Encoding Industrial Hardware Verification Problems into Effectively Propositional Logic

Moshe Emmer, Zurab Khasidashvili
Intel Israel Design Center
Haifa 31015, Israel
{memmer,zurabk}@iil.intel.com

Konstantin Korovin, Andrei Voronkov
School of Computer Science,
University of Manchester, UK
korovin@cs.man.ac.uk, andrei@voronkov.com

***Abstract*—Word-level bounded model checking and equivalence checking problems are naturally encoded in the theory of bit-vectors and arrays. The standard practice of deciding formulas of such theories in the hardware industry is either SAT- (using bit-blasting) or SMT-based methods. These methods perform reasoning on a low level but perform it very efficiently. To find alternative potentially promising model checking and equivalence checking methods, a natural idea is to lift reasoning from the bit and bit-vector levels to higher levels. In such an attempt, in [14] we proposed translating memory designs into the Effectively PRopositional (EPR) fragment of first-order logic.**

The first experiments with using such a translation have been encouraging but raised some questions. Since the high-level encoding we used was incomplete (yet avoiding bit-blasting) some equivalences could not be proved. Another problem was that there was no natural correspondence between models of EPR formulas and bit-vector based models that would demonstrate non-equivalence and hence design errors.

This paper addresses these problems by providing more refined translations of equivalence checking problems arising from hardware verification into EPR formulas. We provide three such translations and formulate their properties. All three translations are designed in such a way that models of EPR problems can be translated into bit-vector models demonstrating non-equivalence.

We also evaluate the best EPR solvers on industrial equivalence checking problems and compare them with SMT solvers designed and tuned for such formulas specifically. We present empirical evidence demonstrating that EPR-based methods and solvers are competitive.

I. Introduction

Use of theorem proving in hardware and software verification is not new. A first classification of the use of theorem proving in formal verification would be to divide it into Higher-Order Logic (HOL) and First-Order Logic (FOL) theorem proving. Because HOL theorem proving is highly interactive and requires from the user both an expertise in theorem proving and a good familiarity of the design (or program) under verification, the use of higher-order theorem proving in hardware verification is limited to particular styles of design for which no good fully-automatic verification methods exist.[1] Unlike HOL, there are highly efficient fully automatic FOL theorem provers, so the potential of FOL for a wider use in formal verification is significantly higher.

In this paper we are interested in equivalence checking and model checking problems in hardware verification involving decision procedures for bit-vectors and arrays. Such problems can be solved efficiently by Satisfiability Modulo Theories (SMT) [16] solvers [5], [6], [21]. More precisely, we are interested in problems in the theory of fixed-size bit-vectors and extensional arrays, known as the theory QF_AUFBV. It has also been shown [14] that such problems can be encoded into the Effectively Propositional (EPR) fragment of FOL, which is decidable and for which efficient FOL solvers exist [20], [15], [3]. The EPR fragment consists of first-order formulas which in clausal normal form contain no function symbols other than constants.

The current understanding (on which many experts in the field agree) is that FOL solvers are good at ”pure first-order problems” involving formulas with (interleaving and nested) universal and existential quantifiers, while SMT solvers are best at quantifier-free theories.[2] In this paper we set out to investigate the scalability of EPR solvers with different proof-calculi to real-life problems involving reasoning with bit-vectors and arrays, and comparing their performance with the best SMT solvers for the theory QF_AUFBV. We propose several sound and complete encodings of problems in this theory into EPR, and discuss and experimentally evaluate the advantages and disadvantages of different encodings. We also discuss and experimentally evaluate advantages and disadvantages of different proof calculi for FOL with respect to solving the EPR problems arising from industrial scale hardware verification.

To the best of our knowledge, no similar analysis was reported before. We find this analysis interesting and important especially because the significance of the EPR fragment in software and hardware verification has been realized only recently [17], [18], by showing that many interesting verification problems can be encoded in this fragment and can often be solved efficiently. We hope that the theoretical and experiential analysis reported in this work will help in cross-learning between the calculi and algorithms employed in SMT and FOL approaches, for the class of problems with bit-vectors and arrays.

This work is partially supported by EPSRC and the Royal Society.

[1]This by no means diminishes the importance of the use of HOL theorem proving in verification – in certain areas of verification it is indispensable.

[2]Few SMT solvers, like Z3, do support limited quantified theories; see [22] for further references.

In the next section, we recall a sound but incomplete encoding of problems with bit-vectors and arrays into EPR, as described in [14]. As a consequence of incompleteness, the powerful abstraction of the size of bit-vectors and arrays on which the encoding to EPR is based, is often the source of false counter-examples in verification. Debugging of verification failures is the main source of inefficiency in hardware design projects, and false failures (also called false negatives, caused by the nature of the verification tool or methodology rather than an actual bug in the design) are simply unacceptable. In Section III, we therefore propose several approaches to achieving the completeness of encoding to EPR, thereby eliminating the possibility of false negatives.

In Section III, and further in Section IV, we analyze the advantages and disadvantages of the proposed sound and complete encodings to EPR, and relate these to the strengths and weaknesses, relative to the problems we are interested in, of several important proof calculi employed in the best EPR solvers (the winners of recent theorem proving competitions in the EPR and other categories). Extensive experimental results comparing the performance of the best solvers for EPR problems with the performance of winning SMT solvers in the category of bit-vectors and arrays on hardware verification problems arising from real-life Intel micro-processor design are reported in Section V. The benchmarks were selected and organized carefully so to expose the strengths and weaknesses of different decision procedures, and their sensitivity to the nature of the benchmarks (such as the presence of extensive bit-level reasoning as opposed to really bit-vector level reasoning, the design style, the writing style of RTL, the nature of compilation of RTL and schematic descriptions into model-checking instances). Conclusions appear in Section VI.

II. The relational encoding

In this paper we consider the theory of fixed-size bit-vectors and extensional arrays. We assume that bit-vector arithmetic operators are synthesized (or bit-blasted) in the verification front-end, and the solver engines do not receive arithmetic operations in the expressions to solve.

For arrays, we assume the standard operations: read (or select), write (or store), and equality (if the array dimensions are the same), and the standard consistency and extensionality axioms [19], [7], [16]:

$$\begin{array}{l} mem\{i \leftarrow e\}(i) = e; \\ mem\{i \leftarrow e\}(j) = mem(j), \quad \text{if } j \neq i; \\ (\forall j : mem_1(j) = mem_2(j)) \rightarrow mem_1 = mem_2. \end{array}$$

The encodings of the theory of bit-vectors and arrays that we consider here are all refinements of an encoding proposed in [14], called the *relational encoding* (as opposed to the *algebraic encoding* that was also considered there and was shown not to scale on even small verification problems). To explain the relational encoding and to describe our contribution clearly, we choose to use as the running example slightly modified toy specification and implementation designs used as the running example in [14].

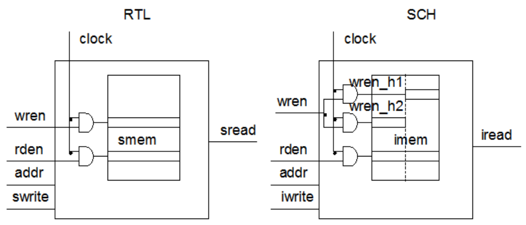

Fig. 1. Specification and Implementation memories

The toy designs are depicted in Figure 1. The specification design corresponds to the register-transfer level description (RTL), while the implementation design corresponds to its schematic implementation (SCH). The specification model contains a memory `smem` with 64 rows and 71 columns, its address bit-vector `addr` is of width 12, and it is used to pass the memory address for both write and read operations: bits `addr`$[5:0]$ are used for the write operation and bits `addr`$[11:6]$ are used for the read operation. Further, `swrite` and `sread` denote the write and read data vectors, respectively, of width 71, and `wren` and `rden` are the control bits enabling write and read operations, respectively. The bits `wren` and `rden`, as well as the clock `clock` and address `addr`, are shared between the specification and implementation designs. The implementation design has memory `imem` with the same dimensions as `smem`. [3] The write operation in the implementation model is different: bit-vector data `iwrite` is split into two parts during the write operation – `iwrite`$[70:36]$ and `iwrite`$[35:0]$. Each of the parts is written to the corresponding part of `imem`, so the implementation memory is shown split into two parts. Beyond the boundaries of the implementation memory unit the data is bitwise negated before being written to the implementation memory and after being read from it, so the write data `iwrite` (respectively, read data `iread`) in the implementation memory is the negation of the write (respectively, read) data in the specification memory.

With the relational encoding [14] of memory equivalence checking problems into EPR, any bit-vector b is considered as a relation on integers. Thus, for every integer k, it holds that $b(k)$ is true if and only if the kth bit of b is 1. If b does not have the kth bit, the relational encoding in [14] assumes that $b(k)$ is either true or false. Such a representation of bit-vectors is a powerful abstraction, since, instead of considering a bit-vector a mapping from a finite range of integers to booleans, as is done in the SMT theory QF_AUFBV, a bit-vector is now a mapping from *all* integers to booleans. The width of bit-vectors is thus abstracted away. In the relational encoding, a memory becomes a binary relation: the first argument denotes an address and the second a bit. For example, `imem`(a, k) denotes the value of the k-th bit of the element at the address

[3] Intel's logic extraction tool can identify memories and address decoders in the schematic models [14].

a in $\texttt{imem}$.

Let us now recall the relational encoding (as in [14]) of our running example. First, define the correspondence between the specification and the implementation designs as the conjunction of correspondence of the memories and of the read data.

$$\forall A \forall B(\texttt{imem}(A,B) \leftrightarrow \neg\texttt{smem}(A,B)). \tag{1}$$

$$\forall B(\texttt{iread}(B) \leftrightarrow \neg\texttt{sread}(B)). \tag{2}$$

The input write data correspondence is specified as follows:

$$\forall B(\texttt{iwrite}(B) \leftrightarrow \neg\texttt{swrite}(B)). \tag{3}$$

To be able to use the relational approach, one should identify bit-vectors in the design used as addresses and add equations enabling to decide when any pair of addresses is equal. For example, in our running example, we might need formulas describing when the term $\texttt{writeAddr}$ corresponding to $\texttt{addr}[5:0]$ is equal to the term $\texttt{readAddr}$ corresponding to $\texttt{addr}[11:6]$. Since the bit indexes involved in these two bit-vectors are different (in particular, are shifted), the corresponding bit-index constants $\texttt{bitInd}_0, \ldots, \texttt{bitInd}_{11}$ are introduced, and the following axiom is added:

$$\begin{aligned}&(\texttt{writeAddr} = \texttt{readAddr}) \leftrightarrow \\ &\quad((\texttt{addr}(\texttt{bitInd}_0) \leftrightarrow \texttt{addr}(\texttt{bitInd}_6)) \wedge \ldots \wedge \\ &\quad(\texttt{addr}(\texttt{bitInd}_5) \leftrightarrow \texttt{addr}(\texttt{bitInd}_{11}))).\end{aligned} \tag{4}$$

The transition relation for the specification memory is as follows, where the prime symbol $'$ is used to denote next-state variables:

$$\begin{aligned}&\forall A(\texttt{clock} \wedge \texttt{wren} \wedge A = \texttt{writeAddr} \rightarrow \\ &\qquad \forall B(\texttt{smem}'(A,B) \leftrightarrow \texttt{swrite}(B))); \\ &\forall A(\neg(\texttt{clock} \wedge \texttt{wren} \wedge A = \texttt{writeAddr}) \rightarrow \\ &\qquad \forall B(\texttt{smem}'(A,B) \leftrightarrow \texttt{smem}(A,B))).\end{aligned} \tag{5}$$

Splitting bit-vectors into parts is done by introducing predicates true on bits belonging to the LSB part. For the running example, predicate $\texttt{less}_{36}$ is introduced, intended to hold (only) on bits with numbers strictly less than 36. We also introduce propositional variables $\texttt{wren}_{h1}$ and $\texttt{wren}_{h2}$ for enabling writing into the two parts of the memory.

$$\begin{aligned}&\texttt{wren}_{h1} \leftrightarrow \texttt{wren} \wedge \texttt{clock}; \\ &\texttt{wren}_{h2} \leftrightarrow \texttt{wren} \wedge \texttt{clock}.\end{aligned} \tag{6}$$

The transition relation for the implementation memory is then given as follows:

$$\begin{aligned}&\forall A(\texttt{wren}_{h1} \wedge A = \texttt{writeAddr} \rightarrow \\ &\qquad \forall B(\texttt{less}_{36}(B) \rightarrow (\texttt{imem}'(A,B) \leftrightarrow \texttt{iwrite}(B)))); \\ &\forall A(\neg(\texttt{wren}_{h1} \wedge A = \texttt{writeAddr}) \rightarrow \\ &\qquad \forall B(\texttt{less}_{36}(B) \rightarrow (\texttt{imem}'(A,B) \leftrightarrow \texttt{imem}(A,B)))); \\ &\forall A(\texttt{wren}_{h2} \wedge A = \texttt{writeAddr} \rightarrow \\ &\qquad \forall B(\neg\texttt{less}_{36}(B) \rightarrow (\texttt{imem}'(A,B) \leftrightarrow \texttt{iwrite}(B)))); \\ &\forall A(\neg(\texttt{wren}_{h2} \wedge A = \texttt{writeAddr}) \rightarrow \\ &\qquad \forall B(\neg\texttt{less}_{36}(B) \rightarrow (\texttt{imem}'(A,B) \leftrightarrow \texttt{imem}(A,B)))).\end{aligned} \tag{7}$$

The definitions of the read operations for the specification memory are as follows.

$$\begin{aligned}&\texttt{clock} \wedge \texttt{rden} \rightarrow \forall B(\texttt{sread}'(B) \leftrightarrow \texttt{smem}(\texttt{readAddr}, B)); \\ &\neg(\texttt{clock} \wedge \texttt{rden}) \rightarrow \forall B(\texttt{sread}'(B) \leftrightarrow \texttt{sread}(B)).\end{aligned}$$

The definitions of the read operations for the implementation memory are similar; one should only replace $\texttt{sread}$ and $\texttt{smem}$ by $\texttt{iread}$ and $\texttt{imem}$.

III. Relational encodings eliminating spurious models

Unfortunately, as pointed out in [14], the powerful abstraction resulting from considering bit-vectors as functions on *all* integers comes in the expense of loosing the completeness of the encoding – false negatives (i.e, counter-examples that are not real) are possible. For example, if a, b, c represent bit-vectors of length 1, the formula

$$a = b \vee a = c \vee b = c \tag{8}$$

is valid, but it is not valid in the abstraction since its negation is satisfiable.

In order to avoid the possibility of false negatives (i.e., spurious models), we would like the relational encoding to become aware of the ranges of bit-vectors and arrays involved in circuit operation, and we would like to record this information in the translation. The main idea of the refined encoding – let us call it *range-aware relational encoding* to EPR – is that for every formula that we generate during the encoding, the range of bits in the involved bit-vectors or arrays is explicitly encoded in the formula using the less-predicates and bit-index constants (such as $\texttt{less}_{36}$ or $\texttt{bitInd}_5$). For this to work, we need to relate the less predicates introduced during the encoding with the bit-index constants introduced during the encoding. Note that there is no need to relate a bit-index $\texttt{bitInd}_k$ with a less predicate $\texttt{less}_n$ for many pairs (k, n): it might be irrelevant to capture the fact that

$$\begin{aligned}&\texttt{less}_n(\texttt{bitInd}_k) \quad \text{if } k < n; \\ &\neg\texttt{less}_n(\texttt{bitInd}_k) \quad \text{otherwise.}\end{aligned} \tag{9}$$

Next we discuss several ways to eliminate false negatives, and discuss the advantages and disadvantages of each approach.

A. Encoding 1: precise ranges

Let us first define range-predicates: For a pair of non-negative integers $n \leq m$, let us define

$$\texttt{range}_{[m,n]}(B) \leftrightarrow \texttt{less}_{m+1}(B) \wedge \neg\texttt{less}_n(B).$$

When equality between arrays is introduced, it should be guaranteed that there will be no bits beyond the range of the data on which the array equality will fail. For example, we write the invariant formula (1) for memories as

$$\forall A \forall B(\texttt{range}_{[70,0]}(B) \rightarrow (\texttt{imem}(A,B) \leftrightarrow \neg\texttt{smem}(A,B))).$$

When equality between bit-vectors of the same range is introduced, we explicitly restrict the corresponding equivalence of bits to the relevant bit-range. For example, we now write the formula (7) as

$$\begin{array}{l}\forall A(\mathtt{wren}_{h1} \wedge A = \mathtt{writeAddr} \rightarrow \\ \quad \forall B(\mathtt{range}_{[35,0]}(B) \rightarrow (\mathtt{imem}'(A,B) \leftrightarrow \mathtt{iwrite}(B)))); \\ \forall A(\neg(\mathtt{wren}_{h1} \wedge A = \mathtt{writeAddr}) \rightarrow \\ \quad \forall B(\mathtt{range}_{[35,0]}(B) \rightarrow (\mathtt{imem}'(A,B) \leftrightarrow \mathtt{imem}(A,B)))); \\ \forall A(\mathtt{wren}_{h2} \wedge A = \mathtt{writeAddr} \rightarrow \\ \quad \forall B(\mathtt{range}_{[70,36]}(B) \rightarrow (\mathtt{imem}'(A,B) \leftrightarrow \mathtt{iwrite}(B)))); \\ \forall A(\neg(\mathtt{wren}_{h2} \wedge A = \mathtt{writeAddr}) \rightarrow \\ \quad \forall B(\mathtt{range}_{[70,36]}(B) \rightarrow (\mathtt{imem}'(A,B) \leftrightarrow \mathtt{imem}(A,B)))).\end{array}$$

Similarly, instead of (2) we now write

$$\forall B(\mathtt{range}_{[70,0]}(B) \rightarrow (\mathtt{iread}(B) \leftrightarrow \neg\mathtt{sread}(B))).$$

We add axioms stating that bit-index terms corresponding to different indexes are not equal. For a less-predicate like $\mathtt{less}_{36}$, that has been introduced, we add the axiom

$$\begin{array}{l}\forall B(\mathtt{less}_{36}(B) \leftrightarrow \\ \quad (B = \mathtt{bitInd}_0) \vee \ldots \vee (B = \mathtt{bitInd}_{35})).\end{array} \tag{10}$$

How can the range-aware encoding solve the incompleteness of the relational encoding of [14]? With the relational encoding, the equation (8) is represented with the following formula, which is false already for 2-bit bit-vectors, say $a = 10, b = 00, c = 11$.

$$\begin{array}{l}(\forall B(a(B) \leftrightarrow b(B))) \vee \\ (\forall B(a(B) \leftrightarrow c(B))) \vee \\ (\forall B(b(B) \leftrightarrow c(B))).\end{array}$$

With the range-aware relational encoding, the same formula is represented by

$$\begin{array}{l}(\forall B(\mathtt{range}_{[0,0]}(B) \rightarrow a(B) \leftrightarrow b(B))) \vee \\ (\forall B(\mathtt{range}_{[0,0]}(B) \rightarrow a(B) \leftrightarrow c(B))) \vee \\ (\forall B(\mathtt{range}_{[0,0]}(B) \rightarrow b(B) \leftrightarrow c(B))).\end{array}$$

From the axiomatization (10) of the less and range predicates, we conclude that the above formula is equivalent to the one below, which is clearly true.

$$\begin{array}{l}(a(\mathtt{bitInd}_0) \leftrightarrow b(\mathtt{bitInd}_0)) \vee \\ (a(\mathtt{bitInd}_0) \leftrightarrow c(\mathtt{bitInd}_0)) \vee \\ (b(\mathtt{bitInd}_0) \leftrightarrow c(\mathtt{bitInd}_0)).\end{array}$$

Note that with the precise-ranges relational encoding the widths of bit-vectors and the dimensions of arrays are still abstracted away. This is different from the information specified to SMT solvers in the theory of fixed-size bit-vectors and extensional arrays. However, since our modeling of every bit-vector or array operations explicitly encodes the relevant ranges, the bit-vector width and array size information becomes redundant.

Theorem 1: The precise ranges encoding is sound and complete: an EPR formula obtained by the precise ranges encoding is satisfiable if and only if it is satisfiable over bit-vectors of the specified size.

B. Encoding 2: bit-index pre-instantiation

In the precise-ranges encoding, the axioms like (10) introduce many equalities between terms describing bit-indexes. Dealing with many such equalities may significantly slow down the EPR solvers. This is explained in the next section. The most straightforward way to avoid these equalities between the bit-index terms (and still retain the completeness) is to pre-instantiate all quantifiers ranging over bit-indexes with concrete index values. This is a meaningful alternative in the case where there is a lot of bit-wise reasoning, like in schematic models, and most of the bit-indexes introduced during pre-instantiation would have been introduced anyway with the precise-ranges approach.

This approach is sensitive to the amount of bit-index constants that will be introduced during pre-instantiation. In 64-bit based real-life micro-processor designs, there normally are no bit-vectors longer than around 71 bits (which consist of 64 bits of data and several other encryption bits and flags). However, because of the writing style of RTL and Schematic, and the way many RTL compilers work, often long vectors are created from nested structures. For example, in our toy example, if two different bit-vectors of width 6 were used for the write and read addresses instead of using a 12-bit vector addr, there will be no need to introduce bits $\mathtt{bitInd}_6, \ldots, \mathtt{bitInd}_{11}$ to the instance. Similarly, if the write and read data bit-vectors swrite and sread were defined as the LSB and MSB halves of a data bit-vector $\mathtt{sdata}[141:0]$, or as a structure with two fields $[70:0]$ swrite and $[70:0]$ iwrite, the functionality of the design would not change but the encoding with index pre-instantiation would force us to introduce extra bit-indexes $\mathtt{bitInd}_{71}, \ldots, \mathtt{bitInd}_{141}$. In our experiments below, we will see how introduction of bit-indexes caused by the RTL and SCH writing style and compilation of RTL and SCH into the model-checking instance can affect the solvers performance.

C. Encoding 3: Skolem predicates

We now introduce a smarter way to avoid introduction of equalities between bit-index terms as in the less predicate axioms like (10). Our approach can be seen as reasoning modulo a fixed domain of indexes and is inspired by approaches used in state-of-the-art finite model finders [1], [8].

First, note that the Skolemization of the invariant formulas (1) and (2) introduces Skolem constants. For example, let boolean variable readeq denote the truth value of the equality (2):

$$\mathtt{readeq} \leftrightarrow \forall B(\mathtt{range}_{[70,0]}(B) \rightarrow (\mathtt{iread}(B) \leftrightarrow \neg\mathtt{sread}(B))).$$

This formula is translated into a collection of following clauses, where $sk0$ is a fresh Skolem constant:

$$\begin{array}{l}\mathtt{readeq} \vee \neg\mathtt{iread}(sk0) \vee \neg\mathtt{sread}(sk0); \\ \mathtt{readeq} \vee \mathtt{iread}(sk0) \vee \mathtt{sread}(sk0); \\ \mathtt{readeq} \vee \neg\mathtt{less}_0(sk0); \\ \mathtt{readeq} \vee \mathtt{less}_{71}(sk0); \\ \mathtt{sread}(B) \vee \neg\mathtt{iread}(B) \vee \mathtt{less}_0(B) \vee \neg\mathtt{less}_{71}(B) \vee \ \mathtt{readeq}; \\ \mathtt{iread}(B) \vee \neg\mathtt{sread}(B) \vee \mathtt{less}_0(B) \vee \neg\mathtt{less}_{71}(B) \vee \ \mathtt{readeq}.\end{array} \tag{11}$$

On the clauses containing occurrences of Skolem constants, we perform the following transformation: For each Skolem constant $\mathtt{sk}_i$ we introduce a new unary predicate $\mathtt{skP}_i$ and the following axiom, where $k, \ldots, m$ is the index range corresponding to $\mathtt{sk}_i$.

$$\mathtt{skP}_i(\mathtt{bitInd}_k) \vee \ldots \vee \mathtt{skP}_i(\mathtt{bitInd}_m). \tag{12}$$

Informally if $\mathtt{skP}_i(\mathtt{bitInd}_j)$ is true then we can assign $\mathtt{sk}_i$ to be equal $\mathtt{bitInd}_j$. Further, wherever $\mathtt{sk}_i$ occurs in a clause $C(\mathtt{sk}_i, X_1, \ldots X_n)$ then we replace this clause with $\neg\mathtt{skP}_i(Y) \vee C(Y, X_1, \ldots X_n)$. After this transformation, the first four formulas in (11) will have the following form:

$$\begin{array}{l}\mathtt{readeq} \vee \neg\mathtt{iread}(B) \vee \neg\mathtt{sread}(B) \vee \neg\mathtt{skP}_0(B);\\ \mathtt{readeq} \vee \mathtt{iread}(B) \vee \mathtt{sread}(B) \vee \neg\mathtt{skP}_0(B);\\ \mathtt{readeq} \vee \neg\mathtt{less}_0(B) \vee \neg\mathtt{skP}_0(B);\\ \mathtt{readeq} \vee \mathtt{less}_{71}(B) \vee \neg\mathtt{skP}_0(B).\end{array}$$

Then we can define predicates $\mathtt{less}_i$ for i as follows: e.g. for $\mathtt{less}_3$ we have unit clauses:

$$\begin{array}{l}\mathtt{less}_3(\mathtt{bitInd}_0), \mathtt{less}_3(\mathtt{bitInd}_1), \mathtt{less}_3(\mathtt{bitInd}_2),\\ \neg\mathtt{less}_3(\mathtt{bitInd}_3), \ldots, \neg\mathtt{less}_3(\mathtt{bitInd}_n).\end{array} \tag{13}$$

Note now we do not need axioms like (10) (introducing equalities between bit-index terms) any more.

Theorem 2: The Skolem predicates encoding is sound and complete: a formula obtained by the precise ranges encoding is satisfiable if and only if the corresponding formula obtained by the Skolem predicates encoding is satisfiable.

Proof: An adaptation of results from [1], [8]. ■

IV. Analysis of proof calculi for EPR

We compare general purpose first-order reasoners with dedicated SMT solvers on the benchmarks generated from industrial memory designs. Since our encodings are falling into the EPR fragment we focus on instantiation-based first-order reasoners which are especially efficient in this fragment, as witnessed by recent CASC competitions [4]. Instantiation-based methods are general purpose reasoning methods for first-order logic which are based on combining efficient propositional, or more generally ground, reasoning techniques with instantiation of first-order formulas. Instantiation-based methods are therefore well-suited for reasoning with fragments closely related to propositional logic such as the EPR fragment and in particular decide the EPR fragment. We consider two sate-of-the-art instantiation-based reasoners: the Darwin system [3], based on the *Model Evolution calculus* [2] and the iProver system [15], based on the *Inst-Gen calculus* [10].

The Model Evolution calculus can be seen as a lifting of efficient propositional DPLL calculus into first-order logic together with a number of DPLL-style techniques such as (dynamic) backtracking and lemma learning. The Model Evolution calculus is space efficient since only the candidate model is growing during the proof search (and optionally, the set of lemmas if lemma learning is applied). On the other hand, such lifting to the first-order logic requires to compute expensive context unifiers and considerably complicates dynamic backtracking and lemma learning, generally rendering them not as effective as in the propositional case.

The Inst-Gen calculus is based on a modular combination of propositional reasoning with refined instantiations of first-order formulas. One of the distinctive features of the Inst-Gen approach is that it allows one to employ off-the-shelf efficient propositional solvers (currently iProver integrates MiniSAT [9]) for reasoning with propositional abstractions of first-order clauses, guiding the instantiation inferences and simplification of clauses. We believe that such a modular integration of industrial-strength propositional solvers gives a considerable advantage when solving large real-life problems. Another important requirement from a solver used in a verification environment is to produce models for satisfiable problems. Such models correspond to bugs in the design and it is crucial to have a model representation amendable to efficient analysis. As a byproduct of this work, iProver has been extended with a representation of models such that the value of each bit in a bit-vector can be retrieved efficiently; this considerably simplified model analysis.

Our experimental results show that already non-tuned general purpose instantiation-based systems are close in performance and in some examples outperform highly optimized dedicated SMT solvers. These initial results are very encouraging and we believe that instantiation-based reasoners can be tuned further by exploiting the problem structure and by optimizing inference selection.

Let us now discuss different effects of our encodings on the EPR reasoners. First we note that the size of bit-vectors is directly related to the size of the search space. Therefore reducing the size of bit-vectors in the encodings is a promising research direction. Moreover, large ranges of bit-indexes produce clauses with large numbers of equational literals like (10). In general, instantiation-based methods are more tolerant to clauses with many literals than resolution-based methods since the number of literals in clauses does not increase during the instantiation process. Nevertheless equational axioms as (10) can produce numerous redundant inferences by substituting the variable B with different indexes during equational reasoning. All this instantiations are redundant and can be avoided as shown in Section III-C.

Let us compare our approach of encoding bit-vector and array reasoning into the EPR fragment with approaches used in SMT solvers. Reasoning in SMT solvers is done at the ground level and frequently results in full bit-blasting. Using first-order logic we can use higher levels of abstraction which can result in memory/bit-vector size independent reasoning. We believe this can lead to better scalability of our approach to large memories and bit-vectors. On the other hand, SMT solvers have advanced built-in bit-vector functions which are needed in many memory designs. Although it is possible to bit-blast such functions in our approach, a better approach

[4]http://www.cs.miami.edu/~tptp/CASC/

would be to devise encodings of these functions into the EPR fragment.

A further research direction is to strengthen our encodings by introducing higher level abstractions and by more sophisticated encoding of bit-vector reasoning. Such an encoding can also pave the way for using powerful resolution-based first-order reasoners such as Vampire [23], as the current encoding tends to produce very long clauses which are known to be hard for resolution-based reasoners.

V. Experimental evaluation

In this section, we evaluate the three above-discussed sound and complete encodings of problems with bit-vectors and arrays into EPR on two fastest EPR solvers, iProver [15] and Darwin [3]. We further compare their performance to that of the fastest SMT solvers for the theory QF_AUFBV – Boolector [5] and MathSAT [6]. We used a standard, and straightforward, encoding of RTL descriptions of hardware to the theory QF_AUFBV (for example, we haven't used the abstraction technique in [11] to reduce the number of involved bits during the encoding). With the incomplete encoding of [14], iProver returned spurious models on all problems which are UNSAT with the complete encodings; therefore we do not report here the EPR solver results with this encoding.

A. Description of benchmarks

In our experiments, we use five equivalence checking problem instances originating from a recent micro-processor design at Intel. Each problem instance corresponds to an equivalence checking problem between an RTL functional block (FUB) and the corresponding FUB in the schematic model.

The first group of experiments reported in Table 2 correspond to the original RTL and schematic FUBs. The schematic model contains lots of bit-level reasoning, and as a result the resulting EPR instances contain lots of bit-level equations (using the bit-indexes). On such instances, the abstraction techniques are less efficient, and the solvers that do not really employ the bit-vector level reasoning can perform almost as efficiently as on the problems with lots of reasoning at higher, bit-vector level reasoning.

Recall that in EPR encodings often there is a need to write axioms at bit level, say in equations like (4). As explained above, one expects that existence of a large amount of such index constants will negatively affect the performance of EPR solvers. To evaluate this point experimentally, for each equivalence checking benchmark we tried to produce an equivalent instance involving significantly fewer bit-indexes. This transformation was performed by manually editing the RTL and SCH descriptions and changing compilation switches when generating the model-checking instances (e.g., the next-state functions) from hardware descriptions. In brief, because of the way how the compiler works, linearization (or flattening) of (nested) structures or modules, causes creation of long bit-vectors containing the original bit-vector fields of structures and bit-vectors of modules as sub-vectors. This phenomenon is an artifact of compilation and does not change the meaning of the design. Undoing or preventing this linearization (even if it was not done as aggressively as possible), allowed us to significantly reduce the amount of long bit-vectors and the amount of bit-indexes involved in the generated EPR instances for FUBs 1 and 2. For the other FUBs the maximal width of bit-vectors remained unchanged. Benchmark results on these modified FUBs are reported in Table 3.

It is well understood that solvers might perform particularly well or badly on SAT vs UNSAT problems, and we aim to evaluate the selected solvers and encoding methods from this angle as well. To generate SAT instances, we manually introduced several common types of bugs into the designs or verification instances (such bugs include mismatches between the corresponding read or write enables, mixture in the order of data bits, incorrect or missing constraints (3) connecting the corresponding write data of the compared slices of specification and implementation designs, etc.). Tables 4 and 5 report runtime results on 5 FUBs obtained by these manipulations from the equivalence checking problems evaluated in Tables 2 and 3, respectively.

The formulas checked for SAT/UNSAT correspond to the induction step formulas [24] at depths smaller or equal to 3 – the depths needed to prove the induction invariant stating the equality of memories (1) and the read data (2). For the sake of performance efficiency, checking these formulas there split into two independent runs of the solvers; in one run, the initial value of the main clock was set to true, while in the second check it was set to false.

B. Performance results

One of the most important observations based on our experimental results is that already at this initial stage, non-tuned general purpose instantiation-based methods can solve industrial-size hardware verification problems within a reasonable time limit. Moreover, there are a number of problems where instantiation-based solvers outperform highly optimised SMT solvers, see Tables 2–5. In particular, instantiation-based methods perform well on the problems with long bit-vectors such as problems FUB 4 and FUB 5 (Tables 2–3), with maximal bit-vector sizes 994 and 1047 respectively. We believe this is one of the promising aspects of the instantiation-based approach which is achieved due to a higher level reasoning.

Let us note that SMT solvers and an instantiation-based solver iProver are all using SAT solvers as the back-end. In the case of Boolector it is PrecoSAT and in the case of MathSAT and iProver it is MiniSAT. Recently developed PrecoSAT is a highly optimized propositional solver which won the latest SAT competition. Thus, comparing MathSAT and iProver better highlights the differences between SMT and instantiation approaches since the same SAT solver is employed. We can see that iProver outperforms MathSAT on many problems both in SAT and UNSAT categories.

These experimental results indicate that instantiation-based methods and SMT technology complement each other and both are useful alternatives for industrial-size hardware verification. There are still a number of problems were SMT solvers perform better than instantiation-based methods, especially on satisfiable problems. Therefore we are planning to explore applicability of recent advances in bit-vector reasoning developed in the SMT framework [5] into instantiation-based reasoning.

Let us compare our different encodings. Tables 2–5 indicate that there is no clear winner among our encodings. There is a trade-off between concise, higher-level encodings such as precise ranges and Skolem predicates encodings; and more explicit bit-index pre-instantiation encoding. These tables show that explicit encodings are better for unsatisfiable problems whereas concise encodings are better for satisfiable problems. The reason for this can be that in many cases low level reasoning is unavoidable for unsatisfiable problems whereas for satisfiable problems it is sufficient to consider concise representations.

Tables 2–5 show experiments with longer/shorter bit-vector encodings. The reduction of bit-vector sizes was not always successful, only in two first FUBs there was a noticeable reduction in the maximal bit-vector size: in FUB 1 from 286 to 185 and in FUB 2 from 640 to 203, in other three cases the instances have changed but the max bit-vector size was unchanged. We can see that in some cases shorter bit-vectors lead to performance improvement and therefore in our future work we will study how to reduce bit-vector sizes in our encodings.

Finally, we run Darwin on several groups of benchmarks (including the simplest FUBs 1-3) with time limit of 500 seconds, but unfortunately it could not solve any single problem. The reason can be the large number of clauses in the resulting problems which ranges from 30 thousands to over 100 thousands of clauses. We believe that a modular integration of propositional reasoning as it is done in iProver is advantageous on such problems. The problem of reducing the size of the encodings is also needed to be addressed.

VI. Conclusions and future work

The aim of this work was to explore the scalability and potential of several approaches to first-order logic theorem proving in solving industrial-sized verification problems involving reasoning with bit-vectors and arrays, and to compare them with SMT-based techniques. Taking into account that the EPR solvers are currently less optimized on industrial sized problems compared to more mature SMT solvers, the reported experiential results and theoretical analysis indicate that several first-order proof calculi do have a great potential in this domain. Furthermore, we believe that smarter encodings into EPR of the problems with bit-vectors and arrays can be developed by exploring abstraction and refinement techniques similar to those proposed for accelerating SMT solving and this can make the EPR-based approaches even more efficient.

Another big promise of using EPR solvers in model checking is that bounded model checking problems have a succinct encoding into EPR, such that the size of the BMC formulas is not affected by the unrolling bound [17]: unlike the SAT-based BMC [4], it is not needed to replicate copies of the temporal assertion and the transition relation for every unrolling depth. One of our major next goals in the EPR related model-checking research is to combine the ability of solving bit-vector and array reasoning instances in EPR at the word level with the EPR-based BMC proposed in [17] for bit-blasted model-checking instances. Furthermore, we believe that EPR solvers can be optimized on model checking instances resulted from this combined encodings.

This reported and future work is part of an ongoing research collaboration between Intel's formal technology group developing efficient model-checking and equivalence checking solutions for Intel's chip design project and between the University of Manchester. The developed word-level equivalence checking method will replace the more traditional sequential equivalence checking solution implemented in Intel's sequential equivalence checking tool, Seqver [12], [13], [14].

Acknowledgements. We would like to thank the developers of Boolector and MathSAT for their collaboration on this work.

References

[1] Baumgartner, P., A. Fuchs, H. de Nivelle, C. Tinelli. Computing finite models by reduction to function-free clause logic. J. of Applied Logic, 2007.

[2] Baumgartner P., C. Tinelli. The model evolution calculus as a first-order DPLL method. Artif. Intell. 172(4-5): 591-632, 2008.

[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] Brummayer R., A. Biere. Boolector: An efficient SMT solver for bit-vectors and arrays, TACAS 2009.

[6] Bruttomesso R., A. Cimatti, A. Franzén, A. Griggio, R. Sebastiani. The MathSAT 4 SMT solver, CAV 2008.

[7] Bradley, A.R., Manna Z., Sipma H.B. What's decidable about arrays? VMCAI 2006.

[8] Claessen K., N. Sörensson. New techniques that improve MACE-style model finding. Workshop on Model Computation (MODEL), 2003.

[9] Eén N., N. Sörensson. An extensible SAT-solver. SAT 2003: 502-518.

[10] Ganzinger H., Korovin K. New directions in instantiation-based theorem proving, LICS 2003.

[11] Johannsen P. Reducing bitvector satisfiability problems to scale down design sizes for RTL property checking, HLDVT 2001.

[12] Kaiss, D., S. Goldenberg, Z. Hanna, Z. Khasidashvili. Seqver: a sequential equivalence verifier for hardware designs, ICCD 2006.

[13] Khasidashvili, Z., D. Kaiss, D. Bustan. A compositional theory for post-reboot observational equivalence checking of hardware, FMCAD 2009.

[14] Khasidashvili Z., Kinanah M., Voronkov A. Verifying equivalence of memories using a first order logic theorem prover. FMCAD 2009.

[15] Korovin, K. iProver–an instantiation-based theorem prover for first-order logic (system description), IJCAR 2008.

[16] Kroening D., Strichman O. *Decision Procedures*, Springer, 2008.

[17] Navarro Pérez J. A., A. Voronkov. Encodings of Bounded LTL Model Checking in Effectively Propositional Logic. CADE 2007.

[18] Navarro-Pérez J.A., A. Voronkov. Proof systems for effectively propositional logic, IJCAR 2008.

[19] McCarthy, J., J. Painter. Correctness of a compiler for arithmetic expressions. Symposium in Applied Mathematics, Vol. 19, Mathematical Aspects of Computer Science, American Mathematical Society, 1967.

[20] de Moura, L.M., N. Bjørner. Deciding effectively propositional logic using DPLL and substitution sets, IJCAR 2008.

[21] de Moura, L.M., N. Bjørner. Z3: An efficient SMT solver, TACAS 2008.

[22] Ge Y., de Moura L.M. Complete instantiation for quantified SMT formulas, CAV 2009.

[23] Riazanov, A., A. Voronkov. The design and implementation of Vampire, AI Communications, 15(2-3):91–110, 2002.

[24] Sheeran, M., S. Singh, G. Stalmarck. Checking safety properties using induction and a SAT-solver, FMCAD 2000.

Solver **Test**	**Boolector** **clock=0/1**	**MathSAT** **clock=0/1**	**iProver pre-inst** **clock=0/1**	**iProver Skolemize** **clock=0/1**	**iProver precise** **clock=0/1**
FUB 1	0.39 / 0.18	13.0 / 3.5	1.6 / 11.6	1.3 / 1.2	9.1 / 12.8
FUB 2	0.36 / 0.3	7.7 / 4.7	1.9 / 7.6	6.7 / 11.6	42 / 78.3
FUB 3	0.05 / 0.04	0.8 / 0.9	1.4 / 0.4	3.6 / 1.8	4.1 / 4.7
FUB 4	0.13 / 138.5	26 / t-o	4.8 / 51.1	44 / t-o	42 / t-o
FUB 5	5861.26 / 3.2	160.15 / 31.75	179.24 / 13.3	t-o / 680.94	1132.36 / 329.1
TOTAL	5862.19 / 142.04	207.65 / t-o	**188.94** / **84**	t-o / t-o	1229.56 / t-o

Fig. 2. Equivalence checking UNSAT problem instances with long bit-vectors.

Solver **Test**	**Boolector** **clock=0/1**	**MathSAT** **clock=0/1**	**iProver pre-inst** **clock=0/1**	**iProver Skolemize** **clock=0/1**	**iProver precise** **clock=0/1**
FUB 1	0.28 / 0.18	12.0 / 3.8	1.6 / 6.4	6.6 / 40.3	8.6 / 11
FUB 2	0.32 / 0.34	14.7 / 11.5	1.8 / 19.0	9.0 / 43.1	19.6 / 31.3
FUB 3	0.04 / 0.04	0.8 / 0.9	1.2 / 0.4	3.6 / 1.9	4.1 / 4.7
FUB 4	0.13 / 138.8	t-o / t-o	t-o / t-o	43.7 / t-o	42.2 / t-o
FUB 5	t-o / 2.98	158.94 / 31.71	149.88 / 11.08	t-o / 592.3	1084.7 / 320.33
TOTAL	t-o / **142.34**	t-o / t-o	t-o / t-o	t-o / t-o	**1159.2** / t-o

Fig. 3. Equivalence checking UNSAT problem instances with shorter bit-vectors.

Solver **Test**	**Boolector** **clock=0/1**	**MathSAT** **clock=0/1**	**iProver pre-inst** **clock=0/1**	**iProver Skolemize** **clock=0/1**	**iProver precise** **clock=0/1**
FUB 1	0.14 / 0.14	36.3 / 34.9	5.1 / 11.2	1.4 / 1.2	9.0 / 29.3
FUB 2	0.22 / 0.24	50 / 38.9	5.3 / 15.3	6.3 / 11.5	24.4 / 57.5
FUB 3	0.04 / 0.04	3.2 / 3.3	15.1 / 0.9	26.4 / 1.7	6.2 / 2.7
FUB 4	0.14 / 0.42	t-o / t-o	147.5 / t-o	39.7 / t-o	46.7 / 46.4
FUB 5	1.66 / 1.56	t-o / t-o	63.65 / 62.57	46.58 / 48.51	379.68 / 439.95
TOTAL	**2.2** / **2.4**	t-o / t-o	236.65 / t-o	120.38 / t-o	465.98 / 575.85

Fig. 4. Equivalence checking SAT problem instances with long bit-vectors.

Solver **Test**	**Boolector** **clock=0/1**	**MathSAT** **clock=0/1**	**iProver pre-inst** **clock=0/1**	**iProver Skolemize** **clock=0/1**	**iProver precise** **clock=0/1**
FUB 1	0.14 / 0.16	42.8 / 36.9	5.0 / 10.4	5.7 / 71.7	8.3 / 11.9
FUB 2	0.21 / 0.26	92.2 / 48.4	6.1 / 11.4	10.3 / 47.9	10.5 / 32.3
FUB 3	0.04 / 0.04	3.1 / 3.2	15.3 / 1.0	26.6 / 1.6	6.0 / 2.7
FUB 4	0.14 / 0.38	t-o / t-o	129.5 / t-o	44.0 / t-o	44.1 / 47.4
FUB 5	1.66 / 1.54	t-o / t-o	291.48 / 92.93	43.61 / 42.89	424.71 / 511.34
TOTAL	**2.19** / **2.38**	t-o / t-o	447.38 / t-o	130.21 / t-o	493.61 / 605.64

Fig. 5. Equivalence checking SAT problem instances with shorter bit-vectors.