representation also require a functional representation, we call them *functional bit-vectors*. There are

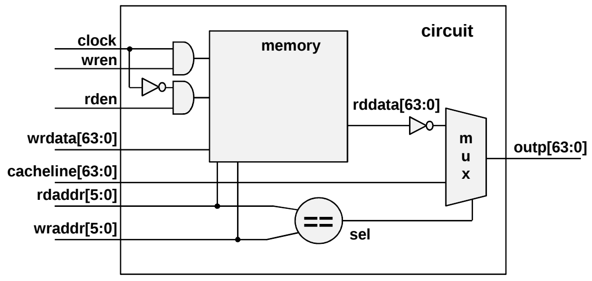

Fig. 1. Running example: a word-level hardware design

two main sources of such bit-vectors. The first consists of bit-vectors representing addresses which are used as arguments in memory predicates. The second consists of bit-vectors that are used in comparisons such as in the `sel` component in our running example in Fig. 1. With a functional bit-vector, in addition to associating a binary predicate over states and bit-indexes, we associate a function over states. We represent the value of an address `addr` in state `S` by the term $\mathtt{addrFunc(S)}$. Thus the value of `mem` at address `addr` and bit (column) `B` is represented by the atom $\mathtt{mem(S, addrFunc(S), B)}$. We use similar notation for functional bit-vectors which are not addresses.

We often need to refer to particular bits of a bit-vector. We use the constant $\mathtt{bitInd}_i$ to denote the i-th bit. Similarly, we use the constant $\mathtt{s}_j$ to denote the j-th state. Thus, the atom $\mathtt{mem}(\mathtt{s}_0, \mathtt{addrFunc}(\mathtt{s}_0), \mathtt{bitInd}_5)$ represents the value of bit 5 at row $\mathtt{addrFunc}(\mathtt{s}_0)$ in memory `mem` in state $\mathtt{s}_0$.

The reader may have noticed that bit-vector width and memory dimension information is not directly encoded. For example, a predicate representing a bit-vector of width 64 does not carry the width information. This loss of information is recovered, if necessary, when specifying bit-vector operations, which will be explained below. Similarly, the functions associated with addresses and other functional bit-vectors do not carry the width information of the corresponding bit-vector.

For addresses, the only information we need is whether they are equal. For two addresses `wraddr` and `rdaddr`, we can axiomatize this using:

$$\begin{aligned}&\mathtt{wraddrFunc(S)} = \mathtt{rdaddrFunc(S)} \leftrightarrow \\ &\quad(\mathtt{wraddr}(\mathtt{S},\mathtt{bitInd}_5) \leftrightarrow \mathtt{rdaddr}(\mathtt{S},\mathtt{bitInd}_5)\ \wedge \ldots \wedge \\ &\quad\ \mathtt{wraddr}(\mathtt{S},\mathtt{bitInd}_0) \leftrightarrow \mathtt{rdaddr}(\mathtt{S},\mathtt{bitInd}_0)).\end{aligned} \tag{1}$$

We assume that free variables are implicitly universally quantified.

By using the predicate $\mathtt{less}_k$, which defines bit-indexes in the range of $[0, k-1]$, the above formulas can be written more concisely, without referring explicitly to all of the bits 5 to 0, as follows:

$$\begin{aligned}&\mathtt{wraddrFunc(S)} = \mathtt{rdaddrFunc(S)} \leftrightarrow \\ &\quad\forall \mathtt{B}(\mathtt{less}_6(\mathtt{B}) \rightarrow (\mathtt{wraddr(S,B)} \leftrightarrow \mathtt{rdaddr(S,B)})).\end{aligned} \tag{2}$$

We axiomatize the $\mathtt{less}_k$ predicates explicitly as in [13]:

$$\mathtt{less}_k(x) \leftrightarrow (x = \mathtt{bitInd}_0 \vee \ldots \vee x = \mathtt{bitInd}_{k-1}). \tag{3}$$

In addition we need axioms stating that all bit-indexes are different:

$$\texttt{bitInd}_i \neq \texttt{bitInd}_j \text{ for } 1 \leq i < j \leq n, \text{ where } n \text{ is the size of the bit-index domain.} \quad (4)$$

In practical hardware examples the domain size of bit-indexes is in order of thousands and therefore adding such axioms can be a bottleneck. Fortunately, the number of different $\texttt{less}_k$ predicates is usually not very big compared to the number of bit-indexes. Since equality over bit-indexes occurs only in formulas such as (3) we can replace axioms (4) by axioms:

$$\begin{array}{l} \texttt{less}_k(\texttt{bitInd}_j) \ // \ \mathit{if}\ j < k \\ \neg\texttt{less}_k(\texttt{bitInd}_j) \ // \ \mathit{otherwise}, \end{array} \quad (5)$$

where $\texttt{less}_k$ occurs in the problem instance.

In our encoding we also frequently use predicates of the form $\texttt{range}_{[m,k]}$ which defines bit-indexes in the range $[m, k]$. The range predicates can be defined either using less predicates $(\texttt{less}_k(\texttt{B}) \wedge \neg\texttt{less}_m(\texttt{B})) \leftrightarrow \texttt{range}_{[m,k-1]}(\texttt{B})$ or explicitly:

$$\texttt{range}_{[m,k]}(x) \leftrightarrow (x = \texttt{bitInd}_m \vee \ldots \vee x = \texttt{bitInd}_k). \quad (6)$$

Explicit representation of range predicates has several advantages: i) we can replace $\texttt{less}_k$ predicates using $\texttt{range}_{[0,k-1]}$ predicates, ii) in our encoding, after such replacement all non-ground occurrences of the range predicates will be negative, iii) based on ii), instead of having both positive and negative axioms for $\texttt{range}_{[0,k-1]}$ as we have for $\texttt{less}_k$ in (5), it is sufficient to introduce positive axioms: $\texttt{range}_{[m,k]}(\texttt{bitInd}_j)$ for $m \leq j \leq k$. Wlog we can assume that range predicates do not overlap: we can factor out intersections by introducing corresponding range predicates. This can enable further higher level reasoning at the interval level. Another way of representing ranges is using integer arithmetic. Experiments with these representations are presented in Section 7.1.

In general, we may need to refer to constant addresses as well, say row 0 specified as bit-vector $b000000$ (using 6 bits). This constant address is represented in our encoding using a term, denoted by $t000000$. Note that since a constant address does not depend on the state of the hardware, we do not need to treat it as a function on states – it is a constant. We can define when $\texttt{wraddrFunc}(\texttt{S})$ and $t000000$ refer to the same row by

$$\begin{array}{l} \texttt{wraddrFunc}(\texttt{S}) = t000000 \leftrightarrow \\ \quad (\texttt{wraddr}(\texttt{S}, \texttt{bitInd}_5) \leftrightarrow \mathit{false} \wedge \ldots \wedge \texttt{wraddr}(\texttt{S}, \texttt{bitInd}_0) \leftrightarrow \mathit{false}). \end{array} \quad (7)$$

Thus, by $\texttt{mem}(\texttt{s}_0, t000000, \texttt{bitInd}_5)$, we can refer to the value of bit 5 in row 0 of $\texttt{mem}$, in state $\texttt{s}_0$. In Section 4.1 we consider different approaches for defining equality over functional address.

3.2 Encoding of Bit-Vector Operations

In our running example bit-vectors $\texttt{wraddr}$ and $\texttt{rdaddr}$ are compared by $\texttt{sel}$, and therefore we treat them as functional bit-vectors. We define $\texttt{sel}$ as follows:

$$\texttt{sel}(\texttt{S}) \leftrightarrow \texttt{wraddrFunc}(\texttt{S}) = \texttt{rdaddrFunc}(\texttt{S}), \quad (8)$$

which in predicate representation can be rewritten as:

$$\texttt{sel}(\texttt{S}) \leftrightarrow \forall \texttt{B}(\texttt{less}_6(\texttt{B}) \rightarrow (\texttt{wraddr}(\texttt{S}, \texttt{B}) \leftrightarrow \texttt{rdaddr}(\texttt{S}, \texttt{B}))). \quad (9)$$

Similarly, the logic of `outp` can then be defined as follows:

$$\begin{aligned}&\mathtt{sel}(\mathtt{S}) \rightarrow \forall \mathtt{B}(\mathtt{less}_{64}(\mathtt{B}) \rightarrow (\mathtt{outp}(\mathtt{S},\mathtt{B}) \leftrightarrow \neg \mathtt{rddata}(\mathtt{S},\mathtt{B})))\\ &\neg \mathtt{sel}(\mathtt{S}) \rightarrow \forall \mathtt{B}(\mathtt{less}_{64}(\mathtt{B}) \rightarrow (\mathtt{outp}(\mathtt{S},\mathtt{B}) \leftrightarrow \mathtt{cacheline}(\mathtt{S},\mathtt{B}))).\end{aligned} \tag{10}$$

3.3 Encoding of the Transition Relation

To express the next state functions, we use `S` to denote the current state and `S'` to denote the next state. The predicate $next(\mathtt{S},\mathtt{S'})$ denotes the transition relation, and its axiomatization is described next on our running example (Fig. 1). Recall that in our design, write is enabled when `wren` $\wedge$ `clock` holds, and read is enabled when `rden` $\wedge$ $\neg$`clock` holds. The transition relation for the write and read operations are therefore written as follows:

$$\begin{aligned}
&\forall \mathtt{S},\mathtt{S'}\,(next(\mathtt{S},\mathtt{S'}) \rightarrow \qquad \text{// write is enabled}\\
&\quad \forall \mathtt{A}((\mathtt{clock}(\mathtt{S'}) \wedge \mathtt{wren}(\mathtt{S'}) \wedge \mathtt{A} = \mathtt{wraddrFunc}(\mathtt{S'})) \rightarrow\\
&\qquad \forall \mathtt{B}(\mathtt{range}_{[0,63]}(\mathtt{B}) \rightarrow (\mathtt{mem}(\mathtt{S'},\mathtt{A},\mathtt{B}) \leftrightarrow \mathtt{wrdata}(\mathtt{S'},\mathtt{B})))));\\
&\forall \mathtt{S},\mathtt{S'}\,(next(\mathtt{S},\mathtt{S'}) \rightarrow \qquad \text{// write is disabled}\\
&\quad \forall \mathtt{A}(\neg(\mathtt{clock}(\mathtt{S'}) \wedge \mathtt{wren}(\mathtt{S'}) \wedge \mathtt{A} = \mathtt{wraddrFunc}(\mathtt{S'})) \rightarrow\\
&\qquad \forall \mathtt{B}(\mathtt{range}_{[0,63]}(\mathtt{B}) \rightarrow (\mathtt{mem}(\mathtt{S'},\mathtt{A},\mathtt{B}) \leftrightarrow \mathtt{mem}(\mathtt{S},\mathtt{A},\mathtt{B})))));\\
&\forall \mathtt{S},\mathtt{S'}\,(next(\mathtt{S},\mathtt{S'}) \rightarrow \qquad \text{// read is enabled}\\
&\quad \forall \mathtt{A}((\neg\mathtt{clock}(\mathtt{S'}) \wedge \mathtt{rden}(\mathtt{S'}) \wedge \mathtt{A} = \mathtt{rdaddrFunc}(\mathtt{S'})) \rightarrow\\
&\qquad \forall \mathtt{B}(\mathtt{range}_{[0,63]}(\mathtt{B}) \rightarrow (\mathtt{rddata}(\mathtt{S'},\mathtt{B}) \leftrightarrow \mathtt{mem}(\mathtt{S'},\mathtt{A},\mathtt{B})))));\\
&\forall \mathtt{S},\mathtt{S'}\,(next(\mathtt{S},\mathtt{S'}) \rightarrow \qquad \text{// read is disabled}\\
&\quad (\neg(\neg\mathtt{clock}(\mathtt{S'}) \wedge \mathtt{rden}(\mathtt{S'})) \rightarrow\\
&\qquad \forall \mathtt{B}(\mathtt{range}_{[0,63]}(\mathtt{B}) \rightarrow (\mathtt{rddata}(\mathtt{S'},\mathtt{B}) \leftrightarrow \mathtt{rddata}(\mathtt{S},\mathtt{B}))))).
\end{aligned} \tag{11}$$

Note that in the formulas above, we have assumed that the cells in the memory `mem` and the read data `rddata` are modeled as latches rather than flip-flops. We assume that the next-state value of a latch is updated by the next-state value of its input if the next-state value of its enable logic is true. For flip-flops, the next-state value is updated by the current-state value of the input data, when the current-state value of the enable logic is true.

3.4 Encoding of Initial and Final State Constraints

If in the initial state $\mathtt{s}_0$ the memory `mem` is reset (with 0 in each cell), we write this condition as follows:

$$\forall \mathtt{A},\mathtt{B}(\mathtt{less}_{64}(\mathtt{B}) \rightarrow \neg \mathtt{mem}(\mathtt{s}_0,\mathtt{A},\mathtt{B})). \tag{12}$$

If memory cells are initialized with different values, we write the initial state constraints for it bit-wise. For example, the next formula states that the value of `mem` in row 0 and column 5 is 0.

$$\neg \mathtt{mem}(\mathtt{s}_0, t000000, \mathtt{bitInd}_5). \tag{13}$$

Initial state values for bit-vectors are specified similarly.

Suppose an assertion prop that we want to verify states that the values of outp and cacheline coincide. We write prop as:

$$\mathtt{prop}(\mathtt{S}) \leftrightarrow \forall \mathtt{B}(\mathtt{less}_{64}(\mathtt{B}) \rightarrow (\mathtt{outp}(\mathtt{S},\mathtt{B}) \leftrightarrow \mathtt{cacheline}(\mathtt{S},\mathtt{B}))). \tag{14}$$

and in order to show correctness of the design we try to refute the negated conjecture

$$\exists \mathtt{S}\ \neg \mathtt{prop}(\mathtt{S}). \tag{15}$$

3.5 Encoding the BMC Problem

Finally, having the encoding for transition relation, initial and final state constraints, we encode the BMC problem as it is presented in Section 2. The issue we are left with is restoring the EPR encoding which we consider in the next section.

To summarize our word-level encoding, note that, unlike the word-level encoding scheme to EPR in [13], and similar to the *bit-level* encoding in [23], the unrolling bound is explicitly represented. There is no need for unrolling in the sense of BMC [4] in order to refer to a bit-vector or a memory at a desired bound. For this reason, our encoding has a higher potential for abstraction (i.e., removing irrelevant parts of the assertion formula before passing it to a solver engine), and we can use both the initial and final state constraints to simplify the assertion formula with constant propagation and other advanced pre- or in-processing techniques. Sequential ATPG [16] also avoids unrolling via backward time-frame expansion, however, it cannot efficiently use the initial state constraints to simplify the assertion formula. Thus we can combine the strengths of forward and backward reachability analysis in one algorithm. Another advantage of our approach is that, thanks to explicit treatment of time, we can infer invariant properties of the system by pure first-order reasoning without using any form of induction, in the spirit of [20]. A new approach that avoids explicit unrolling for incremental SAT-based MC is proposed in [5]; it is unclear at present how this approach relates to ours.

4 Back to EPR

There are two important problems to be solved in order to obtain an EPR encoding of BMC1: one is with *functional bit-vectors* which in BMC1 are functions of states, and the other is with *naming* of subformulas that result in non-EPR after clausification. The two subsections describe a way out – back to EPR: We solve the first problem during the encoding, by proposing a smart way to deal with addresses; functional bit-vectors occurring in bit-vector comparison can be treated similarly. We solve the second problem as part of the pre-processing of the *entire problem instance* and by improving the clausification (this requires the global view of the entire problem instance).

4.1 Unrolling Addresses

Since non-constant addresses become unary functions in our encoding, the BMC1 problem instances for hardware designs with bit-vectors and memories are outside of the EPR fragment. In order to recover an EPR encoding we apply the following transformation. Consider a non-constant address addr, which is transformed into a unary function $\texttt{addrFunc(S)}$, denoting the value of $\texttt{addr}$ in state $\texttt{S}$. We introduce new constants $\texttt{addr}_0, \ldots, \texttt{addr}_n$ where n is the unrolling bound, and a binary predicate $\texttt{Assoc}_{\texttt{addr}}(x, y)$. We add axioms

$$\texttt{Assoc}_{\texttt{addr}}(s_0, \texttt{addr}_0) \wedge \ldots \wedge \texttt{Assoc}_{\texttt{addr}}(s_n, \texttt{addr}_n),$$

which associate each constant $\texttt{addr}_i$ with the state s_i for $0 \leq i \leq n$. We also transform each formula $\phi[\texttt{addr}(x)]$ into a formula

$$\forall y(\texttt{Assoc}_{\texttt{addr}}(x, y) \rightarrow \phi[y]).$$

Consider bit-vectors of a fixed length, say $k > 0$. We introduce a binary predicate $\mathrm{Val}(x, y)$ which defines values of functional bit-vectors, e.g., the value of $\mathrm{Val}(b, i)$ represents the value of the bit-vector b at the index i. We use a unary predicate $\mathcal{A}_k(x)$ to represent the set of all functional bit-vectors of length k used in the hardware model. We define equality between two functional bit-vectors of length k as follows.

$$\begin{aligned} &\forall x, y\, [\mathcal{A}_k(x) \wedge \mathcal{A}_k(y) \rightarrow \\ &\quad (x = y \leftrightarrow \forall \texttt{B}(\texttt{range}_{[0,k-1]}(\texttt{B}) \rightarrow (\mathrm{Val}(x, \texttt{B}) \leftrightarrow \mathrm{Val}(y, \texttt{B}))))]. \end{aligned} \tag{16}$$

We thus replace address equality axioms like (1) and (7) discussed earlier by (16). After Skolemizing (16) and some simplifications we obtain:

$$\begin{aligned} &\forall x, y\, [\mathcal{A}_k(x) \wedge \mathcal{A}_k(y) \rightarrow \\ &\quad (x = y \vee (\texttt{range}_{[0,k-1]}(\mathit{df}_k(x, y)) \wedge (\mathrm{Val}(x, \mathit{df}_k(x, y)) \leftrightarrow \neg\mathrm{Val}(y, \mathit{df}_k(x, y))))))]. \end{aligned} \tag{17}$$

Informally, this formula states that if two bit-vectors are different then the Skolem function $\mathit{df}_k(x, y)$ gives an index, within the range $[0, k-1]$, witnessing the difference. Unfortunately, (17) is outside of the EPR class. In order to get back to an EPR encoding, we represent the function df_k using a new predicate Df_k as follows.

$$\forall x, y\, [\mathit{Df}_k(x, y, 0) \vee \ldots \vee \mathit{Df}_k(x, y, k-1)]. \tag{18}$$

$$\begin{aligned} &\forall x, y, \texttt{B}\ [\mathcal{A}_k(x) \wedge \mathcal{A}_k(y) \wedge \mathit{Df}_k(x, y, \texttt{B}) \rightarrow \\ &\quad (x = y \vee (\mathrm{Val}(x, \texttt{B}) \leftrightarrow \neg\mathrm{Val}(y, \texttt{B}))))]. \end{aligned} \tag{19}$$

In addition, for each constant c of length k representing an address, such as addresses $\texttt{addr}_i$ discussed above, or Skolem constants representing addresses, we need an axiom $\mathcal{A}_k(c)$. In the many-sorted setting the formulas can be simplified assuming we have a sort for all bit-vectors of length k.

Another approach to eliminating the function df_k is to introduce for each pair of address constants a_i, a_j, an index constant $d_{i,j}$ which witnesses the difference of a_i and a_j if they are different in the interpretation. We axiomatize this as follows. Let $a_0, \ldots, a_m$ be the list of all address constants of length k occurring in our problem, including the Skolem constants. Then define the following set of axioms, where $0 \leq i < j \leq m$:

$$a_i = a_j \vee (\texttt{range}_{[0,k-1]}(d_{ij}) \wedge (\mathrm{Val}(a_i, d_{ij}) \leftrightarrow \neg\mathrm{Val}(a_j, d_{ij}))). \tag{20}$$

This alternative encoding can be used when we have a relatively small number of bit-vector constants but of a large bit-vector size.

As an example, the next-state axiom for the write operation (11) will be:

$$\begin{array}{l} \forall \mathtt{S}, \mathtt{S'} \, (next(\mathtt{S}, \mathtt{S'}) \rightarrow \qquad \text{// write is enabled} \\ \;(\forall y (\mathtt{Assoc}_{\mathtt{wraddr}}(\mathtt{S'}, y) \rightarrow \\ \;\;(\forall \mathtt{A}((\mathtt{clock}(\mathtt{S'}) \wedge \mathtt{wren}(\mathtt{S'}) \wedge \mathtt{A} = y) \rightarrow \\ \;\;\;(\forall \mathtt{B}(\mathtt{range}_{[0,63]}(\mathtt{B}) \rightarrow (\mathtt{mem}(\mathtt{S'}, \mathtt{A}, \mathtt{B}) \leftrightarrow \mathtt{wrdata}(\mathtt{S'}, \mathtt{B}))))))))). \end{array} \tag{21}$$

Constant addresses which are non-transient, are represented as constants rather than functions of states, thus associated address constants for them are not introduced.

4.2 Pre-processing and Clausification

Consider the defining axiom (14) for the property prop. If we apply the standard clausification algorithm to (14) then we obtain a non-EPR formula due to the negative occurrence of the universal quantifier in the '$\leftarrow$' direction of the outer equivalence. Our first observation comes from a well-known idea used in the optimized CNF transformation: if the defined predicate occurs only positively in the rest of the formula then we can replace the outer equivalence in the definition with '$\rightarrow$' implication. The new simplified axiom can now be safely transformed into an EPR formula. Unfortunately, this is not always the case in the verification examples we have tried. On the other hand such negative occurrences are usually limited. For example, assume that prop occurs negatively only in a negated conjecture (15). Our next idea is to *inline* the definition of prop into such negative occurrences of prop, i.e., replace prop by its definition. After inlining (14) into (15) we obtain:

$$\exists \mathtt{S}(\exists \mathtt{B}(\mathtt{less}_{64}(\mathtt{B}) \wedge \neg(\mathtt{outp}(\mathtt{S}, \mathtt{B}) \leftrightarrow \mathtt{cacheline}(\mathtt{S}, \mathtt{B})))). \tag{22}$$

Let us note that after inlining we have i) obtained an equivalent formula and ii) eliminated one negative occurrence of the defined predicate. In this way we can remove all negative occurrences of the defined predicate. We can see that Skolemization applied to the new formula (22) produces an EPR formula. Likewise, since we removed all negative occurrences of prop we can now simplify the definition of prop as above, which after Skolemization also becomes an EPR formula. It is still possible (albeit infrequent in practice) that inlining fails to restore EPR classification or results in a large increase in the formula size. In these cases we can apply techniques as in Section 4.1 to the Skolem functions, restoring EPR. For further *EPR-restoring* pre-processing and clausification techniques we refer to [15]. Inlining and EPR-restoring pre-processing is implemented in Vampire's clausifier. [3]

5 Other Optimizations in the Encoding

Writing Next-state Formulas for All States. Hardware is driven into its initial state (or states) after applying a reset sequence. Therefore the initial state is such that, for each latch, if its enable is true, its input and output have the same value. We use this assumption to simplify the next-state functions for latches in this case (i.e., when the latch is

[3] Available at http://www.vprover.org/

updated). For example, instead of the next-state axioms (21) for the write operation (which is a latch vector) we write

$$\begin{array}{l}\forall \mathtt{S}(\forall y(\mathtt{Assoc}_{\mathtt{wraddr}}(\mathtt{S}, y) \rightarrow \qquad // \text{ write is enabled} \\ \quad (\forall \mathtt{A}((\mathtt{clock}(\mathtt{S}) \wedge \mathtt{wren}(\mathtt{S}) \wedge \mathtt{A} = y) \rightarrow \\ \quad\; (\forall \mathtt{B}(\mathtt{range}_{[0,63]}(\mathtt{B}) \rightarrow (\mathtt{mem}(\mathtt{S}, \mathtt{A}, \mathtt{B}) \leftrightarrow \mathtt{wrdata}(\mathtt{S}, \mathtt{B})))))))).\end{array}$$

The latter formula is much easier for theorem provers: it can be applied to any state constant, no need to (constructively) derive that it is a next-state for some other state.

When a latch retains its previous value, we still need to refer to both current and next states in the next-state function. A similar optimization is made for memories whose cells are implemented as latches. However, this optimization does not apply to flip-flops and memories whose cells are implemented as flip-flops.

Abstracting Bit-Vector Widths. Consider a subformula quantified over a bit-index variable within a range, say

$$\forall \mathtt{B}(\mathtt{range}_{[0,63]}(\mathtt{B}) \rightarrow (bv_1(\mathtt{S}, \mathtt{B}) \leftrightarrow \neg bv_2(\mathtt{S}, \mathtt{B}))). \tag{23}$$

Let us call such a subformula a *range* subformula, of range $[63 : 0]$. Such subformulas might also contain free occurrences of address variables.

If a range subformula occurs positively and the full ranges of all involved bit-vectors coincide with the range of the formula, then we transform the subformula into a simpler one, by omitting the relevant range of B. For example, positive occurrences of subformula (23) will be transformed into:

$$\forall \mathtt{B}(bv_1(\mathtt{S}, \mathtt{B}) \leftrightarrow \neg bv_2(\mathtt{S}, \mathtt{B})). \tag{24}$$

The latter formula is easier for theorem provers, since in order to use it in the inference one does not need to know the range of B.

Adding Sorts. Sorts (types) are now supported in the TPTP standard for FOL problems. We work with three sorts: addresses, states, and bit-indexes. This simplifies the encoding, makes solving faster, and improves the representation of models (counter-examples) since constants of different sorts are not mixed any more in the models.

6 Incremental Bounds

Given the BMC1 encoding of a transition system it is desirable to search for falsifying paths incrementally, bound after bound, avoiding repeated computations. We have implemented such an incremental algorithm in our instantiation-based automated reasoning system iProver [19]. In a nutshell, iProver generates instances of the first-order input clauses in a smart way in an attempt to approximate a ground model. The ground reasoning is delegated to a solver for propositional satisfiability, currently MiniSAT [11].

iProver supports incrementality based on propositional unit assumptions as follows. We can add and retract propositional unit assumptions without repeating calculations which were not based on these assumptions. For each bound k we introduce a propositional variable p_k and use unit assumptions to activate and deactivate bound dependent axioms. For example, consider a bound k and bound dependent axioms for reachable states:

$$\mathrm{RState}(s_0) \wedge \dots \wedge \mathrm{RState}(s_k). \tag{25}$$

$$p_k \rightarrow \forall x(\mathrm{RState}(x) \rightarrow x = s_0 \vee \ldots \vee x = s_k). \tag{26}$$

Then we can add the unit assumption p_k which activates the state axiom (26) above. Optionally, we can also add axioms $\neg p_0, \ldots, \neg p_{k-1}$ which would be used by the SAT solver to ignore all state axioms for the previous bounds $1, \ldots, k-1$. The only other bound dependent axioms are those defining the $next$ predicate (see Section 2) and unrolling of addresses (see Section 4.1). Let us note that specifications of the transition relation and initial/final state constraints are independent from the unrolling bounds and remain unchanged.

7 Benchmarks and Experimental Results

To evaluate our encoding we have generated two sets of benchmarks in TPTP format, where the first is already available as part of the TPTP library and the second is about to be released. We use iProver and Z3 in our experiments since these solvers performed best on the first set of examples (the results of other solvers are available as part of the TPTP library). Z3 has a dedicated EPR algorithm [25] and is also among the best on quantified SMT problems with arithmetic, while iProver has won the EPR division in several recent CASC competitions [27].

The first set of benchmarks was generated from a simple finite-state machine model called "Robot" and has been released as part of the TPTP library $v5.3.0$. The unrolling bounds were chosen so that the problem instances fit the required level of complexity for the competition, that is, each problem can be solved by at least one prover within the timeout. In the TPTP library the problems have been named HWV039 to HWV047 and are available in up to four variants in four forms each: as first-order formulas (FOF), in clausal normal form (CNF), as typed first-order formulas (TFF) and as typed first-order formulas with interpreted arithmetic symbols (TFA). In the last form, we treated bit-indexes as integers and modeled `less` and `range` predicates with the $<$ predicate.

The second set of benchmarks originate from real-life hardware verification problems on Intel designs containing memories. We are in the process of releasing scrambled versions of these benchmarks into the TPTP library. For the evaluation in this paper we focus on the real-life benchmark problems which are challenging due to a large number of word-level components.

7.1 Comparison of Encodings of Bit-ranges

We evaluated three different encodings of bit-ranges on the second set of industrial BMC problems. In the first two encodings ranges are modeled with the $\mathtt{range}_{[m,k]}$ and $\mathtt{less}_k$ predicates as described in Section 3.1. In the third encoding ranges are straightforwardly modeled using integer arithmetic.

We ran Z3 and iProver on problems unrolled to several bounds, on Intel Xeon Quad Core machines with 12 GB of memory with 20000s timeout. iProver accepts only CNF

Table 1. Different encodings of bit-indexes and bit-ranges

Problem (bound)	Z3			iProver	
	$\texttt{less}_k$	$\texttt{range}_{[m,k]}$	arithmetic	$\texttt{less}_k$	$\texttt{range}_{[m,k]}$
BPB (bound 2)	—	—	—	42s	**41s**
BPB (bound 4)	—	—	—	**634s**	669s
DCC (bound 2)	78s	56s	**29s**	55s	79s
DCC (bound 4)	1204s	636s	**157s**	266s	238s
DCC (bound 6)	8540s	3396s	3512s	—	**1407s**
PMS (bound 2)	44s	1266s	**9s**	161s	163s
PMS (bound 4)	638s	**149s**	188s	1295s	1298s
PMS (bound 6)	2898s	5730s	**4564s**	—	—
PMS (bound 8)	12303s	**3062s**	—	—	—
ROB (bound 2)	—	—	—	**250s**	282s
SCD (bound 2)	167s	119s	178s	**15s**	**15s**
SCD (bound 4)	434s	316s	346s	**276s**	277s
SCD (bound 6)	886s	548s	699s	**635s**	**635s**
SCD (bound 8)	2037s	**1017s**	1497s	—	—

format with sorts and therefore for iProver the problems were clausified by Vampire; for Z3 we used problems in the original non-clausified sorted TFF and TFA formats.

Let us discuss experimental results shown in Table 1. The first and fourth problems can be solved by iProver and not by Z3; on the third and fifth problems, while iProver succeeded on some bounds, Z3 can reach higher bounds. We can observe that on higher unrolling bounds i) the performance of Z3 on the $\texttt{range}_{[m,k]}$ encoding is considerably better than on the $\texttt{less}_k$ encoding, ii) the range encoding is on a par with the arithmetic encoding. The arithmetic encoding is better on smaller bounds. iProver performs similarly on $\texttt{range}_{[m,k]}$ and $\texttt{less}_k$ encodings, with $\texttt{range}_{[m,k]}$ encoding reaching higher bounds on one problem. To conclude, the results suggest that the range encoding is a reasonable alternative to the arithmetic encoding and for future work we investigate ways of combining iProver and SMT solvers for better reasoning with ranges.

7.2 Comparison with SAT-Based BMC

In Table 2, we compare incremental SAT-based BMC [12] (column incBMC) with EPR-based incremental BMC1 (column incBMC1), on the second set of Intel benchmarks. The column #memories reports the number of memories in the cone of the property and their collective size in terms of number of memory cells (bits). Similarly, columns #BVs give the number of transient and constant bit-vectors, respectively (including bit-vectors of size 1), and their collective size in terms of bits. These two columns show how "word-level" the cone of the property really is, and the cone size. Columns incBMC and incBMC1 report the maximal bound reached by the respective algorithms within 10000 seconds time limit and unrolling bound limit 50.

We used Intel's SAT-based model checker to perform experiments with incremental BMC [18]. It has a state-of-the-art implementation of incremental BMC, and its SAT solver is especially tuned on problem instances originating from formal verification

Table 2. Comparing SAT-based incremental BMC with EPR-based BMC1

Problem	# Memories	# Transient BVs	# Const. BVs	incBMC	incBMC1
PMS1	8 (46080 bits)	1486 (6109 bits)	3 (47 bits)	2	**10**
SCD1	2 (16384 bits)	556 (1923 bits)	5 (45 bits)	4	**12**
SCD2	2 (16384 bits)	80 (756 bits)	3 (10 bits)	4	**14**
BPB2	4 (10240 bits)	550 (4955 bits)	6 (42 bits)	**50**	11
DCI1	32 (9216 bits)	3625 (6496 bits)	3 (9 bits)	**6**	4
DCC2	4 (8960 bits)	426 (1844 bits)	2 (2 bits)	8	**11**
DCC1	4 (8960 bits)	1827 (5294 bits)	5 (106 bits)	7	**8**
ROB2	2 (4704 bits)	255 (3479 bits)	26 (129 bits)	**50**	8

problems on Intel designs. In [18], it is shown that Intel SAT-based BMC tool is on a par with a leading academic model checking tool ABC [22].

Experimental results show that although on smaller memories SAT-based BMC is faster, when memory sizes increase, the advantage of EPR-based BMC1 becomes evident (rows in Table 2 are ordered in decreasing memory size). These results show that EPR-based model checking is a promising alternative to SAT-based model checking at word-level, scalable to industrial hardware designs. We refer to [13] for a comparison with SMT solvers supporting bit-vectors and arrays on unrolled BMC instances.

8 Conclusions and Future Work

In this paper we presented an encoding of bounded model checking at word level into the EPR fragment of first-order logic. The EPR-based encoding allows us to i) represent memories and bit-vectors at word-level and ii) succinctly specify the transition relation and state constraints, independently from the unrolling bound. Due to the presence of memories and bit-vectors, a naive encoding of the BMC problem into first-order logic would result in problems outside of the EPR fragment. We show how to restore the EPR encoding by introducing two methods: i) address unrolling and ii) definition inlining.

Another contribution of this work is in generating challenging benchmarks for first-order theorem provers based on real-life hardware designs used at Intel. We hope this will encourage further research into EPR-based model checking and EPR decision procedures. We have evaluated our encoding on these benchmarks using general purpose theorem provers iProver and Z3 which are not optimized for such problems. Our experimental results show that already at this stage our approach is scalable to industrial verification problems and on large memories can reach higher unrolling bounds compared to optimized SAT-based BMC.

There are many directions for future work, and we mention only few of them here. First, we intend to develop an abstraction-refinement approach to EPR-based model checking by providing the EPR solver with some bit-vector related information (e,g., the bit-vector width) via attributes. Second, we intend to investigate further incremental solving in the EPR-based BMC1 and how derived information on lower bounds can be exploited for reasoning on higher bounds. Finally, we believe that EPR-based BMC1 can be extended to efficiently work with arithmetic operations at word level, by building-in efficient arithmetic reasoning in the EPR decision procedures.

References

1. Abu-Haimed, H., D.L. Dill, S. Berezin. A refinement method for validity checking of quantified first-order formulas in hardware verification, FMCAD 2006.
2. Alberti F., Armando A., Ranise S. ASASP: Automated Symbolic Analysis of Security Policies, CADE 2011, LNCS 6803, 26-33, Springer.
3. Baumgartner P., A. Fuchs, C. Tinelli. Implementing the Model Evolution Calculus. Inter. J. on Artificial Intelligence Tools 15(1): 21-52, 2006.
4. Biere A., A. Cimatti, E. Clarke, Y. Zhu. Symbolic model checking without BDDs, TACAS 1999.
5. Bradley, A.R. SAT-based model checking without unrolling, VMCAI 2011.
6. Bradley, A.R., Manna Z., Sipma H.B. What's decidable about arrays? VMCAI 2006.
7. Brummayer R. D., Biere A. Boolector: An efficient SMT solver for bit-vectors and arrays, TACAS 2009.
8. Bryant R. E., Lahiri S. K., Seshia S. A.. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. CAV 2002.
9. Clarke, E.M., O. Grumberg, D.A. Peled. *Model Checking*, MIT Press, 1999.
10. Claessen K., N. Sörensson. New techniques that improve MACE-style model finding. Workshop on Model Computation (MODEL), 2003.
11. Eén N., N. Sörensson. An extensible SAT-solver. SAT 2003: 502-518.
12. Eén N., Sörensson N. Temporal induction by incremental SAT solving, ENTCS 89(4), 2003.
13. Emmer M., Khasidashvili Z., Korovin K., Voronkov A. Encoding Industrial Hardware Verification Problems into Effectively Propositional Logic FMCAD'10, 2010.
14. Ghilardi S., Nicolini E. , Ranise S., Zucchelli D. Decision procedures for extensions of the theory of arrays. Annals of Mathematics and Artificial Intelligence (AMAI), 2006.
15. Hoder K., Khasidashvili Z., Korovin K., Voronkov A. Preprocessing techniques for first-order clausification. In preparation.
16. Huang, S.-Y., K.-T., Cheng. *Formal Equivalence Checking and Design Debugging*, Kluwer, 1998.
17. Khasidashvili Z., Kinanah M., Voronkov A. Verifying Equivalence of Memories Using a First Order Logic Theorem Prover FMCAD'09, 2009.
18. Khasidashvili Z., Nadel A. Implicative simultaneous satisfiability and applications, HVC 2011.
19. Korovin, K. iProver–an instantiation-based theorem prover for first-order logic (system description), IJCAR, 2008.
20. Kovács, L., Voronkov, A. Finding Loop Invariants for Programs over Arrays Using a Theorem Prover, FASE 2009, 470-485, LNCS 5503, Springer, 2009.
21. Kroening D., Strichman O. *Decision Procedures*, Springer EATCS, 2008.
22. Mishchenko A., S. Chatterjee, R. Brayton, N. Een. Improvements to combinational equivalence checking, ICCAD 2006.
23. Navarro-Perez, J.A., Voronkov A. Encodings of Bounded LTL Model Checking in Effectively Propositional Logic, CADE 2007, 346-361, LNCS 4603, Springer.
24. Manolios P., S.K. Srinivasan, D. Vroon. Automatic memory reductions for RTL model verification, ICCAD 2006.
25. R. Piskac, L. de Moura, N. Bjørner: Deciding Effectively Propositional Logic Using DPLL and Substitution Sets. J. Autom. Reasoning, 2010.
26. Robinson A., Voronkov A. (editors). Handbook of Automated Reasoning, Elsevier and MIT Press 2001.
27. Sutcliffe G. The 5th IJCAR automated theorem proving system competition CASC-J5, AI Communications, vol. Volume 24(1), pp. 75-89, 2011.
28. Velev M. N., Bryant R. E. Verification of pipelined microprocessors by comparing memory execution sequences in symbolic simulation, ASIAN 1997.