DIFTS

FIRST INTERNATIONAL WORKSHOP on
DESIGN and IMPLEMENTATION of FORMAL TOOLS and SYSTEMS
Austin, TX Nov 3rd, 2011
Co-located with FMCAD 2011

Program Proceedings

Program Chairs

Malay K. Ganai	NEC Labs America USA
Armin Biere	Johannes Kelpler University Austria

Technical Program Committee

Clark W. Barrett	New York University USA
Armin Biere	Johannes Kelpler University Austria
Alessandro Cimatti	Fondazione Bruno Kessler Italy
Cindy Eisner	IBM Haifa Research Lab Israel
Malay K. Ganai	NEC Labs America USA
Ganesh Gopalakrishnan	University of Utah USA
Daniel Kroening	Oxford University UK
Robert P. Kurshan	Cadence Design Systems USA
Ken Mcmillan	Microsoft Research USA
Chao Wang	Virginia Tech, USA

Local Arrangement Chairs

Sandip Ray	UT Austin, USA
Nadia Papakonstantinou	NEC Labs America

Preface

The first DIFTS (Design and Implementation of Formal Tools and Systems) workshop was held at Austin, Texas on Nov 3rd, 2011, co-located with Formal Methods in Computer-Aided Design (FMCAD) Conference. The workshop emphasized the insightful experiences in tool and system design. It provided a forum for sharing challenges and solutions that are highly original with ground breaking results. It took a broad view of the formal tools area, and solicited contributions from various domains including decision procedures, verification, testing, validation, diagnosis, debugging, and synthesis. It provided a forum for discussing the engineering aspect of the tools, and various design decisions required to put them in practical use. The workshop provided a discussion forum for a pragmatic view of practicing formal methods.

The workshop received 9 original submissions, out of which 4 were chosen under tool category and 2 were chosen under system category. There were also two invited talks: one given by Andreas Kuehlmann, Sr. VP of R&D at Coverity on "The pain of making research tools into software products", and another by Chris Morison, Chief Architect, Real Intent Inc. on "From putty to product: what it takes to bring a verification tool to market".

First of all we thank FMCAD's steering committee for their approval of the workshop. We also thank Anna Slobodova, David Rager, and Sandip Ray for their help in local arrangements and Nadia Papakonstantinou for her help in web updates. We sincerely thank the program committee members and sub reviewers for selecting the papers and providing candid review feedbacks to the authors. Last but not least, we thank all the authors for contributing to the workshop and to all the participants of the workshop.

Malay K. Ganai and Armin Biere

Program Chairs

Table of Contents

The Pain of Making Research Tools into Software Products

Andreas Kuehlmann
Sr. VP of R&D at Coverity
President of IEEE Council on EDA (CEDA), and an IEEE fellow

Abstract: Building a research implementation of a new algorithm and validating it for a small set of benchmarks is a key component of pushing the state of the art of design and verification tools. Moreover, if the algorithm works, it is often also the most exciting phase of the long journey to have a new idea used by many people. The less exciting and often more painful part includes the productization of the research into a form that can be applied by hundreds or thousands of users on a daily basis. In this talk we discuss some of the challenges that developers encounter in a product development environment from prototyping to code maintenance. The presentation will cover practical aspects of developing software in larger teams from product planning to deployment and support. We will also elaborate on some of the opportunities for new research to help the practical deployment and maintenance of software products.

From Putty to Product: What it takes to bring a Verification Tool to Market

Dr. Chris Morrison
Chief Architect, Real Intent Inc.

Abstract: For a new company interested in developing a formal verification product, the temptation would be to believe that the majority of the time-to-market would be spent on developing the core formal technology. As a corollary, if that core technology could be bought off-the-shelf, the time-to-market would be greatly reduced. While that belief may have been true for the first generation of formal verification products, it is, unfortunately, not credible today. There are many "significant items" in state-of-the-art products that now consume considerable resources. The input language has changed from BLIF to SystemVerilog '09 and VHDL '08. Designs that the formal tools had to consider used to be single-clock and simple reset designs. Now they have 10+ clocks, derived and gated clocks, complex resets and multiple functional modes. The output is no longer a simple text file, but a hyperlinked report tying together VCD waveforms, source RTL, and state machine and schematic viewers. Manual control with many switches is no longer acceptable. A nearly automatic flow with very few switches is demanded. The tool must manage to deliver very high throughput for thousands of checks on multimillion-gate designs as well as acceptable user experience on high-latency checks. User productivity is the combination of tool run time, piloting convenience as well as the debug experience. The infrastructure needed to achieve these goals (which was originally simple and relatively quick to implement) is now very complex involving an intricate flow through a variety of substantial software packages and multiple databases. With the goal of reaching a broad class of customers, current and future verification products must be automated, easy to use and be able to reliably produce actionable results. Delivering on that requires attention to every part of the product rather than just the formal verification technology.

An Application of Formal Methods to Cognitive Radios*

Konstantine Arkoudas and Ritu Chadha and Jason Chiang
Telcordia Research
One Telcordia Drive
Piscataway, NJ 08854
{konstantine, chadha, chiang}@research.telcordia.com

ABSTRACT

We discuss the design and implementation of a formal policy system regulating dynamic spectrum access (DSA) for cognitive radios. DSA policies are represented and manipulated in a proof framework based on first-order logic with arithmetic and algebraic data types. Various algebraic operations combining such policies can be easily implemented in such a framework. Reasoning about transmission requests is formulated as a satisfiability-modulo-theories (SMT) problem. Efficient SMT solvers are used to answer the corresponding queries and also to analyze the policies themselves. Further, we show how to reason about transmission requests in an optimal way by modeling the problem as an SMT instance of weighted Max-SAT, and we demonstrate that this problem too can be efficiently solved by cutting-edge SMT solvers. We also show that additional optimal operations on transmission requests can be cast as classic optimization problems, and to that end we give an algorithm for minimizing integer-valued objective functions with a small number of calls to an oracle SMT solver. We present experimental results on the performance of our system, and compare it to previous work in the field.

1. INTRODUCTION

One of the world's most prized physical resources is the electromagnetic spectrum, and particularly its radio frequency (RF) portion, stretching roughly from 10 KHz to 300 GHz. The RF spectrum is so valuable that its allocation is strictly regulated by world governments, and these days even small parts of it can be sold for billions of dollars in spectrum auctions.

*The research reported in this paper was performed in connection with contract number W15P7T-08-C-P213 with the U. S. Army Communications Electronics Research and Development Engineering Center (CERDEC). The views and conclusions contained in this document are those of the authors and should not be interpreted as presenting the official policies or position, either expressed or implied, of the U. S. Army CERDEC, or the U. S. Government unless so designated by other authorized documents. Citation of manufacturers or trade names does not constitute an official endorsement or approval of the use thereof. The U. S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.

As wireless devices continue to proliferate, demand for access to RF spectrum is becoming increasingly pressing, and increasingly difficult to achieve. After all, the (usable) spectrum is finite, while the demand for it continues to increase without bounds. However, it has been recognized [13, 14] that problems of spectrum scarcity and overuse arise not so much from physical shortage of frequencies, but mostly as a result of the centralized and rigid way in which spectrum allocation has been managed. Spectrum has been allocated in a *static* way: bands of the RF range are statically assigned to licenced users on a long-term basis, who then have exclusive rights to the corresponding region. Examples here include the 824–849 MHz, 1.85–1.91 GHz, and 1.930–1.99 GHz frequency bands, which are reserved for FCC-licenced cellular and personal communications services (PCS). Other parts of the RF spectrum, by contrast, are unlicensed, and anyone can transmit in those frequencies as long as their power does not exceed the regulatory maximum. Consequently, large portions of the RF range remain underutilized for long periods of time, while other parts—such as the ISM bands—are overutilized.

Observations like these provided the impetus behind the NeXt Generation (XG) Communications program, a technology development program sponsored by DARPA [9], which proposed opportunistic spectrum access as a means of coping with spectrum scarcity. The underlying approach has come to be known as dynamic spectrum access (DSA), and has evolved into a general methodology for dealing with spectrum scarcity [17]. The main idea is that a DSA network has two classes of users: primary and secondary. Primary users are licenced to use a particular spectrum band and always have full access to that band whenever they need it. Secondary users are allowed to use such spectrum portions but only as long as their use does not interfere with the operating needs of the primary users. The secondary users are typically cognitive radios [4] that can dynamically adjust their operating characteristics (such as waveform, power, etc.). Secondary users *sense* the spectrum for available transmission opportunities, determine the presence of primary users, and attempt to use the spectrum in a way that interferes as little as possible with the activities of the primary users.

Policies are used for specifying how secondary radio nodes

are allowed to behave. Policies consist of declarative statements that dictate what secondary radios may or may not do, without prescribing how they might do it [18]. Different policies may be applicable in different regions of the world. Even in one and the same region there may be multiple policies in place, reflecting different constraints imposed by different regulatory bodies. Policy systems for cognitive radios should be able to load new policies on the fly.

There are two key questions in designing such a policy system, one representational and the other computational: How should we represent policies, and how can we make efficient use of them? Policies regulating DSA must be able to express rules such as the following: "Allow a transmission if its frequency is in a certain range and the power does not exceed a certain limit, *or* if the transmission is an emergency." Thus, for representation purposes, we need at least the full power of propositional logic, as well as numbers (both integers and reals), and numeric comparison relations. It would also be helpful to have algebraic datatypes at our disposal, so that we could, for example, represent the *mode* of a transmission as one of a finite number of distinct symbolic values, such as *everyDay* or *emergency*. A certain amount of quantification would also be convenient, so that we could, e.g., quantify over all possible transmissions. We must also be able to introduce arbitrary transmission parameters, so a strong type system, preferrably with subtyping, should be available. Hence, first-order logic with arithmetic, and perhaps additional features such as algebraic datatypes and subsorting, would appear to be a natural representation choice, provided that we can make efficient use of it, a point that we will address shortly.

So much for representation. What are the main computational problems that a policy system must solve? The main problem is this: determine whether an arbitrary transmission request should be allowed or denied, given the current set of policies. In fact, if this were all there was to using policies for spectrum access, things would be relatively simple. But we usually want the policy engine to do more. For example, when a transmission request is not permissible, a simple "denied" answer is not very useful. The policy system should also tell the requesting radio why the request was denied, and more importantly, what it would take for its request to be allowed. As a simple example, the system might tell the radio: "Your request cannot be allowed in its present form, but it will be allowed if you only change your transmission power to such-and-such level." Further, in such cases the policy system will usually have many choices as to which parts of the transmission request to modify in order to make it permissible. We then want it to make an optimal choice, i.e., a choice that satisfies the original request to the greatest possible extent.

In this paper we present an implementation of a policy system in Athena, an interactive proof system for polymorphic multi-sorted first-order logic with equality, algebraic data types, subsorting, and a higher-order functional programming language [3]. Athena's rich logic, sort system, and programming language significantly facilitated the representation and manipulation of policies. Another innovation of our paper is the use of SMT solvers [15] for reasoning with and about policies. The problem of determining whether a transmission request is permissible by a given policy can be couched as a satisfiability problem, namely, the problem of determining whether the request is consistent with the policy. So SAT solving would seem to be a natural fit for this domain. However, policy rules in this domain also make heavy use of equality and numeric relations, not to mention symbolic values such as transmission modes. Thus, viewed as an abstract syntax tree, the body of a policy rule has an arbitrarily complicated boolean structure internally, with relational atoms at the leaves, whose semantics come from standard theories such as those of arithmetic, equality, and so on. That is what makes SMT solving an ideal paradigm for this problem. We will show that the problem of making optimal adjustments to transmission parameters can also be formulated quite naturally in this setting as a weighted Max-SAT problem, many instances of which can be efficiently solved despite the problem's high theoretical complexity. Indeed, competitive SMT solvers such as CVC3 [7] and Yices [6] are highly optimized programs that can readily decide extremely large formulas. As a quick comparison, another recently built policy system for spectrum access, Bresap, which uses BDDs as its underlying technology, cannot handle policies with more than 20 atomic expressions [1, p. 83]. By contrast, our system can easily handle policies with many *thousands* of atomic expressions.

The remainder of this paper is structured as follows. The next section describes the representation of spectrum-access policies in Athena.[1] Section 3 discusses the integration of SMT solvers with Athena, and shows how to reduce the problem of reasoning with spectrum-access policies to SMT solving. In section 4 we model the problem of making optimal changes to transmission requests as a Max-SAT problem, and we describe how our system models and solves such problems. We also present an optimizing algorithm that uses a version of binary search in order to minimize integer-valued objective functions with very few calls to an oracle SMT solver (at most $\log(n)$ such calls, where n is the maximum value that can be attained by the objective function). This algorithm can be used to further optimize transmissions, e.g., by minimizing the distance between the values desired by the cognitive radio and the values allowed by the policy. Section 5 presents results evaluating the performance of our policy engine. Finally, section 6 discusses related work and concludes.

2. POLICY REPRESENTATION

[1]While some Athena code is presented, knowledge of the language is not necessary. Any reader familiar with formal methods and functional programming languages should be able to follow the code.

The set of radio transmissions is treated as an abstract data type with fields such as frequency, power, mode, etc. The abstract domain of transmissions is introduced in Athena as follows:

```
domain Transmission
```

A transmission field (or "parameter") can then be introduced as a function from transmissions to values of some appropriate sort. For illustration purposes, we demonstrate the declaration of the following parameters: frequency, power, mode, and time. Additional parameters can be introduced as needed.

```
declare frequency: [Transmission] -> Int
declare power: [Transmission] -> Real
declare mode: [Transmission] -> Mode
declare hours, minutes: [Transmission] -> Int
```

Here `Int` and `Real` are built-in Athena sorts representing the sets of integers and real numbers, respectively, with `Int` a subsort of `Real`. The sort `Mode` is a finite algebraic datatype:

```
datatype Mode := emergency | everyDay | ...
```

There are two distinguished unary predicates on the set of all possible transmissions: `allow` and `deny`. These predicates are used by policies to specify which transmissions are considered permissible and which are not. More concretely, a policy is represented as a pair of sentences: an *allow condition* and a *deny condition*. An allow condition (or AC for short) is of the form: [2]

```
(forall ?x:Transmission.  allow ?x <==> ···)   (1)
```

while a deny condition (DC) is of the form:

```
(forall ?x:Transmission.  deny ?x <==> ···)   (2)
```

Either condition may be absent, i.e., a policy may specify only an allow condition but no deny condition, or only a deny condition but no allow condition. An absent condition is represented by the unit value, `()`. The constructor for policies is therefore a procedure that takes two conditions and simply puts them together in a two-element list:

```
define make-policy :=
  lambda (AC DC) [AC DC]
```

There are two destructor or selector procedures, one for retrieving each component of the policy:

```
define [get-AC get-DC] := [first second]
```

An important operation is the *application* of a given policy condition `C` of the form (1) or (2) to a term `t` of sort `Transmission`. Let `body` be whatever sentence would normally be where the ellipses now appear in (1) (or in (2)). Then the result of the said application is the sentence obtained from `body` by replacing every free occurrence of the quantified variable `?x:Transmission` by the term `t`. In Athena code, this procedure is defined as follows:

```
define apply :=
  lambda (C t)
    match C {
      (forall x ((_ x) <==> body)) =>
        (replace-var x t body) }
```

The built-in ternary Athena procedure `replace-var` is such that `(replace-var x t p)` produces the result obtained from the sentence `p` by replacing every free occurrence of variable `x` by the term `p`. As a concrete example, here is a sample AC:

```
define AC1 :=
 (forall ?x .
   allow ?x <==>
    frequency ?x in [5000 5700] &
    power ?x < 30.0 &
    mode ?x = everyDay & hours ?x <= 11)
```

It says that a transmission is (provisionally) permissible iff its frequency is in the 5000 . . . 5700 MHz range; its power is less than 30.0 dBm; its mode is `everyDay`; and the transmission's time is no later than 11 in the morning. Here `in` is a binary procedure defined as follows:

```
define in :=
  lambda (x range)
    match range {
      [a b] => (a <= x & x <= b)}
```

(Note that variables do not need to be explicitly annotated with their sorts. Athena has polymorphic sort inference, so a Hindley-Milner algorithm will recover the most general possible sorts of all variable occurrences.) We can now construct a policy with the above AC and with no DC as follows:

```
define policy-1 := (make-policy AC1 ())
```

Our second policy example has both an AC and a DC:

```
define policy-2 :=
  (make-policy
     (forall ?x .
       allow ?x <==> mode ?x = emergency)
     (forall ?x .
        deny ?x  <==> power ?x > 25.0))
```

A key operation on policies is that of *merging*. New policies may be downloaded into the policy database at any given time. A newly acquired policy must be integrated with the existing policy that is in place in order to produce a single merged policy with a new AC and a new DC. Merging is therefore a binary operation. The current definition of merging two policies p_1 and p_2 is simple: it is disjunction on both the AC and DC. Specifically, the AC of the merged policy allows a transmission t iff p_1 allows t *or* p_2 allows it; and it denies t iff p_1 denies it *or* p_2 denies it. Merging is

[2] A more accurate term instead of `allow` would be "allow provisionally," because, as we will see, for a transmission request to be granted, the allow condition must hold *and* the deny condition must not hold. With this caveat in mind, we will continue to use the term `allow` in the interest of simplicity.

implemented as an Athena procedure. Other alternative definitions of merging are possible, and these could be straightforwardly implemented in the same way. As an example, here is the result of merging `policy-1` and `policy-2`, as these were defined above:

```
define merged-policy :=
  (merge-policies policy-1 policy-2)

>(get-AC merged-policy)

Sentence:
  (forall ?v4:Transmission
   (iff (allow ?v4)
        (or (and (and (<= 5000
                          (frequency ?v4))
                      (<= (frequency ?v4)
                          5700))
                 (and (< (power ?v4)
                         30.0)
                      (and (= (mode ?v4)
                              everyDay)
                           (<= (hours ?v4)
                               11))))
            (= (mode ?v4)
               emergency)))))

>(get-DC merged-policy)

Sentence: (forall ?v5:Transmission
            (iff (deny ?v5)
                 (> (power ?v5)
                    25.0)))
```

(Note that while Athena accepts input in either infix or prefix form, sentences are output in prefix for indentation purposes.)

We define a *transmission request* as a set of constraints over some object of sort `Transmission`, typically just a free variable ranging over `Transmission`. These constraints, which are usually desired values for some—or all—of the transmission parameters, can be expressed simply as a sentence. A sample transmission request:

```
define request-1 := (frequency ?t = 150 &
                     mode ?t = everyDay &
                     hours ?t = 16)
```

This is a conjunction specifying that the `frequency` of the desired transmission (represented by the free variable `?t`) should be 150, its mode should be `everyDay`, and its time should be between 4:00 and 4:59 (inclusive) in the evening. A transmission request may be incomplete, i.e., it may not specify desired values for every transmission parameter. The preceding request was incomplete, since it did not specify values for the `power` and `minutes` parameters.

3. POLICY REASONING

The main task of the policy system is this: determine whether the policies currently in place allow or deny a given transmission request. This is in fact the simplest formulation of the core reasoning problem, in that it only requires a yes/no answer from the policy system. But in general the policy system needs to solve more interesting versions of this problem, namely: If the transmission request is granted, find appropriate values for any missing parameters, in case the request was incomplete; if the request is not granted, find appropriate values for the transmission parameters that *would* render the transmission permissible. Moreover, in the second case, we often want to compute values in a way that is optimal w.r.t. to the given request, e.g., so that the original transmission request is disrupted as little as possible. We discuss such optimality issues in section 4.

The above problems are naturally formulated as satisfiability problems. However, the most appropriate satisfiability framework here is not that of straight propositional logic, but rather that of *satisfiability modulo theories*, or SMT for short [15]. In the SMT paradigm, the solver is given an arbitrary formula of first-order logic with equality and its task is to determine whether or not the formula is satisfiable with respect to certain background theories. Typical background theories of interest are integer and/or real arithmetic (typically linear, but not necessarily), inductive data types (free algebras), the theory of uninterpreted functions, as well as theories of lists, arrays, bit vectors, etc. Hence, most of the function symbols and predicates that appear in the input formula have fixed interpretations, given by these background theories. An SMT solver will not only determine whether a sentence p is satisfiable; if it is, it will also provide appropriate satisfying values for the free variables and/or constants that occur in p.

Athena is integrated with a number of SMT solvers, such as CVC3 and Yices; the one used for this project is Yices [6], mostly because it can solve Max-SAT problems. The main interface is given by a unary Athena procedure `smt-solve`, which takes an arbitrary first-order sentence p and attempts to determine its satisfiability with respect to the appropriate theories. If successful, `smt-solve` will return a list of values for all free variables and/or constants that occur in p. Some examples:

```
datatype Day :=
  Mon | Tue | Wed | Thu | Fri | Sat | Sun

>(smt-solve ?x = 2 & ?y = 3.4 |
            ?x = 5 & ?d = Mon)

List: [(= ?y 3.4) (= ?x 2) (= ?d Mon)]
```

The input to `smt-solve` here was a disjunction of two conjunctions. The free variable `?d` ranges over the datatype `Day`; while `?x` and `?y` range over `Int` and `Real`, respectively. The output is a list of identities providing values for each free variable that render the input satisfiable. Because the argument to `smt-solve` is a sentence, i.e., a native Athena data value, this provides a very flexible and high-bandwidth programmable interface to SMT solvers. This interface proved extremely useful for our system.

The policy reasoning problems described earlier have a natural formulation in SMT. Specifically, consider an arbi-

trary transmission request *tr*, i.e., a first-order sentence with a free variable x ranging over `Transmission` (there may be other free variables as well). To determine whether *tr* is allowed by the current policy: (1) We construct a longer request, *tr'*, which is the conjunction of (a) *tr*; (b) the application of the current policy's AC to x; and (c) the application of the negation of the current policy's DC to x:

```
tr & (apply (get-total-AC) x) &
     ~ (apply (get-total-DC) x)
```

The procedure `get-total-AC` returns the AC of the current ("total") policy; likewise for `get-total-DC`. So *tr'* represents *all* the constraints imposed on the requested transmission, both by the original transmission request, *tr*, and by the policy itself, whose AC must hold for the transmission variable x while its DC must not. (2) We then run `smt-solve` on *tr'*. If the result is a satisfying assignment, then the request is granted, and all we need to do is report values for any missing parameters (parameters that did not figure in the given request). If, by contrast, `smt-solve` determines that *tr'* is unsatisfiable, then *tr* in its given form must be rejected. In that case we make a blank request consisting of the application of the policy's AC to x conjoined with the application of the negation of the policy's DC to x, and run `smt-solve` on it. This blank request therefore imposes no constraints at all on the transmission apart from those imposed by the policy. If the result is a satisfying assignment, we provide it to the user, otherwise we report that the current policy is unsatisfiable.

The Athena code for this algorithm is expressed in a procedure `evaluate-request`, which accepts an arbitrary request and processes it as described above. Here is the output for the example of the previous section:

```
>(evaluate-request request-1)

Transmission allowed. Appropriate values
for missing parameters:
[(= (power ?t1) 20.0)]
```

4. COMPUTING OPTIMAL ADJUSTMENTS TO TRANSMISSION PARAMETERS

When a transmission request is denied, there are usually many different ways of modifying it so as to make it permissible. For instance, the solver could change the desired transmission's time; or it could change its frequency; or it could change its power and mode; or it could change all of the above. Some of these actions may be preferable to others. For instance, the radio might be less willing to change the time of the transmission, or its power level, rather than the frequency. In such cases we want the policy system to return a satisfying assignment that is optimal in the sense of respecting as many such preferences of the radio as possible.

Our system achieves this in a flexible way by formulating the problem as an (SMT) instance of Max-SAT. In its transmission request, the radio can provide a weight w_i along with the desired value v_i of each transmission parameter p_i. The weight w_i reflects the importance that the radio attaches to p_i taking the value v_i. The SMT solver will then try to find a satisfying assignment for the request that has maximal total weight. In that case, the request is not just a list of constraints `[`c_1 $\cdots$ c_n`]` but a list of pairs of constraints and weights `[[`c_1 w_1`]` $\cdots$ `[`c_n w_n`]]`. A sample transmission request might then be:

```
define weighted-request :=
  [[(frequency ?t = 8000) 10]
   [(power ?t = 35.0)     15]
   [(hours ?t = 13)       30]]
```

indicating that the relative importance of the frequency being 8000 is 10, that of the power being 35.0 is 15, and that of `hours` being `13` is 30. Suppose further that the policy in place has no DC and a disjunctive AC:

```
define AC1 :=
  (forall ?t . allow ?t <==>
     (frequency ?t in [5000 7000] &
      power ?t <= 30.0 & hours ?t > 12) |
     (frequency ?t in [6000 9000] &
      power ?t <= 40.0 & hours ?t <= 8))

define policy := (make-policy AC1 ())
```

Now consider evaluating `weighted-request` with respect to this policy. Clearly, the request cannot be allowed as is. We can make it permissible in more than one way: (a) we could demand a change of frequency and power in accordance with the values prescribed by the first branch of `AC1`, while keeping the `hours` parameter to the requested value of `13`; or (b) we could demand a change of the `hours` parameter only, in accordance with the second disjunctive branch of `AC`, while keeping the desired frequency and power values; or (c) we could demand that all three parameter values change. From these possibilities, only (a) is optimal, in that it honors the original request to the greatest extent possible (as determined by the given weights). Running this example in our system results in the following output:

```
>(evaluate-request weighted-request)

The following parts of the request
could not be satisfied:

(= (power ?t) 35.0) (= (frequency ?t) 8000)

Here is a maximally satisfying assignment:

[(= (mode ?t) everyDay)
 (= (frequency ?t) 5999)
 (= (power ?t) 30) (= (hours ?t) 13)]
```

By contrast, if we had changed the weight of the `hours` parameter from 30 to 20, the result would then change the `hours` parameter instead of the frequency and power, since retaining the values of the two latter parameters would result in a maximum weight of 25.

Note that weights can be arbitrary integers or a special "infinite" token, indicating a hard constraint that *must* be satisfied. In fact this infinite weight is the one that Athena attaches to the constraints obtained from the AC (and the negation of the DC) of the current policy. But radios can also use infinite weights in their requests.

Occasionally there may be additional requirements on the assignments returned by the policy system, beyond honoring the original request to the greatest extent possible. For example, if a parameter value must change, we may want the new value to be as close as possible to the original requested value. If we are not allowed to transmit at the exact desired time, for instance, we may want to transmit as close to it as possible (as can be allowed by the currently installed policies). To meet such requirements we generally need the ability to perform optimization by minimizing some objective function, in this case the absolute difference between desired and permissible parameter values. Most SMT solvers do not perform optimization (apart from Max-SAT, in the case of Yices, though see below), but we can efficiently implement an integer optimizer in terms of SMT solving. The idea is to use binary search in order to discover the optimal cost with as few calls to the SMT solver as possible: at most $O(\log n)$ calls, where n is the maximum value that the objective function can attain. Specifically, let c be an arbitrary constraint that we wish to satisfy in such a way that the value of some "cost" term t is minimized, where *max* is the maximum value that can be attained by the cost function (represented by t). (Tf this value is not known *a priori*, it can be taken to be the greatest positive integer that can be represented on the computer.) The algorithm is the following: We first try to satisfy c conjoined with the constraint that the cost term t is between 0 and half of the maximum possible value: $0 \leq t \leq (max \textbf{ div } 2)$. If we fail, we recursively call the algorithm and try to satisfy c augmented with the constraint $(max \textbf{ div } 2) + 1 \leq t \leq max$. Whereas, if we succeed, we recursively call the algorithm and try to satisfy c augmented with the constraint $0 \leq t \leq (max \textbf{ div } 4)$. Some care must be taken to get the bounds right on each call, but this algorithm is guaranteed to converge to the minimum value of t for which c is satisfied, provided of course that the original constraint c is satisfiable for *some* value of t. This algorithm is implemented by a ternary Athena procedure `smt-solve-and-minimize`, such that

```
(smt-solve-and-minimize c t max)
```

returns a satisfying assignment for `c` that minimizes `t` (whose maximum value is `max`).

For example, suppose that `x`, `y`, and `z` are integer variables to be solved for:

```
define [x y z] := [?x:Int ?y:Int ?z:Int]
```

subject to the following disjunctive constraint:

```
define c :=
  (x in [10 20] & y in [1 20] &
   z in [720 800]) |
  (x in [500 600] & y in [30 40] &
   z in [920 925])
```

Suppose also that the desired values are `x = 13`, `y = 15`, `z = 922`. The task is to find values for these variables that satisfy c while diverging from the desired values as little as possible. We can readily model this problem in a form that is amenable to `smt-solve-and-minimize` as follows. First we define the objective-function term t, as the sum of the three differences:

```
define t := (?d-x:Int + ?d-y:Int + ?d-z:Int)
```

with the difference terms defined as follows:

```
define d-x-def :=
  (ite (x > 13) (?d-x = x - 13)
                (?d-x = 13 - x))
define d-y-def :=
  (ite (y > 15) (?d-y = y - 15)
                (?d-y = 15 - y))
define d-z-def :=
  (ite (z > 922) (?d-z = z - 922)
                 (?d-z = 922 - z))
```

Here `ite` is a simple if-then-else procedure:

```
define ite :=
  lambda (c x y) ((c ==> x) &  (~ c ==> y))
```

Assume that we do not know the exact maximum value that `t` can attain, but we know that it cannot be more than 10^6. We can then solve the problem with the following call:

```
define diff-clauses :=
  (d-x-def & d-y-def & d-z-def)
define query := (c & diff-clauses)

>(smt-solve-and-minimize query t 1000000)

List: [(= ?x 13) (= ?y 15) (= ?z 800)
       (= ?d-x 0) (= ?d-y 0) (= ?d-z 122)]
```

This solution was found by a total of 8 calls to the SMT solver (for a total time of about 100 milliseconds). Why were only 8 calls required when we started the binary search with a maximum of 10^6? One would expect about $\log 10^6$ calls to the smt solver, i.e., roughly 20 such calls. However, our implementation uses the information returned by each call to the SMT solver to speed up the search. That often results in drastic shortcuts, cutting down the total number of iterations by more than a factor of 2.

This procedure enables our system to optimize any quantity that can be given an integer metric. Moreover, unlike independent branch-and-bound algorithms for integer programming, it has the advantage that it allows not just for numeric constraints, but for arbitrary boolean structure as well, along with constraints from other theories such as reals, lists, arrays, bit vectors, inductive datatypes, etc. While more sophisticated approaches to optimization in the SMT paradigm

are beginning to emerge (e.g., [2]), the implementation described here has been quite adequate for our purposes.

5. PERFORMANCE

In order to test our system we wrote an Athena program that generates test instances, with two independently controllable variables: parameter number and policy number. In particular, we defined a procedure `make-policy-set` with two parameters, `param-num` and `policy-num`. The output is a list of `policy-num` policies, where each policy involves `param-num` transmission parameters. The latter came from a pre-declared pool of 100 transmission parameters, half of them integer-valued and the other half real-valued. We distinguished the following types of policies:

- *permissive* policies, which have only an AC;
- *prohibitive* policies, which have only a DC and no AC;
- *mixed* policies, which have both.
- *Ordering*$/N$ policies. These policies are parameterized over $N > 0$. Specifically, an ordering/N policy is one whose AC and/or DC is of the following form:

  `(forall ?t:Transmission .` D `?t <==>`
  `(`f_{i_1} `?t` R_1 c_{i_1}`) & ... & (`f_{i_k} `?t` R_k c_{i_k}`))`

  where D is either `allow` or `deny`, and $\forall j \in \{1, \ldots, k\}$: f_{i_j} is a transmission parameter; $R_j \in \{<, >, <=, >=\}$; and c_{i_j} is a constant number of the appropriate sort. We refer to the ordering constraints `(`f_{i_j} `?t` R_j c_{i_j}`)` as the *atoms* of the AC (or DC). An ordering/N policy may be permissive, prohibitive, or mixed, but its total number of atoms (i.e., the number of the AC's atoms plus the number of the DC's atoms) must equal N.
- *Equational*$/N$ policies. The AC and/or DC of such a policy is of the same form as shown above, except that each R_i is the identity relation. These identities are the *atoms* of the condition. The total number of atoms (of the AC and DC together) must be N.
- *Inequational*$/N$ policies. Same as above, except that the relation in question here is inequality.
- *Range*$/N$ policies. The AC and/or DC of such a policy is of the same form as that of an ordering/N policy, except that each R_i is the relation `in`, and each constant c_i is a range `[`a b`]`.
- *Disjunctive-Conjunctive*$/N$ policies. The AC and/or DC of such a policy is a disjunction of conjunctions, where each atom is of the form `(`f_x `?t in [`$\cdots$`])`. The number of atoms are required to add up to N.

A call of the form `(make-policy-set N P)` returns a list L containing `P` policies, each of which involves `N` transmission parameters. This list is roughly equally partitioned among all the preceding types. Specifically, 20% of the policies in L are ordering/`N` policies. About 1/3 of these ordering/`N` policies are permissive, 1/3 are prohibitive, and 1/3 mixed. The next 20% of L consists of equational/`N` policies, and these are again evenly split between permissive-, prohibitive-, and mixed-policy subsets. The third 20% contains inequational/`N` policies, and so forth. Once a list L of policies is thereby obtained, we combine them all into a single policy by folding the `merge-policies` operator over L, with the `empty-policy` as the identity element. It is with respect to this merged policy that we tested transmission requests. The requests were generated randomly.

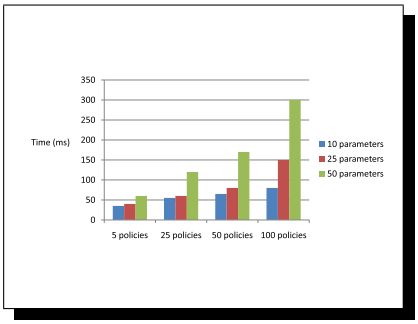


Figure 1: SMT solving times for processing regular transmission requests.

Figure 1 shows the SMT-solving times for processing plain transmission requests (i.e., without weights attached to the various transmission parameters), for various combinations of policy-set sizes and transmission parameter numbers, where the policy sets are evenly mixed as described above. Figure 2 shows the corresponding times for optimal processing of transmission requests, i.e., requests that attach weights to each of the transmission parameters and are solved as Max-SAT problems. In order to stress-test the implementation further, we repeated the experiments with sets of policies that were more structurally complex, doing away with simple ordering, equational, and inequational policies, and using instead only range and disjunctive-conjunctive policies. The corresponding results are shown in Figure 3 and in Fig-

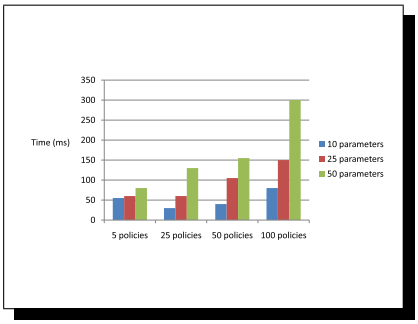


Figure 2: Max-SAT SMT solving times for optimized processing of transmission requests.

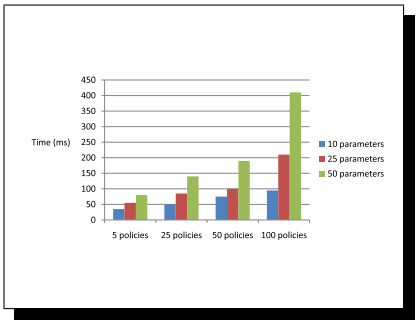


Figure 3: Solving times for processing of transmission requests with structurally complex policies.

ure 4. With these sort of structurally complex policies in place, the highest increase occurs in optimal processing of transmission requests with more than 50 policies and parameters.

The graphs show only the time spent on SMT solving, which is the bulk of the work that needs to be done when processing transmission requests. We do not include the time spent on translating from Athena to SMT notation and back. This translation is a straightforward linear-time algorithm (linear in the size of the formula to be translated, on average, since it uses hash tables for the necessary mappings between Athena and SMT namespaces), and the time spent on it in most cases is neglibible. For huge policies containing hundreds of thousands of nodes, the translation does take longer, though still typically a fraction of a second. Virtually all of the translation cost is incurred when translating the policy, since the transmission requests are tiny by comparison. Observe, however, that there is no reason to be translating the entire policy anew each time the engine needs to process a new request. Instead, the (merged) policy can be translated off-line, once, and subsequently cached in SMT notation in some file. Thus, the translation time is an one-time, off-line cost.

All experiments were performed on a 2.4 GHz Intel Core i3-370M dual-core processor with 4GB RAM, running Windows 7. The `time` command of the Cygwin shell was used

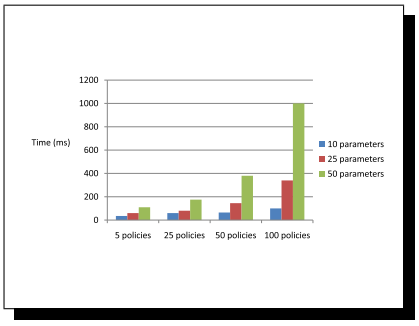


Figure 4: Max-SAT SMT solving times for transmission requests with structurally complex policies.

to get the timing data. In most cases, the reported system time (the `sys` entry of `time`'s output) was 0.000, i.e., too small to be reliably measured, and hence the time reported here is `real` (wall clock) time, which includes time segments spent by other processes and times during which the SMT solver was blocked (e.g., waiting for I/O to complete). Therefore, the times reported here are overly conservative; the actual times spent on SMT solving are smaller.

6. RELATED WORK & CONCLUSIONS

We have presented an implementation of a policy system for dynamic spectrum access. Policies are represented and manipulated in Athena, a formal proof system for first-order logic with polymorphism, subsorting, inductive datatypes, arbitrary computation, and definitional extension. Most of these features have proven useful in the engine's implementation; others would become useful if the engine were to be extended so that it used, e.g., structured ontologies instead of flat data types.

Previous work in this area includes BBN's XGPLF [8] and SRI's Coral [10, 11]. XGPLF does not have a formal semantics and is limited in what it can express (it is based on OWL [19]). Moreover, XGPLF cannot model inheritance. To the best of our knowledge, the only implementation of a policy reasoning engine based on XGPLF is the one built by the Shared Spectrum Company [12]. They used SWI-Prolog as the underlying reasoning engine. In field tests using resource-limited radio devices, SSC replaced the Prolog reasoner with "a simplified reasoner developed in C/C++" [12, p. 504]; no details are provided regarding the design, implementation, or correctness of this simplified reasoner. SSC's engine can only return yes/no answers to transmission requests.

Coral (Cognitive Radio Policy Language) is a new language specifically designed for expressing policies governing the behavior of cognitive radios. Like Athena, Coral offers a rich and extensible first-order logical language that includes arithmetic and allows for the introduction and definition of new function symbols. It also features subtyping and can therefore express inheritance and ontologies. Some notable differences from Athena include: (a) Unlike Coral, Athena has polymorphism. Thus, e.g., it is not necessary to introduce lists for integers and lists for booleans separately; a generic parameterized definition is sufficient. (b) Athena has automatic sort inference based on the Hindley-Milner algorithm, which is convenient in practice because it allows for shorter—and usually cleaner—specifications. (c) Athena has a general-purpose formally defined programming language that can seamlessly interact with its logical layer, and which can be used to dynamically construct and manipulate logical formulas very succinctly. This programming language offers procedural abstraction, as well as side effects through mutation, including reference cells and vectors, features which are often important for efficiency. The ability to compute with terms and sentences as primitive data

values was very useful. While Coral can express some computations via universally quantified identities which can then be interpreted as executable rewrite rules by a tool such as Maude [16], it does not offer procedural abstraction, i.e., it is not possible to define arbitrary procedures.

Two policy engines based on Coral have been implemented [10, 11]. The initial implementation used Prolog as the underlying reasoning engine, and was only able to return yes/no answers to transmission requests. The second (and current) implementation uses a custom-made proof system to reason about transmission requests. The system has four kinds of proof rules: forward chaining, backward chaining, partial evaluation based on conditional rewriting, and constraint propagation and simplification. The proof system is implemented in the rewriting engine Maude. One issue with this implementation is that a positive answer is given only if the transmission request is an *exact match* of the operative policy conditions. But that is not likely to be the case in practice. Most transmission requests are likely to be incomplete or to diverge from the policy, at least in some small degree. In such cases, the Coral implementation does not return values for the relevant transmission parameters. Rather, it returns an arbitrarily complicated first-order formula representing all the possible "opportunity constraints" that the radio could use to modify its request so as to make it permissible. But that is not likely to be of much use to the requesting party. It is not realistic to require that whoever made the transmission request should be able to understand and reason about arbitrarily complicated logical formulas in order to understand the policy system's output.

A second issue with this implementation is efficiency. The Coral team has not published precise benchmarks describing their engine's performance for variable numbers and types of policies (and for variable numbers of transmission parameters), but they have released a demo of their implementation that can be used to evaluate transmission requests with respect to policies that have a fixed set of transmission parameters, namely: location (latitude and longitude); time (hours, minutes, and seconds); sensed power; emissions; network id; and role (slave/master). Even with this fairly small set of transmission parameters, and with only 11 active policies in place, it has been reported [1, p. 18] that evaluating a single transmission request took the Coral engine 58 seconds, and resulted in an output formula comprising 75 constraints. In the preceding section we saw that our implementation can evaluate transmission requests with several dozens of parameters and with over 50 complex policies in place in a fraction of a second; this is so even when the requests are processed optimally. Moreover, the Coral engine has no notion of optimality. It does not allow the requesting party to specify that some parameters are more important than others, or to ask that the request should be satisfied to the greatest extent possible, or with as little divergence from the requested values as possible.

Another policy engine for spectrum access is Bresap, a system that was recently implemented at Virginia Tech [1, 5]. Unlike the other systems discussed above, Bresap is limited to policies that can be adequately represented as finite Boolean functions, which it encodes as binary decision diagrams (BDDs). Therefore, unlike Coral and our system, Bresap is not suitable as an expressive language for specifying policies. For instance, it has no facilities for introducing new policy concepts and/or rules. Indeed, Bresap is not a language. It is a system that reads certain types of policies expressed in XML and converts them to BDDs. This is done by assigning a distinct Boolean variable to each atomic expression that appears in the policy. For instance, a policy such as "allow transmission in the frequency range $a \dots b$ if the power does not exceed m" would result in the introduction of two distinct Boolean variables x_1 and x_2, with x_1 corresponding to the atom $a \leq f \leq b$ (where f stands for the transmission's frequency); and with x_2 corresponding to the atomic expression $p \leq m$ (where p stands for the transmission's power). The accepting condition could then be represented by the boolean function $x_1 \wedge x_2$. Graph algorithms are used to merge different policy BDDs. With a single BDD in place representing the result of merging all active policies, Bresap can then accept an incoming transmission request and evaluate it. A transmission request must be an attribute-value list of the form $a_1 = v_1, \dots, a_n = v_n$, where a_i is a transmission parameter (such as mode or frequency) and v_i is the corresponding desired value. Notably, Bresap is capable of handling underspecified (incomplete) requests. It is also capable of attaching costs to transmission parameters (where a given cost indicates the penalty paid for changing the value of that parameter), and then modifying the parameter values of a transmission request in a way that minimizes the overall cost.

A drawback of Bresap is that the underlying policy representation framework, that of BDDs (or propositional logic, more generally), lacks semantics for the numeric constraints that pervade spectrum access policies. To put it simply, BDDs do not know arithmetic, and thus do not have the wherewithal to "understand" that $<$ is transitive, that $1 + x$ is greater than x, that 10 and $14 - 4$ are the same object, and so on. Whereas an SMT solver immediately realizes that $f < 10$ conjoined with $f \geq 30$ is contradictory, because it uses a dedicated reasoning procedure for arithmetic, in the approach of Bresap these two atoms would result in the introduction of two Boolean variables, x_1 and x_2, the former representing $f < 10$ and the latter representing $f \geq 30$. Thus, the conjunction of x_1 and x_2 is represented by the innocuous-looking Boolean function $x_1 \wedge x_2$, a perfectly consistent condition. To represent the fact that in the present context this is actually inconsistent, one has to append to it ad hoc clauses such as $C = (\neg x_1 \vee \neg x_2) \wedge (x_1 \vee x_2)$. This extra information, C, conjoined with $x_1 \wedge x_2$, would then allow us to conclude that the condition is unsatisfiable. That is, in fact, what Bresap does: It "semantically analyzes" the various atomic expressions in the policy in order to generate ad-

ditional constraints—so-called "auxiliary rules" [1, section 3.5]—that prevent the engine from taking "illogical" paths in the BDD. This approach has some disadvantages.[3] First, it is an essentially ad hoc encoding of arithmetic facts, the generation of which should not be part of the policy engine's trusted computing base. Second, the additional encoded information, even if it is polynomial in size, can significantly blow up the resulting BDD. Other efficiency issues in Bresap include the conversion algorithm that produces the BDD from the atomic Boolean expressions of the policy. This algorithm has exponential time complexity in the number of expression variables. As a result, Bresap is not currently able to handle policies with more than 20 atomic Boolean expressions [1, p. 83].

In conclusion, we believe that the combination of a programmable and expressive formal framework such as Athena with an efficient SMT solver is a highly suitable implementation vehicle for spectrum access policy engines. It combines rich expressivity with efficient performance, a consideration that is likely to be crucial for cognitive radios. To the best of our knowledge, SMT solvers have not been used before for reasoning about policies, although we believe that they are ideal for this task. Indeed, we believe that there is not much that is special about spectrum access, and that the same approach we have introduced here could be used to represent and reason about policies in other domains.

7. REFERENCES

[1] A. A. Deshpande. Policy Reasoning for Spectrum Agile Radios. Masters thesis, Electrical and Computer Engineering Department, Virginia Tech, 2010.

[2] A. Cimatti and A. Franzn and A. Griggio and R. Sebastiani and C. Stenico. Satisfiability Modulo the Theory of Costs: Foundations and Applications. In *TACAS 2010*, pages 99–113, 2010.

[3] K. Arkoudas. Athena. `http://www.pac.csail.mit.edu/athena`.

[4] B. A. Fette. *Cognitive Radio Technology*. Academic Press, 2nd edition, 2009.

[5] B. Bahrak and A. A. Deshpande and M. Whitaker and J. M. Park. BRESAP: A Policy Reasoner for Processing Spectrum Access Policies Represented by Binary Decision Diagrams. In *New Frontiers in Dynamic Spectrum, 2010 IEEE Symposium on*, pages 1 –12, April 2010.

[6] B. Dutertre and L. de Moura. The Yices SMT Solver. Tool paper, available online from `yices.csl.sri.com/tool-paper.pdf`.

[7] C. Barrett and C. Tinelli. CVC3. In W. Damm and H. Hermanns, editors, *Proceedings of the* 19^{th} *International Conference on Computer Aided Verification (CAV '07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer-Verlag, 2007. Berlin, Germany.

[8] BBN Technologies. XG Working Group, XG Policy Language Framework, Request for Comments, version 1.0. `www.ir.bbn.com/projects/xmac/pollang.html`, 2004.

[9] Darpa. News Release for Next Generation (XG) Communications Program Request for Comments. `www.darpa.mil/news/2004/xg_jun_04.pdf`.

[10] G. Denker, D. Elenius, R. Senanayake, M.-O. Stehr, and D. Wilkins. A Policy Engine for Spectrum Sharing. In *New Frontiers in Dynamic Spectrum Access Networks, 2007. DySPAN 2007. 2nd IEEE International Symposium on*, pages 55 –65, April 2007.

[11] D. Elenius, G. Denker, M. O. Stehr, R. Senanayake, C. Talcott, and D. Wilkins. CoRaL–Policy Language and Reasoning Techniques for Spectrum Policies. *Policies for Distributed Systems and Networks, IEEE International Workshop on*, pages 261–265, 2007.

[12] F. Perich. Policy-Based Network Management for NeXt Generation Spectrum Access Control. In *New Frontiers in Dynamic Spectrum Access Networks, 2007. DySPAN 2007. 2nd IEEE International Symposium on*, pages 496 –506, April 2007.

[13] FCC Spectrum Policy Task Force. Report of the spectrum efficiency working group. Available from `www.fcc.gov/sptf/files/SEWGFinalReport_1.pdf`.

[14] I. F. Akyildiz and W.-Y. Lee and M. C. Vuran and S. Mohanty. NeXt Generation / Dynamic Spectrum Access / Cognitive Radio Wireless Networks: A Survey. *Computer Networks Journal (Elsevier)*, 50:2127–2159, September 2006.

[15] L. de Moura and B. Dutertre and N. Shankar. A Tutorial on Satisfiability Modulo Theories. In *Computer Aided Verification*, volume 4590 of *LNCS*, pages 20–36. Springer, 2007.

[16] M. Clavel and F. Durán and S. Eker and P. Lincoln and N. Martí-Oliet and J. Meseguer and C. Talcott. The Maude 2.0 System. In R. Nieuwenhuis, editor, *Rewriting Techniques and Applications (RTA 2003)*, number 2706 in LNCS, pages 76–87. Springer, 2003.

[17] Q. Zhao. A Survey of Dynamic Spectrum Access. *IEEE Signal Processing Magazine*, 24(3):79–89, 2007.

[18] R. Chadha and L. Kant. *Policy-Driven Mobile Ad hoc Network Management*. Wiley-IEEE Press, 2007.

[19] World Wide Web Consortium. OWL 2 Web Ontology Language Document Overview. Available from `www.w3.org/TR/owl2-overview/`, 2009.

[3] Note that this criticism is not peculiar to Bresap. It applies to all policy engines that represent semantically rich policies (requiring at least linear arithmetic) as Boolean functions.

Data Structure Choices for On-the-Fly Model Checking of Real-Time Systems

Peter Fontana
Department of Computer Science
University of Maryland, College Park
Email: pfontana@cs.umd.edu

Rance Cleaveland
Department of Computer Science
University of Maryland, College Park
Email: rance@cs.umd.edu

***Abstract*—This paper studies the performance of sparse-matrix-based data structures to represent *clock zones* (convex sets of clock values) in an on-the-fly predicate equation system model checker for timed automata. We analyze the impact of replacing the dense difference bound matrix (DBM) with both the linked-list CRDZone and array-list CRDArray data structure. From analysis on the paired-example-by-example differences in time performance, we infer the DBM is either competitive with or slightly faster than the CRDZone, and both perform faster than the CRDArray. Using similar analysis on space performance, we infer the CRDZone takes the least space, and the DBM takes less space than the CRDArray.**

I. Introduction

Automatic verification of real-time systems is undertaken using notations for verifiable programs and checkable specifications (see [1]–[6]). One common program notation is timed automata [7]. There are specification notations such as timed computation tree logic (TCTL) [1], [8] and timed extensions of a modal mu-calculus, including one in [3] and another given in [5]. Specifications in a timed modal mu-calculus can be written as lists of equations, known as timed modal equation systems [5], [6], [9]. For information on the untimed modal-mu calculi, see [10]–[12], and see [10], [11] for information on modal equation systems.

One approach to model checking timed automata with timed modal mu-calculus specifications is to use predicate equation systems (PES), which were invented independently by Groote and Willemse (as parameterized boolean equation systems) [13] and by Zhang and Cleaveland [6], [9]. Predicate equation systems provide a general framework for program models including parametric timed automata [6] and Presburger systems [14]. They also admit a natural on-the-fly approach to model checking based on proof search: a formula corresponding to the assertion that a timed automaton satisfies a mu-calculus property can be checked in a goal-driven fashion to determine its truth. Zhang and Cleaveland [6] demonstrated the efficiency of this approach *vis à vis* other real-time model-checking approaches.

In this paper we consider the special model checking case of timed automata and timed modal equation systems representing safety properties (also studied in [6]), for which there are still many opportunities for performance improvements. One component of such a model checker that has a noticeable influence on performance is the data structure for the sets of clock values. When analyzing safety properties, each desired set of clock values forms a convex set of clock values, or *clock zone* (see Definition 3). The conventional way to store a clock zone is as a *difference bound matrix* (DBM) (see Definition 4) [15], which stores the constraints as a matrix. This approach is used by UPPAAL [16] and described in [17]. To potentially save space and time, instead of representing the set of constraints as a matrix, one can represent the set as an ordered linked path of constraints where any clock difference not on the path has the implicit constraint $< \infty$. If we generalize this to allow for a union of zones to be represented by a directed graph of constraints (representing a tree of paths as opposed to a single path), we get a *clock restriction diagram* (CRD) [18]. If we compress the nodes to have them represent upper and lower bound constraints as well as explicitly encoding both valid and invalid paths, we get a *clock difference diagram* (CDD) [2]. These two structures are extensions of binary decision diagrams (BDDs) (see [19] for information).

To improve performance, we take the above idea of a linked implementation and incorporate the sparseness of the implementation of CRDs while simplifying (or shrinking) the structure to only support a single clock

Research supported by NSF Grant CCF-0820072.

zone (CRDs and CDDs in general can encode unions of clock zones). This simplified structure is a sparse sorted linked-list implementation of a DBM, the *CRDZone* (see Definition 5). We also implement an array-list version of the CRDZone, the *CRDArray* (see Definition 6). A CRDZone may be seen as a sparse sorted linked-list implementation of a DBM, and the CRDArray a sparse array-list implementation of the CRDZone. We examine the time and space performance of all three clock zone implementations: the matrix DBM, linked-list CRDZone and array-list CRDArray.

The contributions of this paper are:

- We run experiments comparing time and space performance of a model checker (on safety (reachability) properties) with the DBM, CRDZone and CRDArray data structure implementations.
- We formalize and extend the analysis style performed in the model checking experiments of [2], [6], [9], [18], [20], [21] by utilizing *paired data* (each implementation checked the same examples) and applying descriptive statistics on the paired example-by-example differences on time and space consumption. See Section VI for details on the statistics and Section VI-B for the analysis.

After analyzing the experimental results, for time performance we infer the DBM is either competitive with or slightly faster than the CRDZone and both perform faster than the CRDArray for the examples in this experiment. In terms of space, we infer the CRDZone takes up the least space, and the DBM and takes less space than the CRDArray for the examples in this experiment.

II. Program Model and Specifications

A. Timed Automata

A timed automaton encodes the behavior of a real-time system [7], [22].

Definition 1 (Clock constraint $\phi \in \Phi(CX)$)**.** Given a set of clocks CX, a *clock constraint* ϕ is constructed with the following grammar, where x_i is a clock and $c \in \mathbb{Z}$:

$$\phi ::= x_i < c \mid x_i \leq c \mid x_i > c \mid x_i \geq c \mid \phi \wedge \phi$$

$\Phi(CX)$ is the set of all possible clock constraints.

Definition 2 (Timed automaton)**.** A *timed automaton* $TA = (L, L_0, \Sigma, CX, I, E)$ is a tuple where:

- $\mathbf{L}$ is a finite set of locations with the initial set of locations $\mathbf{L_0} \subseteq L$.
- $\mathbf{\Sigma}$ is the set of actions and $\mathbf{CX}$ is the set of clocks.
- $\mathbf{I} : L \longrightarrow \Phi(CX)$ gives a clock constraint for each location l. $I(l)$ is called the *invariant* of l.
- $\mathbf{E} \subseteq L \times \Sigma \times \Phi(CX) \times 2^{CX} \times L$ is the set of *edges*. In an edge $e = (l, a, \phi, Y, l')$ from l to l' with action a, $\phi \in \Phi(CX)$ is the *guard* of e, and Y represents the set of clocks to *reset* to 0.

Some sources [6], [23] and our PES checker allow clock assignments ($x_1 := x_2$) in addition to clock resets on edges, other sources [17] allow constraints on clock differences and other sources [1] allow states to be labelled with atomic propositions that each state satisfies.

Timed automata use *clock valuations* $\nu \in \mathcal{V}$ ($\mathcal{V} = CX \longrightarrow \mathbb{R}^{\geq 0}$ is the set of all clock valuations), which at any moment stores a non-negative real value for each clock $x \in CX$. The semantics of a timed automaton are described as an infinite-state machine, where each state is a location-valuation pair (l, ν). Transitions represent either time advances or edge executions (performing an action). For a formal definition of the semantics of a timed automaton, see [7].

Example 1 (Example of a timed automaton)**.** Consider the timed automaton in Figure 1, which models a train in the generalized railroad crossing (GRC) protocol.

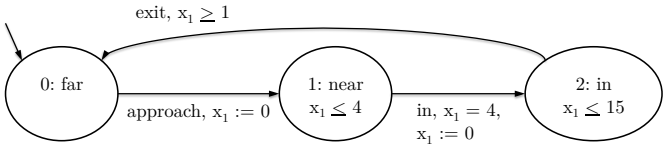


Fig. 1. Timed automaton TA_1, a model of a train in the generalized railroad crossing (GRC) protocol.

There are three locations—0: far (initial location), 1: near and 2: in, with one clock x_1. There are the actions $approach$, in and $exit$ in Σ. Here, location 1 has the invariant $x_1 \leq 4$ while 0 has no invariant. The edge $(1\text{: near}, in, x_1 = 4, \{x_1\}, 2\text{: in})$ has the guard $x_1 = 4$ and resets x_1 to 0.

B. Modal Equation Systems (MES)

We use a *modal equation system* (MES) to represent real-time temporal properties that timed automata can possess. A MES is an ordered list of equations with variables on the left hand side and basic timed temporal logical formulae on the right. Each equation involves a variable X, a basic formula ϕ and a greatest fixpoint (ν) or a least fixpoint (μ), and the equation is labeled with the fixpoint ($X \stackrel{\nu}{=} \phi$ or $X \stackrel{\mu}{=} \phi$). For a formal definition of MES syntax and semantics, see [6], [9].

Example 2 (Continuation of Example 1)**.** Again consider the timed automaton in Figure 1. The MES

$$X_1 \stackrel{\nu}{=} \quad \text{far} \ \wedge \forall([-](X_1)) \tag{1}$$

represents the safety property "the train is always in state 0: far", read as "the variable X_1 is the greatest fixpoint of being in state 0: far and for all time advances ($\forall$), for all next actions ($[-]$), X_1 is true." This is an invalid specification for the timed automaton because the execution

$$\begin{aligned}(0\text{: far}, [x_i = 0]) &\xrightarrow{2.5} (0\text{: far}, [x_i = 2.5]) \\ &\xrightarrow{approach} (1\text{: near}, [x_i = 0]) \xrightarrow{2} \dots \end{aligned} \tag{2}$$

reaches location 1: near and thus violates the property.

III. Data Structures for Clock Value Sets

A. Clock Zones

This definition of a clock zone is taken from [7], [19].

Definition 3 (Clock zone)**.** A *clock zone* is a convex combination of single-clock inequalities. Each clock zone can be constructed using the following grammar, where x_i and x_j are arbitrary clocks and $c \in \mathbb{Z}$:

$$\begin{aligned}Z ::= x_i < c \mid x_i \leq c \mid x_i > c \mid x_i \geq c \\ \mid x_i - x_j < c \mid x_i - x_j \leq c \mid Z \wedge Z\end{aligned} \tag{3}$$

Clock zones extend clock constraints with inequalities of clock differences. These inequalities are used for model checking even though clock difference inequalities are not used in timed automata. Moreover, in general, the representation of a clock zone is not unique.

Example 3. Let $z = 1 \leq x_1 < 3 \wedge 0 \leq x_2 \leq 5$. There is the *implicitly* encoded constraint $x_2 - x_1 \leq 4$. To see this, consider the longer path of constraints (x_0 is a dummy clock that always has value 0):

$$\begin{array}{rll} & x_2 - x_0 \leq 5 & (x_2 \leq 5) \\ + & x_0 - x_1 \leq -1 & (1 \leq x_1) \\ \hline & x_2 - x_1 \leq 4 & \end{array}$$

To provide a standardized, or canonical, form for clock zone representations, we use shortest path closure [17]. This form makes every implicit constraint explicit. This can be implemented in $O(n^3)$ time using Floyd-Warshall all-pairs shortest path algorithm, described in [24], [25]. Other standard forms exist [18], [20].

While converting to a canonical form takes a considerable amount of time, it is needed to simplify and standardize the algorithms for the zone operations including time successor ($succ(z)$) computations and subset checks. For time successor, having the zone in canonical form allows the time elapse operation to simply set all single-clock upper bound constraints to $< \infty$.

B. Clock Zone Data Structures: Difference Bound Matrix (DBM), CRDZone and CRDArray

One way to represent a clock zone is a *difference bound matrix (DBM)*, described in Definition 4. See [15], [17] for a more thorough description.

Definition 4 (Difference bound matrix (DBM))**.** Let $n - 1$ be the number of clocks. A *DBM* is an $n \times n$ matrix where entry (i, j) is the upper bound of the clock constraint $x_i - x_j$, represented as $x_i - x_j \leq u_{ij}$ or $x_i - x_j < u_{ij}$. The 0^{th} index is reserved for a dummy clock x_0, which is always 0, allowing bounds on single clocks to be represented by the clock differences $x_i - x_0$ and $x_0 - x_j$. See Figure 2 for a picture of the DBM structure and Example 4 for a concrete example.

Definition 5 (CRDZone)**.** A *CRDZone* is a sparse sorted linked-list representation of a clock zone. Each constraint is encoded like a constraint in a DBM, as an upper bound constraint on $x_i - x_j$, labeled as (i, j), with x_0 always being 0. The CRDZone has these properties:

1) Nodes are in lexicographical order of clock constraint: $(i_1, j_1) \prec (i_2, j_2)$ iff $i_1 < i_2$ or ($i_1 = i_2$ and $j_1 < j_2$).
2) The $(0, 0)$ node is always given to ensure a universal head node with an initial value of $x_0 - x_0 \leq 0$.
3) If a CRDZone node (i, j) is missing then the zone has an implicit constraint: $(i, i) : x_i - x_i \leq 0$ and $(i, j), i \neq j : x_i - x_j < \infty$.

See Figure 3 for a picture of the CRDZone structure and Example 4 for a concrete example.

This lexicographical ordering is the same ordering used in CDDs and CRDs [2], [18]. While the ordering

$$\begin{array}{c} \\ i \end{array} \overbrace{\begin{array}{c} j \\ \left\{ \begin{bmatrix} \dots & \dots & \dots \\ \dots & \dots & \dots \\ \dots & \dots & x_i - x_j \leq u_{ij} \end{bmatrix} \right. \end{array}}$$

Fig. 2. DBM: a matrix with constraint $x_i - x_j \leq u_{ij}$ in entry (i, j).

is conjectured to influence performance, this is the only ordering we implemented. Likewise, having different implicit constraints, such as making $x_i - x_0 \leq 0$ (for all i) implicit, is conjectured to influence performance.

Definition 6 (CRDArray)**.** The *CRDArray* is an array list implementation of the CRDZone. Thus, instead of using linked nodes, we use an array to store the nodes with the 0^{th} element being the head. We statically allocate the array to hold the maximum number of elements and store a back-pointer to the back of the array list. See Figure 4 for a picture of the CRDArray structure and Example 4 for a concrete example.

Using a dynamic allocation instead of our static allocation for the CRDArray array list is conjectured to save space at the expense of time.

Example 4 (Clock zone in various representations)**.** Consider the clock zone from Example 3, which is $z = 1 \leq x_1 < 3 \wedge 0 \leq x_2 \leq 5 \wedge x_2 - x_1 \leq 4$.

DBM representation of z:

$$\begin{bmatrix} x_0 - x_0 \leq 0 & x_0 - x_1 \leq -1 & x_0 - x_2 \leq 0 \\ x_1 - x_0 < 3 & x_1 - x_1 \leq 0 & x_1 - x_2 < \infty \\ x_2 - x_0 \leq 5 & x_2 - x_1 \leq 4 & x_2 - x_2 \leq 0 \end{bmatrix}$$

CRDZone representation of z:

$$\boxed{x_0 - x_0 \leq 0} \longrightarrow \boxed{x_0 - x_1 \leq -1} \longrightarrow \boxed{x_0 - x_2 \leq 0}$$

$$\longrightarrow \boxed{x_1 - x_0 < 3} \longrightarrow \boxed{x_2 - x_0 \leq 5} \longrightarrow \boxed{x_2 - x_1 \leq 4}$$

CRDArray representation of z:

$$[x_0 - x_0 \leq 0 | x_0 - x_1 \leq -1 | x_0 - x_2 \leq 0$$
$$| x_1 - x_0 < 3 | x_2 - x_0 \leq 5 | x_2 - x_1 \leq 4]$$

Remark 1 (On DBM vs. CRDZone and CRDArray methods). Due to the sparse implementation and removal of implicit nodes, the CRDZone and CRDArray can improve time by reducing the number of nodes, and thus the number of nodes looked at during a full traversal. This can speed up traversal-based algorithms such as intersect and subset check. However, algorithms like clock reset, emptiness check and canonical form use $O(1)$ access of middle nodes in DBMs (the CRDZone and CRDArray do not have $O(1)$ access for all nodes), resulting in a performance slowdown for those CRDZone and CRDArray methods. For space, the CRDZone and CRDArray can store fewer nodes but must store the explicit indices, resulting in more space per node.

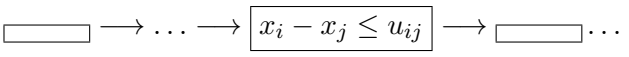


Fig. 3. CRDZone: A linked list with nodes in lexicographical order of constraint $x_i - x_j \leq u_{ij}$.

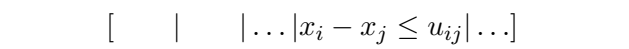


Fig. 4. CRDArray: An array list with nodes in lexicographical order of constraint $x_i - x_j \leq u_{ij}$.

IV. On-The-Fly Model Checking: Converting to a PES and Proof Search

Our model checker takes in a *predicate equation system* (PES) (taken from [9], [13]), which is a general framework representing logical expressions that involve fixpoints and first order quantifiers. We take a timed automaton and a MES and convert it to a PES. Currently the PES model checker can only check safety properties, which involve only greatest fixpoints in both the PES and the input MES. For more information on a PES, including its syntax, semantics and how to convert a timed automaton and a MES to a PES, see [6], [9].

The model checker takes the conclusion sequent (the sequent we wish to prove) and applies proof rules in a recursive goal-driven tree-like fashion on the premise sequents, trying to prove each premise sequent until it reaches a proof rule with no premise (called a leaf) or a circularity (a previously seen premise sequent). When checking a proof, we will often encounter circularity. In general, when the circularity reached is a greatest fixpoint, we can stop and declare the proof branch valid. For the formal conditions for circularity and the proof rules, see [6], [9].

V. Experiments: Various Data Structure Implementations

We compare the DBM implementation to the CRDZone and CRDArray implementations. Each implementation uses shortest path closure to compute canonical form. The only difference in the DBM, CRDZone and CRDArray versions is the data structure implementation.

Remark 2 (On our analysis approach). We ask: *what does it mean for an implementation to perform better than another?* **We consider consider *better* to be measured in the number (or percentage) of examples that one system outperforms another in.** The larger aim is for any implementation, if we were to know all the examples that it would run (including and beyond the experiment examples), we would *like* one implementation to perform

(strictly) better for at least 51% of this hypothetical set. This influences our analysis.

Given our meaning of *better* in Remark 2, we consider the median, 25% and 75% percentile values as *insights* into typical examples and use the histograms to get a bigger picture of the sample distribution of the performance differences for the experiment, and weigh these more heavily than the mean and standard deviation values. The mean and the standard deviation provide us with an alternative picture of the overall performance and give hints of either a unusual sample distribution (since in a symmetric distribution the mean equals the median) or the presence of potential outliers.

The benchmark choice was modeled off of [6], with the addition of a model of the generalized railroad crossing (GRC) protocol [26]. We also used all the protocols in [6], which are the Carrier Sense, Multiple Access with Collision Detection (CSMA/CD), the Fiber Distributed Data Interface (FDDI), Fischer's Mutual Exclusion (FISCHER), the Leader Election protocol (LEADER and LBOUND) and the PATHO Operating System (PATHOS) protocol, where each of these protocols is described some in [6]. There are 53 benchmarks that ran on each implementation.

Experiments were run on a Linux machine with a 3.4 GHz Intel Pentium 4 Dual Processor (each a single core) with 4 GB RAM. Time and space measurements (maximum space used) were made using the `memtime` (http://www.update.uu.se/~johanb/memtime/) tool (using time `elapsed` and `Max VSize`). The data tables are in the Appendix.

TABLE I
DESCRIPTIVE STATISTICS FOR PAIRED DBM − (MINUS) CRDZONE EXAMPLES, FOR TIME (S) AND SPACE (MB).

Statistic	DBM − CRDZone (Time)	DBM − CRDZone (Space)
Mean	-67.55	34.96
Standard Deviation	428.35	212.65
25% Percentile	-1.24	0.00
Median	0.00	1.85
75% Percentile	0.06	25.70

TABLE II
DESCRIPTIVE STATISTICS FOR PAIRED DBM − (MINUS) CRDARRAY EXAMPLES, FOR TIME (S) AND SPACE (MB).

Statistic	DBM − CRDArray (Time)	DBM − CRDArray (Space)
Mean	-112.95	-47.75
Standard Deviation	655.65	235.63
25% Percentile	-3.16	-20.54
Median	-0.29	-2.81
75% Percentile	0.00	-0.01

TABLE III
DESCRIPTIVE STATISTICS FOR PAIRED CRDZONE − (MINUS) CRDARRAY EXAMPLES, FOR TIME (S) AND SPACE (MB).

Statistic	CRDZone − CRDArray (Time)	CRDZone − CRDArray (Space)
Mean	-45.40	-82.71
Standard Deviation	229.06	160.91
25% Percentile	-2.02	-52.67
Median	-0.21	-19.35
75% Percentile	-0.03	-1.63

VI. STATISTICS, ANALYSIS AND DISCUSSION

A. Histograms and Descriptive Statistics

Running the different data structure implementations with the same examples yields *paired data*. Hence, we can take the two implementations and pair them example-by-example on their time and space differences to analyze their performance. When we pair the DBM − CRDZone samples, we take the DBM measurement and subtract the CRDZone measurement for the same example to get a DBM − CRDZone paired data point. For instance, the MUX-5-a paired point is -0.92s, 1.94MB, since the DBM point is 1.22s, 14.67MB, and the CRDZone point is 2.14s, 12.73MB. Pairings are likewise done to obtain the paired samples for DBM − CRDArray and CRDZone − CRDArray. For more information, see a Statistics text such as [27].

Tables I, II and III contain descriptive statistics on the paired difference in example-by-example performance of the DBM, CRDZone and CRDArray. Figures 5, 6 and 7 have histograms that plot the overall time and space differences between the DBM, CRDZone and CRDArray implementations. Bin colors and are changed to help more easily find the -0.001 to 0.001 (equal performance, since our measurement precision is 0.01 units), and -0.25 to -0.001 and 0.001 to 0.25 bins (slight differences).

We do not use 95% confidence intervals, paired two-sample hypothesis (z) tests or ANOVA (Analysis of Variance) because the independence assumption of the samples (the example benchmarks) does not hold. Furthermore, we do not use a Wilcoxon signed-rank test for the median because the symmetry assumption of the distribution is not believed to hold, and thus we cannot analyze the hypothetical benchmark distribution referred to in Remark 2. We do use paired sampling since we have its only requirement—perfect correlation of the samples. More information is in [27].

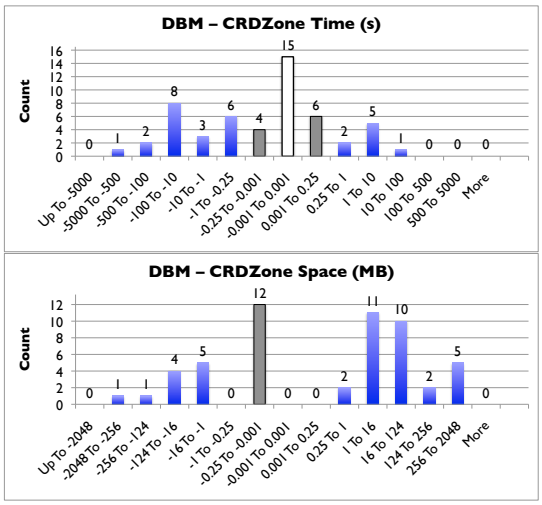


Fig. 5. Histograms comparing the DBM − (minus) CRDZone time (s) (top) and space (MB) (bottom) differences.

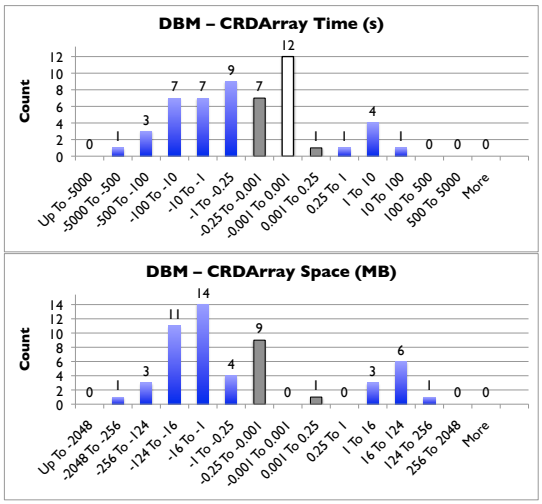


Fig. 6. Histograms comparing the DBM − (minus) CRDArray time (s) (top) and space (MB) (bottom) differences.

B. Analysis of Results

1) DBM vs. CRDZone: The CRDZone performs slower for 45% of the tested examples (at least as slow for 74%) with a median difference of 0.00s slower, while the CRDZone has a mean difference of 67.55s slower. Thus, we infer the CRDZone is either slightly slower or competitive to the DBM for this benchmark, but due to insufficient evidence (45% of the examples is not enough) do not infer that the DBM performs strictly faster than the CRDZone.

The CRDZone takes less space for 57% of the tested examples (at most as much space for 57%) with a median amount of 1.85MB less space and a mean amount of 34.96MB less space. The CRDZone takes at least

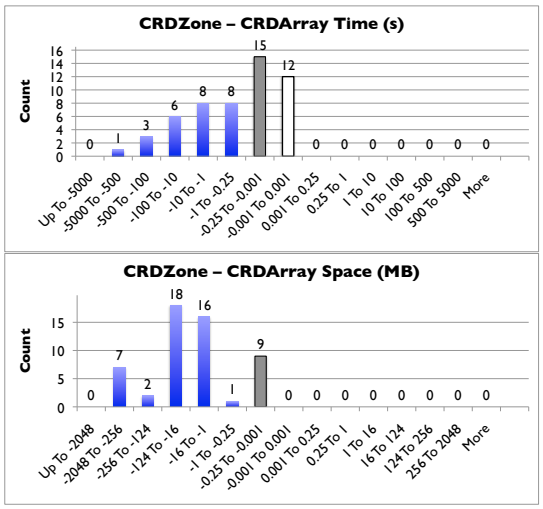


Fig. 7. Histograms comparing the CRDZone − (minus) CRDArray time (s) (top) and space (MB) (bottom) differences.

0.25MB less space for 28 such examples and more than 0.25MB space for only 11 examples. Thus (even though 57% is not a large majority), we infer the CRDZone takes less space overall for this benchmark.

2) DBM vs. CRDArray: The CRDArray performs slower for 64% of the tested examples (at least as slow for 89%) with a median difference of 0.29s slower and a mean difference of 112.95s slower. Thus we infer the CRDArray performs slower overall for this benchmark.

The CRDArray takes more space for 77% (at least as much space for 77%) of the examples with a median amount of 2.81MB more space and mean amount of 47.75MB more. Thus we infer the CRDArray takes more space overall for this benchmark.

3) CRDZone vs. CRDArray: The CRDArray performs slower for 77% of the tested examples (at least as slow for 100%) with a median difference of 0.21s slower and a mean difference of 45.40s slower. Thus we infer the CRDArray is slower overall for this benchmark.

The CRDArray takes more space for 100% of the examples with a median amount of 19.35MB more space and a mean amount of 82.71MB more. Thus we infer the CRDArray takes more space overall for this benchmark.

C. Discussion of Results

The CRDZone and CRDArray method that converts zones to canonical form was implemented using array-based algorithms, where it temporarily converts the clock zone to and from a matrix (the algorithm is similar to a DBM algorithm for those methods). It is possible that performance can be improved by trying an algorithm that

does not require copying to and from a matrix. All time vs. space tradeoffs were taken to save time.

Furthermore, we ran a CRDZone execution twice (first execution reported) and compared the distribution of their differences to get an idea of noise and/or measurement error. The differences in the histograms are larger than the differences of the noisy implementation, so thus we suspect the differences in performances are due to more than just uncertain measurement/noisy data. However, slight differences ($\leq$ 0.25s or $\leq$ 0.25MB) may be due to execution noise or examples that require very little time and/or space. The PATHOS-7-b is one such resource-light example, taking at most 0.10s and 2.89MB for each implementation. In contrast, the MUX-7-a example is resource-heavy, being the only example to takes more than 2000s for each implementation.

When profiling the implementations using `gprof` (http://www.gnu.org/s/binutils/), CRDZones take more time than DBMs for clock resets and emptiness checks, but the invariant checking method (mostly clock zone intersections) takes less time for CRDZones than DBMs.

From the data (see the Appendix), we notice that the data structure implementation choice has a noticeable influence on model checking performance for specific examples. To see this, consider the examples MUX-7-a, where the DBM finished 3118.94 seconds earlier than the CRDZone, and LEADER-100-c, where the CRDZone checked the example in 55% of the time required for the DBM. There are also noticeable differences relative to the CRDArray.

D. Related Work

A different way of modeling programs using discrete-time is discussed in [4]. PES are in [9], and they have been used to model check various systems in [6], [9], [14]. Difference bound matrices originated from [15] and are used in various studies such as in those in [17], [23]. Other tools that can model check timed automata (safety and liveness properties) include *KRONOS* [23], *UPPAAL* [16], *RED* [18] and *Rabbit* [28]. We built upon Zhang's implementation using predicate equation systems in [6], [9], which supports safety properties. A similar experiment involving using a reduced canonical form is in [20], which focuses on the influence of different standard clock zone forms instead of comparing list vs. array implementations.

A remark on a sparse DBM representation saving space, with neither a mention on time nor experimental data, is given in [17]. There is an experiment comparing CDDs to another BDD-like structure used by *Rabbit* in [21], but this experiment compares data structures for non-convex sets of clock valuations (unions of clock zones), and the comparison is across different tools with different model checking approaches, and not the same tool with different data structures.

VII. Conclusions and Future Work

Here are our conclusions from the experiment:

1) **Time:** (DBM $\leq_t$ CRDZone) $<_t$ CRDArray). For this benchmark, we infer that the DBM is either competitive with or slightly faster than the CRDZone and both perform faster than the CRDArray. There is insufficient evidence to conclude that the DBM is strictly faster.
2) **Space:** (CRDZone $<_s$ DBM) $<_s$ CRDArray. For this benchmark, we infer that the CRDZone takes the least amount of space and the DBM takes less space than the CRDArray for this experiment.

For potential reasons for performance differences and analysis of some theoretical differences, see Remark 1.

Future work is to model check least fixpoint (μ) equations, to implement all of the proof rules given in [6], [9] and to model check any PES. Comparing lists of DBMs to a CRD or CDD to improve the performance of the expanded implementation is future work, as well as comparing different kinds of standard forms and different CRDZone node orderings.

Acknowledgements

We thank Dezhuang Zhang for providing the code base [6] and for his insights.

References

[1] R. Alur, C. Courcoubetis, and D. Dill, "Model-Checking in Dense Real-Time," *Inf. Comput.*, vol. 104, pp. 2–34, 1993.

[2] G. Behrmann, K. G. Larsen, J. Pearson, C. Weise, and W. Yi, "Efficient Timed Reachability Analysis Using Clock Difference Diagrams," in *CAV '99*, vol. 1633. Springer Berlin / Heidelberg, 1999.

[3] T. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine, "Symbolic Model Checking for Real-time Systems," *Inf. Comput.*, vol. 111, pp. 193–244, 1994.

[4] L. Lamport, "Real-Time Model Checking Is Really Simple," *Correct Hardware Design and Verification Methods*, pp. 162 –175, 2005.

[5] O. V. Sokolsky and S. A. Smolka, "Local model checking for real-time systems," in *CAV '95*. Springer-Verlag, 1995, pp. 211–224.

[6] D. Zhang and R. Cleaveland, "Fast Generic Model-Checking for Data-Based Systems," in *FORTE*, F. Wang, Ed., vol. 3731. Springer, 2005, pp. 83–97.

[7] R. Alur, "Timed Automata," in *CAV '99*. London, UK: Springer-Verlag, 1999, pp. 8–22.

[8] C. Baier and J.-P. Katoen, *Principles of Model Checking*. Cambridge, MA: The MIT Press, 2008.

[9] D. Zhang and R. Cleaveland, "Fast On-the-Fly Parametric Real-Time Model Checking," in *RTSS '05*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 157–166.

[10] J. Bradfield and C. Stirling, "Local Model Checking for Infinite State Spaces," *Theor. Comput. Sci.*, vol. 96, no. 1, pp. 157–174, 1992.

[11] R. Cleaveland and B. Steffen, "A Linear-Time Model-Checking Algorithm for the Alternation-Free Modal Mu-Calculus," *Form. Methods Syst. Des.*, vol. 2, no. 2, pp. 121–147, 1993.

[12] E. A. Emerson and C. L. Lei, "Efficient Model Checking in Fragments of the Propositional Mu-Calculus," in *LICS '86*. IEEE Computer Society Press, 1986, pp. 267–278.

[13] J. F. Groote and T. A. Willemse, "Parameterised Boolean Equation Systems," *Theoretical Computer Science*, vol. 343, no. 3, pp. 332 – 369, 2005.

[14] D. Zhang and R. Cleaveland, "Efficient Temporal-Logic Query Checking for Presburger Systems," in *ASE '05*. New York, NY, USA: ACM, 2005, pp. 24–33.

[15] D. L. Dill, "Timing Assumptions and Verification of Finite-State Concurrent Systems," in *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*. London, UK: Springer-Verlag, 1990, pp. 197–212.

[16] G. Behrmann, A. David, and K. G. Larsen, "A Tutorial on Uppaal," in *Formal Methods for the Design of Real-Time Systems*, vol. 3185. Springer Berlin / Heidelberg, 2004, pp. 200–236.

[17] J. Bengtsson and W. Yi, "Timed Automata: Semantics, Algorithms," in *Lecture Notes in Computer Science*, vol. 3098. Springer, 2004, pp. 87–124.

[18] F. Wang, "Efficient Verification of Timed Automata with BDD-Like Data-Structures," in *VMCAI 2003*. London, UK: Springer-Verlag, 2003, pp. 189–205.

[19] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. The MIT Press, 1999.

[20] K. Larsen, F. Larsson, P. Pettersson, and W. Yi, "Efficient Verification of Real-Time Systems: Compact Data Structure and State-Space Reduction," in *RTSS 1997*. IEEE Computer Society, December 1997, pp. 14–24.

[21] D. Beyer and A. Noack, "Can decision diagrams overcome state space explosion in real-time verification?" in *FORTE 2003*. Springer Berlin / Heidelberg, 2003, vol. 2767, pp. 193–208.

[22] E.-R. Olderog and H. Dierks, *Real-Time Systems: Formal Specification and Automatic Verification*. Cambridge University Press, 2008.

[23] S. Yovine, "Model Checking Timed Automata," in *Lectures on Embedded Systems, European Educational Forum, School on Embedded Systems*. London, UK: Springer-Verlag, 1998, pp. 114–152.

[24] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, *Network Flows: Theory, Algorithms and Applications*. Prentice-Hall, 1993.

[25] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd ed. MIT Press, 2001.

[26] C. L. Heitmeyer, B. G. Labaw, and R. D. Jeffords, "A Benchmark for Comparing Different Approaches for Specifying and Verifying Real-Time Systems," Naval Research Laboratory, Tech. Rep. ADA462244, 1993.

[27] J. L. Devore, *Probability and Statistics for Engineering and the Sciences*, 6th ed. Duxbury Press, 2003.

[28] D. Beyer, C. Lewerentz, and A. Noack, "Rabbit: A tool for bdd-based verification of real-time systems," in *CAV '03*. Springer Berlin / Heidelberg, 2003, vol. 2725, pp. 122–125.

TABLE IV
EXPERIMENT RESULTS—A EXAMPLES—TIME (S): CORRECT SYSTEM, CORRECT SPECIFICATION.

Example	DBM	CRDZone	CRDArray
CSMACD-3-a	**0.10**	0.20 (200%)	0.20 (200%)
CSMACD-4-a	**3.16**	4.48 (142%)	6.50 (206%)
FDDI-20-a	**2.04**	3.03 (149%)	4.66 (228%)
FDDI-40-a	**58.49**	79.2 (135%)	126.82 (217%)
FDDI-50-a	**169.66**	230.7 (136%)	370.71 (219%)
MUX-5-a	**1.22**	2.14 (175%)	2.75 (225%)
MUX-6-a	**35.49**	74.44 (210%)	98.08 (276%)
MUX-7-a	**2623.61**	5742.55 (219%)	7383.73 (281%)
LEADER-6-a	**0.41**	0.71 (173%)	0.92 (224%)
LEADER-7-a	**12.99**	25.89 (199%)	34.22 (263%)
LBOUND-6-a	**0.51**	1.02 (200%)	1.32 (259%)
LBOUND-7-a	**17.36**	37.07 (214%)	49.64 (286%)
PATHOS-4-a	**13.7**	35.23 (257%)	50.58 (369%)
GRC-3-a	**0.92**	1.63 (177%)	2.12 (230%)
GRC-4-a	**252.05**	431.63 (171%)	748.01 (297%)

TABLE V
EXPERIMENT RESULTS—A EXAMPLES—SPACE (MB): CORRECT SYSTEM, CORRECT SPECIFICATION.

Example	DBM	CRDZone	CRDArray
CSMACD-3-a	**2.88**	7.55 (262%)	11.02 (382%)
CSMACD-4-a	209.97	**104.47** (50%)	179.53 (86%)
FDDI-20-a	**5.96**	9.00 (151%)	13.57 (227%)
FDDI-40-a	**27.55**	57.24 (208%)	100.30 (364%)
FDDI-50-a	**53.91**	116.79 (217%)	209.29 (388%)
MUX-5-a	14.57	**12.73** (87%)	18.55 (127%)
MUX-6-a	**84.05**	116.35 (138%)	168.38 (200%)
MUX-7-a	**625.42**	1667.94 (267%)	2302.39 (368%)
LEADER-6-a	**3.57**	6.59 (185%)	7.82 (219%)
LEADER-7-a	**20.98**	104.02 (496%)	133.39 (636%)
LBOUND-6-a	**3.93**	8.66 (220%)	10.39 (264%)
LBOUND-7-a	**27.89**	157.54 (565%)	199.99 (717%)
PATHOS-4-a	40.73	**38.11** (94%)	57.45 (141%)
GRC-3-a	10.48	**7.87** (75%)	11.23 (107%)
GRC-4-a	318.22	**220.64** (69%)	355.02 (112%)

APPENDIX
EXPERIMENTAL DATA

For the experiments, we use three kinds of examples:

- **Valid A Examples (in Tables IV and V):** Correct system implementations with valid safety specifications.
- **Invalid B Examples (in Tables VI and VII):** A examples with invalid specifications.
- **Invalid C Examples (in Tables VIII and IX):** A examples with buggy implementations of the systems that do not satisfy the A specifications.

TABLE VI
EXPERIMENT RESULTS—B EXAMPLES—TIME (S): CORRECT SYSTEM, INVALID SPECIFICATION.

Example	DBM	CRDZone	CRDArray
CSMACD-4-b	**0.10**	**0.10** (100%)	0.20 (200%)
CSMACD-5-b	**0.51**	**0.51** (100%)	0.71 (139%)
CSMACD-6-b	3.35	**2.73** (81%)	3.97 (119%)
FDDI-30-b	**1.53**	**1.53** (100%)	2.23 (146%)
FDDI-40-b	4.66	**4.58** (98%)	6.60 (142%)
FDDI-60-b	8.64	**5.07** (59%)	5.48 (63%)
MUX-20-b	**0.41**	**0.41** (100%)	0.51 (124%)
MUX-30-b	0.92	**0.91** (99%)	1.21 (132%)
MUX-40-b	1.93	**1.73** (90%)	2.23 (116%)
LEADER-10-b	**0.10**	**0.10** (100%)	**0.10** (100%)
LEADER-20-b	**0.10**	0.20 (200%)	0.20 (200%)
LBOUND-10-b	**0.10**	**0.10** (100%)	0.20 (200%)
LBOUND-40-b	6.82	17.46 (256%)	29.54 (433%)
PATHOS-7-b	**0.10**	**0.10** (100%)	**0.10** (100%)
PATHOS-8-b	**0.10**	**0.10** (100%)	**0.10** (100%)
PATHOS-9-b	**0.10**	**0.10** (100%)	**0.10** (100%)
GRC-3-b	**0.10**	**0.10** (100%)	**0.10** (100%)
GRC-4-b	**0.51**	0.61 (120%)	0.82 (161%)
GRC-5-b	**9.75**	13.4 (137%)	19 (195%)

TABLE VII
EXPERIMENT RESULTS—B EXAMPLES—SPACE (MB): CORRECT SYSTEM, INVALID SPECIFICATION.

Example	DBM	CRDZone	CRDArray
CSMACD-4-b	**2.88**	2.89 (100%)	13.67 (474%)
CSMACD-5-b	144.14	**72.38** (50%)	123.52 (86%)
CSMACD-6-b	1134.30	**553.21** (49%)	961.90 (85%)
FDDI-30-b	9.60	**9.06** (94%)	19.08 (199%)
FDDI-40-b	17.19	**16.03** (93%)	39.00 (227%)
FDDI-60-b	27.85	**14.53** (52%)	63.52 (228%)
MUX-20-b	19.37	**11.15** (58%)	16.96 (88%)
MUX-30-b	28.28	**16.87** (60%)	28.58 (101%)
MUX-40-b	43.01	**21.85** (51%)	42.81 (100%)
LEADER-10-b	**2.88**	2.89 (100%)	2.89 (100%)
LEADER-20-b	**2.88**	4.59 (159%)	5.69 (197%)
LBOUND-10-b	**2.88**	2.89 (100%)	3.38 (117%)
LBOUND-40-b	18.29	**15.23** (83%)	30.73 (168%)
PATHOS-7-b	**2.88**	2.89 (100%)	2.89 (100%)
PATHOS-8-b	**2.88**	2.89 (100%)	2.89 (100%)
PATHOS-9-b	**2.88**	2.89 (100%)	2.89 (100%)
GRC-3-b	**2.88**	2.89 (100%)	2.89 (100%)
GRC-4-b	58.74	**32.08** (55%)	53.42 (91%)
GRC-5-b	717.21	**379.44** (53%)	648.00 (90%)

TABLE VIII
EXPERIMENT RESULTS—C EXAMPLES—TIME (S): BUGGY SYSTEM, CORRECT SPECIFICATION.

Example	DBM	CRDZone	CRDArray
CSMACD-6-c	0.51	**0.41** (80%)	0.51 (100%)
CSMACD-7-c	2.03	**1.82** (90%)	2.03 (100%)
CSMACD-8-c	9.55	**8.42** (88%)	9.55 (100%)
FDDI-30-c	0.51	**0.41** (80%)	0.41 (80%)
FDDI-40-c	1.52	**0.92** (61%)	1.01 (66%)
FDDI-60-c	6.71	**3.98** (59%)	4.17 (62%)
MUX-6-c	**139.02**	258.32 (186%)	401.84 (289%)
LEADER-60-c	6.81	**3.96** (58%)	4.06 (60%)
LEADER-70-c	14.42	**8.12** (56%)	8.13 (56%)
LEADER-100-c	82.94	**45.78** (55%)	45.88 (55%)
LBOUND-6-c	**0.10**	**0.10** (100%)	0.20 (200%)
LBOUND-7-c	**0.61**	0.81 (133%)	1.12 (184%)
LBOUND-8-c	**12.48**	32.00 (256%)	52.9 (424%)
PATHOS-5-c	**0.10**	**0.10** (100%)	**0.10** (100%)
PATHOS-6-c	**0.10**	**0.10** (100%)	**0.10** (100%)
PATHOS-7-c	**0.10**	**0.10** (100%)	**0.10** (100%)
GRC-3-c	**0.10**	**0.10** (100%)	**0.10** (100%)
GRC-4-c	**0.51**	0.81 (159%)	1.02 (200%)
GRC-5-c	**9.65**	13.31 (138%)	18.88 (196%)

TABLE IX
EXPERIMENT RESULTS—C EXAMPLES—SPACE (MB): BUGGY SYSTEM, CORRECT SPECIFICATION.

Example	DBM	CRDZone	CRDArray
CSMACD-6-c	85.32	**18.53** (22%)	79.30 (93%)
CSMACD-7-c	337.43	**191.84** (57%)	320.89 (95%)
CSMACD-8-c	1369.07	**787.75** (58%)	1338.62 (98%)
FDDI-30-c	4.88	**4.60** (94%)	9.93 (203%)
FDDI-40-c	9.55	**6.41** (67%)	19.63 (206%)
FDDI-60-c	24.07	**14.24** (59%)	55.57 (231%)
MUX-6-c	1607.73	**1047.64** (65%)	1723.25 (107%)
LEADER-60-c	29.24	**10.79** (37%)	63.66 (218%)
LEADER-70-c	51.18	**15.30** (30%)	108.80 (213%)
LEADER-100-c	203.89	**40.65** (20%)	431.71 (212%)
LBOUND-6-c	**2.88**	2.89 (100%)	4.48 (155%)
LBOUND-7-c	12.52	**10.34** (83%)	14.42 (115%)
LBOUND-8-c	75.66	**59.23** (78%)	90.48 (120%)
PATHOS-5-c	**2.88**	2.89 (100%)	2.89 (100%)
PATHOS-6-c	**2.88**	2.89 (100%)	2.89 (100%)
PATHOS-7-c	**2.88**	2.89 (100%)	2.89 (100%)
GRC-3-c	**2.88**	2.89 (100%)	2.89 (100%)
GRC-4-c	58.74	**35.94** (61%)	59.47 (101%)
GRC-5-c	717.29	**379.35** (53%)	647.94 (90%)

The experimental data for the 53 example benchmarks is provided in Tables IV, V, VI, VII, VIII and IX, with the best entry(ies) in each row **bolded** and percentage change relative to the DBM, to the nearest %, in parenthesis. Time data is given to the nearest 0.01s (second) and space data is given to the nearest 0.01MB (Megabyte). Given the percentage rounding, sometimes an example with slightly different performance may still have a 100% value.

metaSMT: Focus On Your Application Not On Solver Integration

Finn Haedicke Stefan Frehse Görschwin Fey Daniel Große Rolf Drechsler
Institute of Computer Science
University of Bremen, 28359 Bremen, Germany
{finn,sfrehse,fey,grosse,drechsle}@informatik.uni-bremen.de

***Abstract*—Decision procedures are used as core technique in many applications today. In this context, automated reasoning based on Satisfiability Modulo Theories (SMT) is very effective. However, developers have to decide which concrete engine to use and how to integrate the engine into the application. Even if file formats like SMT-LIB standardize the input of many engines, advanced features remain unused and the integration of the engine is left to the programmer.**

This work presents *metaSMT*, a framework that integrates advanced reasoning engines into the program code of the respective application. *metaSMT* provides an easy to use language that allows engine independent programming while gaining from high performance reasoning engines.

State-of-the-art solvers for satisfiability and other theories are available for the user via *metaSMT* with minimal programming effort. For two examples we show how *metaSMT* is used in current research projects.

I. Introduction

In recent years, *formal methods* have become attractive to solve complex computational hard problems. Decision procedures are applied in many applications, like e.g., *Model Checking* [1], [2], *Synthesis* [3], [4] and, *Automatic Test Pattern Generation* (ATPG) [5].

Despite the successful research and application of decision procedures, the increasing complexity of software and hardware systems demands for more effective reasoning engines to overcome complexity issues. In the last years, solvers for *Satisfiability Modulo Theories* (SMT) have been developed. Different theories are combined to formulate the problem. Various works gave empirical evidence that SMT reasoning engines increase the efficiency of formal methods [6], [7], [8].

The performance of SMT reasoning engines remains an active research topic. Annual SMT competitions [9], [10] show their advances. However, SMT reasoning engines have different strengths on different problem instances. Therefore, evaluating different engines with respect to a given problem instance allows to find the best performing engine.

When using SMT in a concrete algorithm, the most common way is to generate a problem instance in SMT-LIB format [11]. Taking a user created SMT-LIB file as input, an SMT solver decides whether the instance is satisfiable or unsatisfiable. However, many solvers additionally have custom native interfaces. These interfaces are used to pass the instance to the engine and check for satisfiability. Furthermore, advanced features are available, e.g., computing interpolants which are utilized in *SAT-based Model Checking* [2]. Moreover learnt information generated while reasoning can be reused very efficiently in consecutive reasoning processes to prune the search space [12]. Usually this can only be done by calling native functions which access the learnt information.

This work presents *metaSMT* a publicly available, easy to use and powerful tool[1] which provides an integration of the native *Application Programming Interface* (API) of modern reasoning engines into C++ code. The advantages of *metaSMT* are: (1) engine independence through efficient abstraction layers (2) simple use of various decision procedures (3) extensibility in terms of input language and reasoning engines (4) customizability in terms of optimization and infrastructure (5) translation of the input language into native engine calls at compile time.

The remaining work is structured as follows: Section II gives a basic introduction into SMT and the programming methods used in *metaSMT*. Afterwards an example of a *metaSMT*-based application is given before in Section IV the architecture of *metaSMT* is described. Section V describes how this architecture is implemented in *metaSMT* and Section VI gives an empirical evaluation of *metaSMT* including its use in current research projects. The work closes with conclusions.

II. Preliminaries

This section provides background information. However, basic knowledge of C++ is assumed.

A. Satisfiability Modulo Theories

Boolean satisfiability is a decision problem, also known as the SAT problem. The problem asks whether there exists an assignment of Boolean variables such that the Boolean function evaluates to true. The problem has been proven NP-complete [13]. In spite of the huge complexity of the problem, sophisticated algorithms and clever heuristics help to solve instances with many thousands variables and clauses very efficiently. Usually, SAT solvers work on a *Conjunctive Normal Form* (CNF) of a Boolean function that is a disjunction of conjunctions of literals, where each literal is variable or its negation.

Satisfiability Modulo Theories is also a decision problem but with more complex theories rather than only propositional logic. A detailed introduction is given

This work was supported in part by the German Federal Ministry of Education and Research (BMBF) within the project SANITAS under contract no. 01M3088.

[1] Available online at http://www.informatik.uni-bremen.de/agra/eng/metasmt.php

Listing 1. SMT instance for $a \cdot b = 21466342967$

```
(benchmark factorization.smt
  :logic QF_BV
  :extrafuns ((a BitVec[32]))
  :extrafuns ((b BitVec[32]))

  :assumption (not (= a bv1[32]))
  :assumption (not (= b bv1[32]))

  :formula (=
    bv21466342967[64]
    (bvmul
      (zero_extend[32] a)
      (zero_extend[32] b)
  ))
)
```

in [14]. Already available SMT-solvers handle complex formulas. In addition to the logics the SMT-LIB standard also specifies a textual format that is commonly used for input files of the solvers. Listing 1 shows an example of such an SMT file: Two variables, `a`, `b` are declared as bit-vectors and constrained to be the two factors of a product resulting in the 64 bit number $21,466,342,967$. When called with this input, an SMT-QF_BV solver outputs satisfiable and e.g., the assignment $a = 740,218,723$ and $b = 29$.

Moreover, in addition to the SMT-LIB format many solvers provide an *Application Programming Interface* (API) that exposes features like incremental solving, where the SMT instance can be changed after the satisfiability check. Learnt information about the instance is kept and reused. Consequently, using the API may increase the overall performance.

An SMT solver can be utilized within an application in different ways. Each of the following options has certain advantages and disadvantages:

1) Generate an instance file according to SMT-LIB specification and call the solver. Different SMT solvers can easily be evaluated. However, the instance generation and result retrieval requires handling of files and text within the application.
2) Use the solver specific API to call an SMT solver's functions directly. In particular, incremental satisfiability can be exploited. But the application is restricted to a specific solver.
3) Introduce an abstraction layer specific to the application. This technique combines high performance using incremental satisfiability with the ability to evaluate different solvers. However a custom layer is not portable to different applications.

This work separates the programming model from the reasoning engine. With *metaSMT* the application specifies the instance in a simple, common notation. Many reasoning engines are available without modifications of the algorithm.

The Java package jSMTLIB [15] provides an interface for the usage of different SMT solvers. The package reads, checks and generates SMT-LIB files and executes the respective SMT-solver executables. However, jSMTLIB currently does not provide an embedded language for instance generation directly from the application.

B. Boost.Proto

The Boost project [16] is a collection of libraries that cover many features not included in the C++ standard library. In particular, Boost.Proto [17] provides tools to integrate *Domain Specific Embedded Languages* (DSEL) into C++. Given a programming language, a DSEL in that language is dedicated to a domain, e.g., parsing [18] or vector arithmetic [19]. The DSEL provides a syntax designed for this domain and, therefore, is easier to handle than the original language. Due to space constraints an in-depth description of Boost.Proto is omitted, the reader is referred to the Boost documentation.

Technically, this work uses Boost.Proto to implement a domain specific language for SMT logics in *metaSMT*.

III. Motivating Example

Before the subsequent sections give a complete description of *metaSMT* this section demonstrates how *metaSMT* is used in a complete example application.

The usual integration of reasoning engines iteratively calls API functions to construct the problem instance. However, this reduces the readability of the source code and makes it difficult to understand the program. A typical example can be constructed using the Boolector API for the C programming language. Figure 1 (a) shows the code to construct the simple constraint $c = a \cdot b$ using the Boolector C API (not including memory management). The same expression, written quite concise in the SMT-LIB format is shown in Figure 1 (b). The goal of *metaSMT* is to allow this compact syntax to be used in C++ programs. Due to the limitations of C++ this requires adaptions. Most notably, the expressions cannot easily be written in a symbolic expression (S-expression) syntax as in the SMT-LIB format, where each function is enclosed in parenthesis. The syntax for calling functions is used instead. Moreover, the solver is passed into the expression using the context e.g., as `btor_ctx` in Figure 1 (c).

In order to illustrate the programming interface of *metaSMT* an example written in C++ code is presented in Listing 2. The listing shows the factorization of an integer into two integers. More precisely, given an integer in bit-vector representation $\vec{c} \in \mathbb{B}^{2n}$, compute two integers $\vec{a}, \vec{b} \in \mathbb{B}^n$, such that $\vec{a} \cdot \vec{b} = \vec{c}$. To enforce non-trivial factorization, both integers a and b may not be 1. As the bit-vectors $\vec{a}$ and $\vec{b}$ are zero-extended, no overflows are possible and as a result a valid assignment to the constraint either gives a valid factorization of $\vec{c}$ or proves that it is prime. The integer value of $\vec{c}$ is randomly chosen and changed in each iteration of a loop for 10,000 iterations. A similar example is reconsidered in the empirical evaluation.

The first line in Listing 2 defines the solver context, which is not explained here but described later in this work. The context specifies which reasoning engine to use and how the input is handled by *metaSMT*. The next line is a user parameter, which defines the bit-vector width of the operands. The algorithm is therefore scalable to an abitrary bit-width. In lines 4-6 the bit-vectors `a`, `b` and `c` are declared and initialized.

Lines 8 and 9 constrain the operands to be different from 1 in bit-vector representation. In line 11 the

```
boolector_assume( btor,
  boolector_eq(btor, c,
   boolector_mult(btor, a, b)
  ))
```

(a) Boolector API calls

```
:assumption
    (= c (bvmul a b))
```

(b) SMT-Lib format

```
assumption( btor_ctx,
    equal(c, bvmul(a, b)) )
```

(c) metaSMT C++ Code

Fig. 1. Examples for $c = a \cdot b$

Listing 2. *metaSMT* factorization and prime test

```
1  Context ctx;
2  const unsigned width = <parameter>;
3
4  bitvector a = new_bitvector(width);
5  bitvector b = new_bitvector(width);
6  bitvector c = new_bitvector(width);
7
8  assertion(ctx, nequal(a, bvuint(1,width)) );
9  assertion(ctx, nequal(b, bvuint(1,width)) );
10
11 assertion( ctx, equal( zero_extend(width, c)
       , bvmul( zero_extend(width, a),
       zero_extend(width, b)) ));
12
13 for (unsigned i=0; i < 10000; ++i) {
14   unsigned r = random_number ( 2, 2^width -
         1  );
15   assumption( ctx, equal(c, bvuint(r, 2*
         width)) );
16
17   if( solve( ctx ) ) {
18     unsigned a_value = read_value( ctx, a );
19     unsigned b_value = read_value( ctx, b );
20
21     printf("factorized %d into %d * %d\n", r
           , a_value, b_value);
22   } else {
23     printf("%d is prime.", r);
24   }
25 }
```

multiplication $\vec{a} \cdot \vec{b} = \vec{c}$ is constrained. However the multiplication is done in double width to avoid overflows.

These constraints are identical for each iteration of the loop starting in line 13, therefore they are declared outside the loop as an `assertion`, which is permanent.

Inside the loop, in lines 14-15 `c` is set equal to a random number from 2 to $2^{\text{width}} - 1$ using an `assumption`, which is only valid for the next satisfiability check of the solver, i.e., for one loop iteration.

After setting up the SMT instance, the satisfiability check is performed in line 17. If the instance is satisfiable, the values of `a` and `b` are determined using `read_value`. Both operands are printed out in line 21. Otherwise the instance is unsatisfiable, the else branch is executed, which outputs `c` is prime.

IV. Architecture

In the following sections the architecture of *metaSMT* is described. At first the basic layers are introduced. Then each layer is described in detail. The terms *frontend* for the input languages, *middle-end* for the intermediate layer and *backend* for the solvers are taken from compiler design to denote the *metaSMT* layers.

A. metaSMT *Layers*

metaSMT consists of three basic layers depicted in Figure 2. The frontend layer provides primitives of

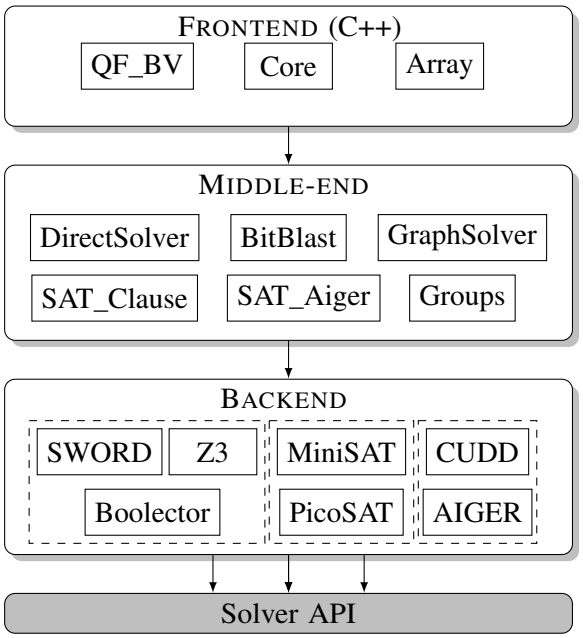


Fig. 2. *metaSMT* layer Architecture

the input languages, defined in the SMT-LIB format (e.g., Core Boolean logic, QF_BV). The middle-end layer provides translations, intermediate representation and optimizations of the input expressions. Optionally, expressions can directly be passed to the the backend layer where the solvers are integrated via their native API. Various configurations of middle-ends with backends are possible. The frontends allow to combine each translation middle-end with any compatible backend. However, not every backend supports every logic. Therefore, some middle-ends supply emulation or translations to other logics, e.g, a bit-vector expression can be translated into a set of Boolean expressions.

The frontends are independent from the underlying two layers and have no semantics attached. To evaluate frontend expressions, a *context* is used that defines their meaning. The context is the combination of at least one middle-end and one backend, where the middle-end defines how the input language is mapped to function calls of the backend.

B. Frontends

The frontends define the input languages for *metaSMT*. This includes Core Boolean logic and SMT QF_BV as well as a version of Array logic over bit-vectors. Each frontend defines its own set of available functions as well as public datatypes.

The Core Boolean logic defines the public datatype `predicate` which describes propositional variables.

Furthermore, Boolean constants are available, i.e., `true` and `false`. This logic also defines primitive Boolean functions, e.g., `Or`, `And`. The frontend creates a static syntax tree for the expression described in the code. This syntax tree is passed to the middle-end.

C. Middle-ends

The core of *metaSMT* are basic optimizations and translations from the frontend to the backend. While the frontends provide languages and the backends provide solver integrations, the middle-ends allow the user to customize *metaSMT*, i.e., on how the input language is mapped to the backend. Even in the middle-end itself, several modules can be combined.

1) DirectSolver: To enable a low-overhead translation from a frontend to a backend the `DirectSolver` is provided. All the elements of the input expression are directly evaluated in the backend. Variables are guaranteed to be constructed only once and are stored in a lookup table. For example, given a multiplication operation in QF_BV logic directly corresponds to a multiplication operation in the SMT solver Boolector.

The direct middle-end is very lightweight and allows the compiler to inline all function calls. For a modern compiler the resulting executable should perform equally well to a hand-written application using the same backend.

2) GraphSolver: Instead of adding the frontend expressions directly to the solver, they are first inserted into a directed graph. The graph models the explicit syntax tree of the expression as a *Directed Acyclic Graph* (DAG). Formally a node in the graph is a tuple $(\text{Operation}, \text{Static Arguments})$ where the SMT command and its static arguments are captured (e.g. `extract` and the range to extract). The edges point from an operation to the SMT expressions used in this command. A label on the edges stores the position of the subexpression in the command. Each time a new expression is evaluated it is first searched in a lookup table before a new node is created, when the node is not found. When the instance is checked for satisfiability, the graph is traversed and evaluated in the backend.

The graph-based translation provides a way to automatically detect common subexpressions and efficiently handle them to create smaller SMT instances which potentially increases performance of the reasoning process. This is especially useful if the user wants to automate this process, but either does not want to manually optimize the SMT instance or does not know the instance in advance because it is created dynamically inside the program.

3) Groups: This middle-end provides an interface and implementation of *constraint groups* for solvers that do not have native support for groups. A group is a set of constraints that belong together. The user can create groups, add expressions to them and delete them at any time. The solver will then disable all expressions in the group. Groups are emulated using *guard* variables and temporary assumptions, e.g., the expression $x \wedge y$ in group 1 is transformed to $g_1 \rightarrow (x \wedge y)$ using the guard variable g_1 and an implication. Depending on the solver deleting a group can either lead to the removal of the constraints or to the constraint just being disabled permanently.

4) BitBlast: This emulation of a QF_BV bit-vector backend uses only Core Boolean logic operations to allow the transparent use of SAT or BDD solvers with bit-vector expressions. The translation is performed in a standard way: Given only the Core Boolean logic, each bit-vector is transformed into a vector of Boolean variables. The bitwise operations can be applied easily, e.g., an exclusive-or over two bit-vectors is a bitwise exclusive-or for each pair of Boolean variables. The bit-vector predicates (`equal`, `less-than`, etc.) are mapped to a sequence of Boolean predicates, e.g., a conjunction of exclusive-nors for `equal`. Arithmetic operations are reduced to an equivalent Boolean expression.

D. Backends

The respective solvers and other constraint solving techniques are integrated as backends. For each reasoning engine a dedicated backend is created that maps from the internal *metaSMT* API to the API of the engine. Backends do not have an explicit interface to inherit from. They implement the required methods for the languages they support using C++ template mechanisms to integrate them into a context. This allows the compiler to optimize the code and, in the case of `DirectSolver`, produces code that is close to a hand-coded implementation using the same API.

This section gives an overview of the backends integrated into *metaSMT*. They are grouped by the input language they support. The compatibility of the solvers is also summarized in Table I.

1) Core Boolean logic backends: Several core logic backends as well as higher level backends are available. Core logic is directly supported by backends that accept all common Boolean operations. For example, the *Binary Decision Diagram* (BDD) package CUDD [20] supports all Boolean operations and is integrated in *metaSMT*. Furthermore, with some minor transformations based on De-Morgan *And-Inverter-Graph*s (AIGs) are also able to handle Boolean operations. Those AIGs are internally represented by the AIGER package [21]. SAT solvers can receive Boolean logic expressions either via the *SAT_Clause* adapter that creates one or more clauses per logic operation or via the *SAT_Aiger* adapter, that builds an AIG for the expression using the AIGER backend. Afterwards, the AIG is translated into a set of clauses. This infrastructure allow the usage of any SAT solver supporting CNF as input language either by an API or externally through files. PicoSAT [22] as well as MiniSAT [23] are directly supported as Core logic backends via their APIs. Other solvers are supported by generating CNF files and calling the executable of the SAT solvers.

Furthermore, all SMT QF_BV backends natively support Core logic as a subset of the language.

2) SMT QF_BV backends: Native SMT bit-vector solvers like Boolector [24], SWORD [25] and Z3 [26] are directly connected through their API for QF_BV support. Furthermore, the BitBlast middle-end provides an emulation for QF_BV using only basic logic operations. This emulation permits using QF_BV expressions in solvers that do not support them natively but support Core Boolean logic e.g., CUDD or SAT-solvers.

TABLE I
BACKEND COMPATIBILITY

BACKEND	CORE	QF_BV	ARRAY (BV)	SAT
AIGER [21]	yes	*emulated*	no	no
Boolector [24]	yes	yes	yes	no
CUDD [20]	yes	*emulated*	no	no
MiniSAT [23]	*emulated*	*emulated*	no	yes
PicoSAT [22]	*emulated*	*emulated*	no	yes
SWORD [25]	yes	yes	no	no
Z3 [26]	yes	yes	yes	no
BitBlast	yes	yes	no	no
SAT_Aiger	yes	*emulated*	no	no
SAT_Clause	yes	*emulated*	no	no

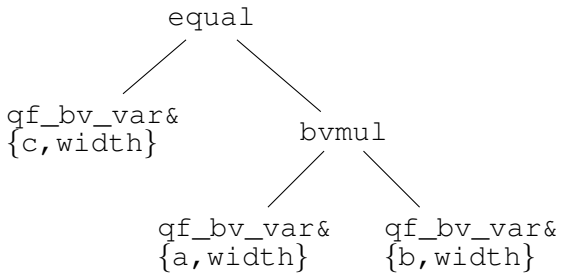


Fig. 3. Syntax Tree for `equal(c, bvmul(a, b))`.

3) SMT QF_ABV backends: In addition to Core Boolean logic and bit-vector logic the Boolector and Z3 backends also support arrays in the form of QF_QBV logic. Therefore *metaSMT* supports declaring and working with arrays over bit-vectors.

V. IMPLEMENTATION

This section describes how the architecture is implemented in *metaSMT* and how *metaSMT* is integrated in C++ programs.

A. Syntax and Semantics

For the evaluation of *metaSMT* expressions a context is used which defines syntax and semantics. The context concept and different kinds of contexts are described in this section.

The syntax component is provided by Boost.Proto. An expression like `equal(c, bvmul(a, b))` is created from the custom Boost.Proto functions `equal` and `bvmul` as well as the variables `a`, `b` and `c`. From the expression the syntax tree in Figure 3 is created. The nodes are labeled with the C++ *type* and strings inside the curly braces denote the content of the respective nodes. For *metaSMT* the tree is used as static type of the expression. The expression and the syntax tree are data, i.e., they neither have semantics attached nor trigger any actions.

The semantics for the expression is introduced by the *metaSMT context*, that defines how the syntax tree is evaluated and transformed for a specific solver. The evaluation of Boost.Proto-based expressions is performed in the *metaSMT* translation middle-end (e.g., GraphSolver or DirectSolver) so that the backends do not need to handle Boost.Proto expressions directly. This reduces the overhead to implement new backends.

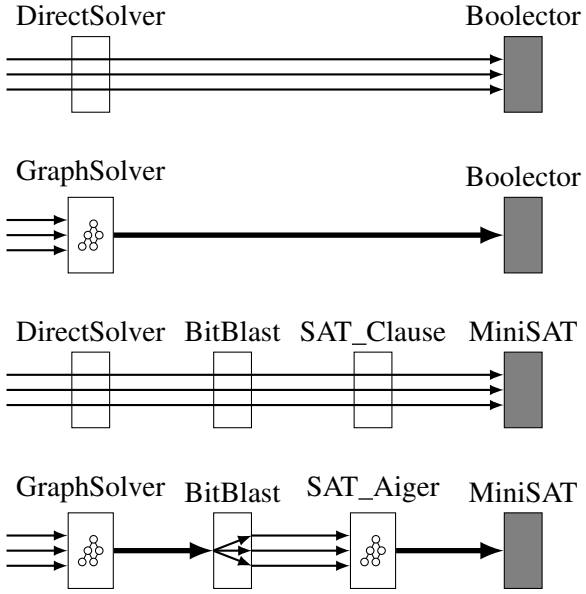


Fig. 4. Data flow in different contexts

Listing 3. *metaSMT* command grammar

```
command     ::= assert_cmd | assume_cmd
              | eval_cmd
              | solve_cmd  | result_cmd
assert_cmd  ::= 'assertion('  context ','
                expression ');'
assume_cmd  ::= 'assumption(' context ','
                expression ');'
eval_cmd    ::= 'evaluate('   context ','
                expression ');'
solve_cmd   ::= 'solve(' context ');'
result_cmd  ::= 'read_value(' context ','
                variable ');'
variable     ::= boolean_variable
               | bitvector_variable
expression  ::= <expression in metaSMT DSEL>
```

Figure 4 gives some example contexts and visualizes the data flow inside. This illustrates how different contexts can change the evaluation of a constraint. The first context defines a solver using Boolector without intermediate representation (DirectSolver). The context directly supports Core Boolean logic and QF_BV. In contrast, in the last example, MiniSAT is used. QF_BV as well as Core Boolean logic are emulated for this clause based backend. Furthermore this context uses a graph and an AIG as intermediate representations.

The GraphSolver-based and AIGER-based context first create an internal representation and pass the complete expression directly before solving. When using approaches without intermediate representation, the requests are forwarded to the next layer until they reach the backend. Only explicit transformations are applied before passing the expression (e.g., BitBlast, SAT_Clause).

B. Usage and API

The example from Listing 2 contains most of the core commands of *metaSMT*. These are summarized in Listing 3.

Listing 4. Programmatic constraint construction using temporary Variables

```
bitvector x = new_bitvector(8);
for( ... ) {
  bitvector tmp = x;
  x = new_bitvector(8);
  assertion(ctx, equal( x, bvmul(tmp, ...)));
}
...
solve(ctx)
```

The first three functions accept frontend expressions, however they have different effects. The functions `assertion` and `assumption` create the constraint instance where the first adds a constraint permanently to the (incremental) solver context while the latter adds the constraint for the next call of the solver only. In both cases the expression needs to have a Boolean result. The third function `evaluate` does not change the instance but returns a context specific representation of the input expression only.

To query a context for satisfiability, the `solve` function is provided. The result is a Boolean value directly representing `SAT` (true) or `UNSAT` (false). After a call to `solve` the assumptions are discarded while the assertions are still valid for the subsequent calls.

Getting a `SAT` result for `solve(ctx)`, i.e., the instance is satisfiable, a model is generated. The model can be retrieved with the `read_value` function. The function takes a context and a variable and returns the assignment of this variable in the given context. The result of `read_value` is automatically convertible to many C++ datatypes, including strings, bit-vectors (vector of `bool`, `tribool`, `bitset`) or integers.

In addition to these core commands, custom middle-ends may provide additional extensions. The *Group* middle-end for example provides functions to add groups, change the active group and delete groups. These functions cannot be used in any other context.

C. Expression Creation

Typically it is necessary to create the *metaSMT* expression at run time, e.g., in a loop. As *metaSMT* syntax trees are statically typed, an extension of the syntax tree is not possible. To work around this limitation, *metaSMT* provides two options. The first option is to create a partial expression and constrain equality to a temporary variable that is later reused to create the complete expression. This would allow strict grammar checking but introduces a temporary variable and a constraint, see Listing 4.

The second option is the use of the `evaluate(Ctx, Expr)` function and the context's `result_type`. The function takes a context and a frontend expression and returns the context specific representation of the expression. The result of the evaluation is of the backend specific type `Ctx::result_type`. This expression can be stored and later be used in other expressions. Note however that the return value is solver specific and therefore not portable or reusable in other contexts, not even contexts of the same type.

A powerful exception to this rule is the `result_type` of a `GraphSolver`-based context, where the result is a node in the internal graph.

Listing 5. Using a shared graph for different contexts

```
GraphSolver_Context<SWORD> sword;
GraphSolver_Context<Boolector> btor(sword);

GraphSolver_Context<SWORD>::result_type x =
    evaluate(sword, bvuint(0, 8));

for( ... ) {
  x = evaluate(sword, equal( x, bvmul(x,
      ...)));
}
assertion(sword, x);
assertion(btor, x);
solve(sword) == solve(btor);
```

When a `GraphSolver` is constructed using the copy constructor, a shared graph is internally used by the contexts. The newly created solver also copies all assertions and assumptions, so that both solvers have the same internal state. In this setup the results of `evaluate` can be shared among the solvers. Each backend will only evaluate the parts of the graph that are required as parts of assertions or assumptions. The application of `evaluate` is demonstrated in Listing 5. This can be used for example when building multiple instances from the same base. At a specific point the context can be copied and from there both contexts can diverge into seperate instances.

VI. Emperical Evaluation

This section presents two different applications of *metaSMT*. Furthermore, a comparison of *metaSMT* with the native API of an SMT solver is presented. The experiments have been performed on a system with AMD Opteron 2222 SE processor, 32 GB RAM and a Linux operating system. In the following, the run times are always given in CPU seconds.

A. Exact Synthesis

This section presents examples from exact synthesis of reversible circuits [28], [29]. A reversible circuit computes a bijective function, i.e., there is a unique input and output mapping. Those circuits purely consist of reversible gates – here, the universal Toffoli gate and basic quantum gates are considered. The synthesis problem searches for a reversible circuit consisting of reversible gates which computes the function specified by a truth table. There are several exact approaches to synthesize those circuits. We considered the approach from [28], which translates the problem into a decision problem and asks for a circuit realization for a given number of gates. The size of the problem instances grows exponentially with number of input variables, because the entire truth table has to be taken into account. This usually results in hard problem instances even for small truth tables.

The underlying problem formulation has been encoded in QF_BV and an incremental algorithm searches for a valid circuit implementation. Using *metaSMT* sixteen different configurations have been evaluated. The configurations consist of four internal API backend solvers, i.e. Boolector, Z3, MiniSAT, and PicoSAT. Additionally, metaSMT is used to generated CNF files to run the external solvers PicoSAT, MiniSAT, PrecoSAT, and Plingeling [30]. All eight backends are used

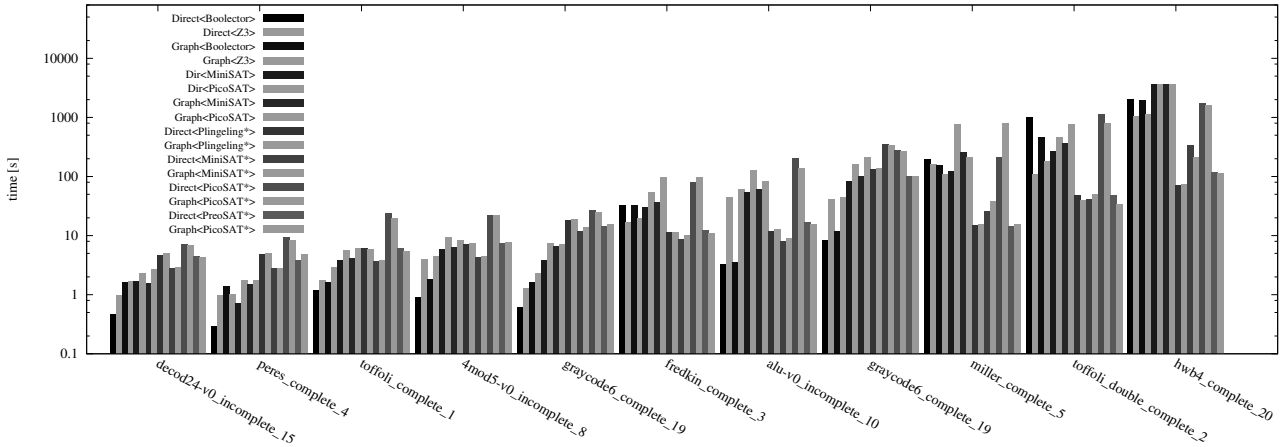


Fig. 5. Run times of reversible circuit synthesis for hard instances using a *metaSMT*-based version of RevKit [27]

with the `DirectSolver` middle-end as well as the `GraphSolver` middle-end.

For the synthesis 11 specifications were used which are publicly available from [31]. A timeout of 3,600 seconds was applied. The results are presented in Figure 5. On the x-axis the respective benchmark is shown, whereas the y-axis denotes the run time in seconds for each configuration in logarithmic scale. Externally called solvers are marked with $*$.

From Figure 5 it is clear that no single solver dominates the benchmarks. For example, for the benchmarks from `decode24-v0_incomple_5` to `graycode6_complete_19` Boolector performs much better than any other solver. But for the benchmarks from `miller_complete_5`, Boolector is outperformed by the SAT solvers. MiniSAT as well as PicoSAT are evaluated as internal and external versions. The accumulated run times of the solvers MiniSAT and PicoSAT over all benchmarks are 18,790 seconds for the internal version and 8,989 seconds for the external version. Surprisingly, the externally called solvers are much better here than the internal called solvers, though the internal solvers may benefit from learnt information. Overall, the externally called PrecoSAT solver needs 326.24 seconds over all benchmarks and Boolector needs 3,285.05 seconds.

To summarize, all the presented results can be achieved very easily by using *metaSMT*. Only one parameter needs to be switched to run a different solver.

B. Mutation Testing

Given a program, by mutation several faulty variants of the program are created and combined into a single program called meta-mutant. Using *metaSMT* the meta-mutant is encoded into a set of constraints. Each satisfying assignment yields a test case that discovers at least one fault. Experiments of [32] where executed on a set of *metaSMT* contexts. The results are shown in Table II. Comparing the Boolector backend with the MiniSAT backend using DirectSolver as well as GraphSolver middle-ends.

The results significantly vary with the difficulty of the instance. For easy instances the directly connected Boolector contexts and the graph-based MiniSAT perform better. However, for difficult instances with a run time over 1 minute, the graph-based Boolector context is fastest while MiniSAT-based contexts require significantly more time. For these harder instances the GraphSolver middle-end outperforms the direct variant of the same backend. This effect is most likely due to the removal of redundancies in the graph. For MiniSAT this amounts to run time reductions of around 50%. With *metaSMT* as abstraction layer it is easy to evaluate the effects of different contexts or optimizations. When changes are only in the abstraction layer no application code needs to be changed and only little effort is required.

TABLE II
RESULT FOR MUTATION TESTING USING DIFFERENT CONTEXTS

Name	Time [s]			
	Direct Boolector	Graph Boolector	Direct MiniSAT	Graph MiniSAT
isl	0.05	0.22	0.34	0.47
min	0.34	0.48	0.82	0.65
mid	4.73	5.07	7.87	4.36
fmin3	5.02	5.85	4.45	2.55
fmin5	21.08	25.96	14.92	9.56
fmin10	310.52	260.38	997.65	550.94
tri	207.69	193.99	1596.99	652.64

C. Comparison with direct API

For the factorization algorithm from Listing 2 a hand coded implementation using the Boolector C API is compared to a *metaSMT* DirectSolver-based implementation with the Boolector backend. The resulting application is available as part of the *metaSMT* package.

The experiment has the following setup: The algorithm from Listing 2 was changed to work on sequences instead of generating random numbers. A sequence of 10,000 random 64 bit numbers is generated and the algorithm it applied to it 10 times. The same sequence is used for both the hand coded and the *metaSMT*-based implementation of the algorithm. The complete experiment was repeated 5 times with Boolector being executed first and 5 times with *metaSMT* being executed first. Altogether each solver was forced to solve 1,000,000 constraints of 64 bit numbers factorized into two 32 bit numbers.

The results showed no significant difference caused by the *metaSMT* abstraction layer: $1,736s$ for plain Boolector compared to $1,729s$ for *metaSMT* with Boolector, i.e., 1.7 seconds for 10,000 factorizations.

D. Other applications

In addition to the aforementioned projects *metaSMT* is also used in a Constraint Random Stimuli generation Library. In the library elements of the SystemC Verification Library [33] and techniques from [34] are combined.

VII. Conclusions

metaSMT is a library that abstracts details of reasoning engines. Based on *metaSMT* very little programming effort is required to integrate formal methods into a user's application. Once this has been done, a wide range of solvers as well as optimization techniques can be selected.

Various research projects already integrate *metaSMT*. Future work on *metaSMT* includes the development of the following features: New frontend logics will complete the support for SMT logics (e.g. uninterpreted functions, integer arithmetic), while new middle-ends will increase solving performance (e.g. portfolio or multi-threaded contexts) and new backends will provide access to additional SMT solvers.

VIII. Acknowledgments

We would like to thank Hoang M. Le, Heinz Riener and Fereshta Yazdani for the helpful discussions, their proposals for improvements, testing and contributions to *metaSMT*.

References

[1] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. LNCS, vol. 1579. Springer Verlag, 1999, pp. 193–207.

[2] K. McMillan, "Interpolation and SAT-based model checking," in *Computer Aided Verification*, ser. LNCS. Springer Berlin / Heidelberg, 2003, vol. 2725, pp. 1–13.

[3] E. Arbel, O. Rokhlenko, and K. Yorav, "SAT-based synthesis of clock gating functions using 3-valued abstraction," in *Formal Methods in Computer-Aided Design, 2009*, 2009, pp. 198 –204.

[4] F. Haedicke, B. Alizadeh, G. Fey, M. Fujita, and R. Drechsler, "Polynomial datapath optimization using constraint solving and formal modelling," in *Computer-Aided Design (ICCAD), 2010 IEEE/ACM International Conference on*, 2010, pp. 756 –761.

[5] R. Drechsler, S. Eggersgluss, G. Fey, A. Glowatz, F. Hapke, J. Schloeffel, and D. Tille, "On acceleration of SAT-based ATPG for industrial designs," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 27, no. 7, pp. 1329 –1333, 2008.

[6] M. K. Ganai and A. Gupta, "Accelerating high-level bounded model checking," in *International Conference on Computer-aided design*. New York, NY, USA: ACM, 2006, pp. 794–801.

[7] P. Bjesse, "A practical approach to word level model checking of industrial netlists," in *International Conference on Computer Aided Verification*, 2008, pp. 446–458.

[8] A. Armando, J. Mantovani, and L. Platania, "Bounded model checking of software using SMT solvers instead of SAT solvers," *Int. J. Softw. Tools Technol. Transf.*, vol. 11, pp. 69–83, 2009.

[9] "SMT-COMP 2009," http://www.smtcomp.org/2009, 2009.

[10] "SMT-COMP 2010," http://www.smtcomp.org/2010, 2010.

[11] S. Ranise and C. Tinelli, "The Satisfiability Modulo Theories Library (SMT-LIB)," http://www.smtlib.org, 2006.

[12] O. Shtrichman, "Pruning techniques for the SAT-based bounded model checking problem," in *CHARME*, ser. LNCS, vol. 2144, 2001, pp. 58–70.

[13] S. Cook, "The complexity of theorem proving procedures," in *3. ACM Symposium on Theory of Computing*, 1971, pp. 151–158.

[14] C. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli, *Satisfiability Modulo Theories*, ser. Frontiers in Artificial Intelligence and Applications. IOS Press, February 2009, vol. 185, ch. 26, pp. 825–885.

[15] D. R. Cok, "jSMTLIB: Tutorial, validation and adapter tools for smt-libv2," in *NASA Formal Methods*, ser. LNCS, 2011, vol. 6617, pp. 480–486.

[16] "Boost C++ libraries," http://www.boost.org/.

[17] E. Niebler, "Proto: A compiler construction toolkit for DSELs," in *Proceedings of the 2007 Symposium on Library-Centric Software Design*, ser. LCSD '07. New York, NY, USA: ACM, 2007, pp. 42–51.

[18] J. de Guzman and D. Nuffer, "The Spirit library: Inline parsing in C++," *C/C++ User Journal*, vol. 21, no. 9, 2003.

[19] T. L. Veldhuizen, "Arrays in Blitz++," in *Proceedings of the Second International Symposium on Computing in Object-Oriented Parallel Environments*, 1998, pp. 223–230.

[20] F. Somenzi, *CUDD: CU Decision Diagram Package Release 2.4.1*. University of Colorado at Boulder, 2009.

[21] "Aiger," http://fmv.jku.at/aiger/.

[22] A. Biere, "Picosat essentials," *JSAT*, vol. 4, no. 2-4, pp. 75–97, 2008.

[23] N. Eén and N. Sörensson, "An extensible sat-solver," in *SAT*, ser. LNCS, E. Giunchiglia and A. Tacchella, Eds., vol. 2919. Springer, 2003, pp. 502–518.

[24] R. Brummayer and A. Biere, "Boolector: An efficient SMT solver for bit-vectors and arrays," in *Tools and Algorithms for the Construction and Analysis of Systems*, 2009, pp. 174–177.

[25] R. Wille, G. Fey, D. Große, S. Eggersglüß, and R. Drechsler, "Sword: A SAT like prover using word level information," in *VLSI of System-on-Chip*, 2007, pp. 88–93.

[26] L. M. de Moura and N. Bjørner, "Z3: An efficient smt solver," in *TACAS*, ser. LNCS, C. R. Ramakrishnan and J. Rehof, Eds., vol. 4963. Springer, 2008, pp. 337–340.

[27] M. Soeken, S. Frehse, R. Wille, and R. Drechsler, "RevKit: A toolkit for reversible circuit design," in *Workshop on Reversible Computation*, 2010, pp. 69 – 72.

[28] D. Große, R. Wille, G. Dueck, and R. Drechsler, "Exact multiple-control toffoli network synthesis with SAT techniques," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 28, no. 5, pp. 703 –715, May 2009.

[29] R. Wille, D. Große, M. Soeken, and R. Drechler, "Using higher levels of abstraction for solving optimization problems by boolean satisfiability," in *Symposium on VLSI, 2008. ISVLSI '08. IEEE Computer Society Annual*, 2008, pp. 411 –416.

[30] A. Biere, "Lingeling, plingeling, picosat and precosat at SAT race 2010," Tech. Rep., 2010. [Online]. Available: http://fmv.jku.at/papers/Biere-FMV-TR-10-1.pdf

[31] R. Wille, D. Große, L. Teuber, G. W. Dueck, and R. Drechsler, "RevLib: An online resource for reversible functions and reversible circuits," in *Int'l Symp. on Multi-Valued Logic*, 2008, pp. 220–225, RevLib is available at http://www.revlib.org.

[32] H. Riener, R. Bloem, and G. Fey, "Test case generation from mutants using model checking techniques," in *International Conference on Software Testing, Verification and Validation Workshops*, 2011, pp. 388 – 397.

[33] *SystemC Verification Standard Specification Version 1.0e*, SystemC Verification Working Group.

[34] R. Wille, D. Große, F. Haedicke, and R. Drechsler, "SMT-based stimuli generation in the SystemC verification library," in *Advances in Design Methods from Modeling Languages for Embedded Systems and SoC's: Selected Contributions on Specification, Design, and Verification from FDL 2009*, ser. Lecture Notes in Electrical Engineering, D. Borrione, Ed. Springer Netherlands, 2010, vol. 63, pp. 227–244.

A Study of Sweeping Algorithms in the Context of Model Checking

Zyad Hassan, Yan Zhang, and Fabio Somenzi
Dept. of Electrical, Computer, and Energy Engineering
University of Colorado at Boulder
Boulder, CO 80309

***Abstract*—Combinational simplification techniques have proved their usefulness in both industrial and academic model checkers. Several combinational simplification algorithms have been proposed in the past that vary in efficiency and effectiveness. In this paper, we report our experience with three algorithms that fall in the combinational equivalence checking (sweeping) category. We propose an improvement to one of these algorithms. We have conducted an empirical study to identify the strengths and weaknesses of each of the algorithms and how they can be synergistically combined, as well as to understand how they interact with `ic3` [1].**

I. Introduction

Combinational simplification eliminates redundancies and increases sharing of logic in a design. It has been successfully employed in logic synthesis, equivalence checking, and model checking.

In the model checking context, combinational simplification often dramatically improves the performance of the proof engines. This has made it into a primary component in both industrial [2] and academic [3], [4] model checkers. Several combinational simplification algorithms have been proposed in the past, such as DAG-aware circuit compression [5], [6] and sweeping methods [7]–[9]. Sweeping methods merge functionally equivalent nodes. They include BDD sweeping [7], SAT sweeping [8], [10], and cut sweeping [9].

When designing a model checker, the strengths and weaknesses of each of the sweeping methods should be taken into account. To the best of our knowledge, no studies have been carried out so far to evaluate and compare the different sweeping methods, with the exception of a limited study reported in [9].

The effect of combinational simplification on several model checking engines has been studied in the past. In [8], SAT sweeping has been shown to positively affect bounded model checking (BMC) [11]. In [12], it is shown that combinational redundancy removal techniques benefit interpolation considerably. The recently introduced `ic3` [1] incrementally discovers invariants that hold relative to stepwise reachability information. Designing a model checking flow that involves `ic3` requires understanding how combinational simplification algorithms affect it.

This paper makes the following contributions:

- We carry out a comparative study of the different sweeping methods.
- We propose a BDD-based cut sweeping method that is more effective than the original cut sweeping.
- We propose a combined sweeping approach in which more than one sweeping method is applied. We show that the combined approach can achieve more simplification than any of the methods can achieve individually.
- We perform an empirical study of the effect of sweeping on `ic3`.

The rest of the paper is organized as follows. Section II contains preliminaries. In Section III, we introduce the BDD-based cut sweeping method. In Section IV, we explain the rationale behind the combined sweeping approach. In Section V we present the experimental results and in Section VI we conclude.

II. Preliminaries

A. AND-inverter-graph

The input and output of our sweeping algorithms are AND-inverter-graphs (AIGs). An AIG is a directed acyclic graph that has four types of nodes: source nodes, sink nodes, internal nodes and the constant TRUE node. A *primary input* (PI) is a source node in an AIG. A *primary output* (PO) is a sink node and has exactly one predecessor. The internal nodes represent 2-input AND gates. A node v is a *fanin* of v_0 if there is an edge (v, v_0); it is a *fanout* of v_0 if there is an edge (v_0, v). $Left(v)$ and $Right(v)$ refer to the left and right predecessors of v.

$Fanin(v)$ and $Fanout(v)$ denote the fanins and fanouts of node v. An edge in an AIG may have the *INVERTED* attribute to model an inverter. The Boolean function of a PI is a unique Boolean variable. For an edge, it is the function of its source node if the edge does not have the INVERTED attribute, and the complement otherwise. For an internal node, it is the conjunction of the functions of its incoming edges. The Boolean function of a PO is that of its incoming edge.

A *path* from node a to b is a sequence of nodes $\langle v_0, v_1, v_2, \ldots, v_n \rangle$, such that $v_0 = a$, $v_n = b$ and $v_i \in Fanin(v_{i+1}), 0 \leq i < n$. The *height* of a node v, $h(v)$, is the length of the longest path from a PI to v. The *fanin (fanout) cone* of node v is the set of nodes that have a path to (from) v. Two nodes are *functionally equivalent (complementary)* if they represent the same (complementary) Boolean function(s). Functionally equivalent (complementary) nodes can be merged transferring the fanouts of the redundant nodes to their representative. To simplify our presentation, in what follows we deliberately ignore detection of complementary functions.

B. BDD Sweeping

BDD sweeping identifies two nodes as equivalent if they have the same BDD representation. The original algorithm of Kuehlmann and Krohm [7] works in two stages: equivalence checking and false negative detection. In the first stage, it builds BDDs for each node and merges nodes having the same BDD. The algorithm introduces an auxiliary BDD variable (cut point) for each node that has been merged, which potentially leads to false negatives. In the second stage, it takes the BDDs of the output functions and for each of them, replaces the auxiliary variables with their driving functions. The algorithm is complete in the sense that it can find all the equivalences in the circuit given a sufficiently large limit on the BDD sizes. However, a large limit often hurts efficiency. In this paper, we intend to use BDD sweeping in conjunction with SAT sweeping which is complete and avoids inherent BDD inefficiencies [8]. For that, we employ a version of BDD sweeping that is incomplete yet faster than the original.

The algorithm we adopt iterates the following steps on each node v of the AIG in some topological order. It builds a BDD for v and checks if there exists another node that has the same BDD. If so, it merges these two nodes and continues. Otherwise, if the BDD size exceeds a given threshold, the algorithm introduces an auxiliary BDD variables for v to be used in place of the original BDD when dealing with the fanouts of v. An important point is that the original BDD is kept for equivalence checking, but is not used to produce new BDDs. The algorithm is complete if the threshold is so large that no auxiliary variable is introduced. In practice, however, this can be prohibitive.

C. SAT Sweeping

The advancements in SAT solver technology over the past two decades has proliferated SAT-based reasoning methods. SAT sweeping is one such method proposed by Kuehlmann [8] for combinational redundancy identification. SAT sweeping queries the SAT solver to prove or refute the equivalence of two nodes in the circuit. The basic SAT sweeping algorithm works as follows. First, the circuit is simulated with random inputs, and candidate equivalence classes are formed, where nodes having the same simulation values are placed together. Next, for each two nodes belonging to the same class, the SAT solver is queried for their equivalence. If the SAT solver reports they are equivalent, one of them is merged into the other. Otherwise, the counterexample provided is simulated to break this equivalence class, and possibly rule out other candidate equivalences as well. This process is repeated until a resource limit is reached, or until all classes become singletons, indicating the absence of further equivalences.

In our implementation of SAT sweeping, several heuristics were applied. We mention each of them briefly.

1) Simulating Distance-1 Vectors: This heuristic was proposed in [13]. Instead of just simulating the counterexample to equivalence derived by the SAT solver, all distance-1 vectors, that have a single bit flipped, are simulated as well. Only the bits corresponding to the inputs that are in the fanin cone of the two nodes being checked for equivalence are flipped. We have found this heuristic to be very effective in practice. In [13], this process is repeated for vectors that were successful in breaking up equivalence classes until convergence. In our implementation, we only simulate the distance-1 vectors for the original counterexample: for the benchmark suite we experimented with, recurring on successful vectors is too expensive for the number of refinements it achieves.

2) Clustering: Simulating distance-1 vectors often results in considerable refinement of the equivalence classes. This is desirable, since an equivalence class is often broken up more cheaply by simulation than by the SAT solver. Moreover, we have observed that with distance-1 vector simulation, it becomes very likely that nodes remaining in an equivalence class are indeed equivalent. Therefore, rather than checking the equiva-

lence of two nodes at a time, we check the equivalence of all nodes in an equivalence class using a single SAT query. If they are all indeed equivalent, we find that using a single SAT query rather than $n-1$ queries where n is the number of nodes in the class.

3) Height-Based Merging: When two nodes are proved equivalent, we merge the node with a larger height into the one with a smaller height, instead of merging based on a topological order as in [13]. The intuition being that a node having a larger height often has a larger fanin cone, which suggests that merging it would lead to a larger reduction. Nodes coming later in a topological order do not necessarily have a larger height than nodes coming earlier.

D. Cut Sweeping

Cut sweeping [9] is a fast yet incomplete approach for combinational equivalence checking. It iteratively computes cuts for each AIG node and compares the functions associated to the cuts.

A *cut* is defined with respect to an AIG node, called *root*. A cut $C(v)$ of root v is a set of nodes, called *leaves*, such that any path from a PI to v intersects $C(v)$. A cut-set $\Phi(v)$ consists of several cuts of v. For cut-sets $\Phi(v_1)$ and $\Phi(v_2)$, the *merge operator* $\otimes$ is defined as

$$\Phi(v_1) \otimes \Phi(v_2) = \{C_1 \cup C_2 \mid C_1 \in \Phi(v_1), C_2 \in \Phi(v_2)\} \quad . \tag{1}$$

Assume $k \geq 1$. A *k-feasible* cut is a cut that contains at most k leaves. A k-feasible cut-set is a cut-set that contains only k-feasible cuts. The *k-merge* operator, $\otimes_k$, creates only k-feasible cuts. *Cut enumeration* recursively computes all k-feasible cuts for an AIG. It computes the k-feasible cut-set for a node v as follows:

$$\Phi(v) = \begin{cases} \{\{v\}\} & v \in PI \\ \{\{v\}\} \cup \Phi\left(Left(v)\right) \otimes_k \Phi\left(Right(v)\right) & v \in AND \end{cases} , \tag{2}$$

where *PI* and *AND* refer to the set of PIs and 2-input AND gates respectively. Note that cuts are not built for POs because they are never merged.

The function of a cut is the Boolean function of the root in terms of the leaves. It can be represented in different ways, for instance, using bit vectors or BDDs. Two cuts are equivalent if their cut functions are equal. Hence, two nodes are functionally equivalent if their cut-sets contain equivalent cuts.

Cut sweeping is parametrized by k and N, the maximum cut size and the maximum cut-set size, respectively. For each node v in some topological order of the AIG, the algorithm builds a k-feasible cut-set $\Phi(v)$. Each cut in $\Phi(v)$ is associated with a truth table. Next, it searches for a node equivalent to v by looking for a cut equivalent to some cut in $\Phi(v)$. If it succeeds, the two nodes are merged. Otherwise, a heuristic is applied to prune $\Phi(v)$ to at most N cuts. After pruning, the algorithm stores $\Phi(v)$ as the cut-set of v and builds a cross-reference between each of its cuts and v.

The heuristic for pruning, which we call the *quality heuristic*, computes a value q for each cut:

$$q(C) = \sum_{n \in C} \frac{1}{\mid Fanout(n) \mid} \quad . \tag{3}$$

The cuts with the smallest values of q are kept. The intuition of the quality heuristic is two-fold. First, it tries to localize the equivalence and thus favors smaller cuts. Second, it normalizes cuts by attempting to express them with the same set of variables. The chosen variables are those that have a large fanouts, i.e., that are shared by many other nodes.

A good truth-table implementation is critical to the performance of cut sweeping. In [9], truth tables are implemented as bit vectors. An advantage of bit vectors is the constant-time cost of Boolean operations. On the other hand, bit interleaving is required to extend the bit vectors to the same length so that the corresponding bits represent the same minterm[1].

III. BDD-based Cut Sweeping

Representing functions having a small number of inputs using bit vectors is very efficient. However, the number of bits required grows exponentially with the number of variables, which can easily lead to memory blow-up. As an alternative, BDDs, which are more suitable for large functions, can also be used to represent cut functions. Furthermore, the strong canonicity of BDDs makes it trivial to check for equivalence. The use of BDDs also enables a heuristic which we describe below.

The proposed algorithm differs from the original one in two aspects. First, we introduce a new parameter s, the maximum size of a BDD, to replace k. That is, instead of k-feasible cuts, we keep cuts whose functions contain at most s BDD nodes. Node count, as opposed to number of inputs, is a more natural characterization of BDD size.

The second difference comes from the pruning heuristic. We define the height h of a cut C as the average height of its nodes:

$$h(C) = \sum_{v \in C} \frac{h(v)}{|C|} \quad . \tag{4}$$

[1]A good reference of bit-interleaving can be found at http://graphics.stanford.edu/~seander/bithacks.html.

A smaller h indicates that the leaves in the cut are closer to the PIs. The *height heuristic* keeps at most N cuts choosing the ones with smallest values of h.

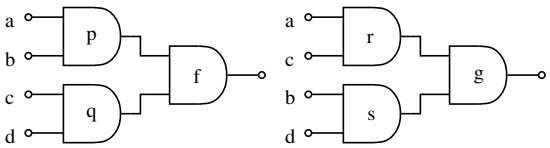


Fig. 1. Two implementations of a 4-input AND gate

A motivating example for the new heuristic is in Figure 1, which shows two different 4-input AND gates. Nodes a, b, c, and d are PIs. Nodes p, q, r, and s are internal nodes. Nodes f and g can only be merged if their cut-sets both contain $\{a, b, c, d\}$. However, if the internal nodes have many more fanouts than the PIs, the quality heuristic may select cuts containing the internal nodes instead, causing the merge to be missed.

As mentioned before, the quality heuristic tries to normalize the cuts on certain "attractors." This reduces the possibility that equivalent functions are represented differently. However, this might also lead to the loss of the opportunity to find equivalences that cannot be expressed by those "attractors," as in Figure 1.

On the other hand, the height heuristic tries to push the cut boundary as far as possible. A supporting argument is that, if a node is employed in equivalent cuts, then replacing it with its predecessors preserve equivalence. Furthermore, new merges that are otherwise undiscoverable (consider other equivalences that require a and b in the above example) may be found. The height heuristic does not attract cuts to certain nodes, which may result in different cuts for equivalent nodes. As shown in the experiments, the effectiveness of the height heuristic reduces as the height of nodes increases.

The two heuristics have their own strengths and weaknesses. A natural question is whether it is possible to combine them to benefit from their individual strengths. We can choose a few cuts with each heuristic. This may lead to more merges but may also worsen the efficiency if it significantly increases the number of cuts. To prevent such an increase, a combined heuristic only records height cuts for the lower nodes, while it keeps both types of cuts for the others.

There is some connection between cut sweeping with each of the two heuristics and BDD sweeping. With the height heuristic, cut sweeping tries to build cuts as large as possible, as BDD sweeping does. However, BDD sweeping can store cuts that exceed the threshold while cut sweeping only keeps those below the threshold. The quality heuristic tries to attract cuts on certain nodes, which is similar to the placement of auxiliary variables in BDD sweeping. Nevertheless, the number of "attractors" in the quality heuristic tends to be much larger than in BDD sweeping.

IV. Combining Sweeping Methods

The idea of combining several simplification algorithms is not new. Many existing model checkers iterate several simplification algorithms before the problem is passed to the model checking engines. However, we are unaware of any studies that have been carried out to identify the best way they could be combined. In this section we attempt to give general guidelines to which a combined approach should adhere. We support our claims by empirical evidence collected in the experiments reported in Section V.

The problem we address is as follows: given a time budget for combinational simplification, how should it be allotted to the different algorithms? The sweeping algorithms discussed in previous sections vary in their completeness and speed, with cut sweeping being the most incomplete method yet the fastest of the three methods, SAT sweeping being a complete, yet the slowest, and BDD sweeping lying in between, both in terms of completeness and speed.

Possible solutions include allocating the whole time budget to a single algorithm, or dividing it among two or more algorithms. The fact that some methods are better in approaching certain problems than others, suggests that more than one method should be applied. If two or more methods are to be applied in sequence, the intuition suggests that the lower effort methods should precede the higher effort ones. The advantages of doing so are two-fold. First, although the higher-effort methods are likely to discover the merges that a lower-effort method discovers, in general, it will take them more time to do so. Second, preceding higher-effort methods by lower-effort methods is beneficial in having them present a smaller problem that is easier to handle.

Finally, the percentages of total time that should be allotted to each of the methods to yield the maximum possible reduction is studied in Section V.

V. Results

The experiments are divided into three parts. The first part compares different variations of the cut sweeping

algorithm. The second part shows the results of the combined sweeping methods, and the third part is concerned with the effect of sweeping on `ic3`[2].

We use the benchmark set from HWMCC'10 [14], a set of 818 benchmarks in total. The experiments are run on a machine with a 2.8GHz Quad-Core CPU and 9GB of memory. We use CUDD [15] as our BDD package for all the BDD-related algorithms.

A. BDD-based Cut Sweeping

Variations of cut sweeping are applied to the HWMCC'10 benchmark set. The differences between variations are two-fold. First, either the number of variables, k, or the number of BDD nodes, s, is used to drop over-sized cuts. Second, we experiment with several heuristics for pruning cut-sets: the quality heuristic, the height heuristic, and two combined heuristics. The naive combined heuristic ("combined-1") chooses one cut based on the height heuristic and the others based on the quality heuristic. The other heuristic ("combined-2") sets a threshold on the node height (350 in our experiments). For nodes that are below the threshold, it only keeps a height cut. For higher nodes, it produces cut-sets consisting of a height cut and two quality cuts. We denote a method by a k or an s followed by the the heuristic name. All the variations use BDDs to represent the cut functions.

The results are shown in Table I; they are aggregated over the 818 benchmarks. Based on experiments, both the threshold of BDD sweeping and s in BDD-based cut sweeping are set to 250. The total number of AIG nodes before sweeping is 7.22M. "Final" is the size of AIGs after sweeping. "Generated" and "Kept" are the number of cuts generated and kept by the corresponding methods. For an individual benchmark, its "height" is the average height of all merged cuts. The "Height" column is computed by taking the average of the "height" of all the benchmarks. A smaller value indicates that more merges are found by cuts that are close to the PIs. Note that since we use BDDs, the results in terms of efficiency of bit-vectors based methods may not be as good as in [9]. Therefore, when dealing with them, we just compare the effectiveness.

Results indicate that the resulting AIGs are consistently smaller with s than with k. There are a few interesting observations. First, the ratios $GeneratedCuts/Merge$ and $KeptCuts/Merge$ are improved significantly with s. This means that with s, each cut has a larger chance of resulting in a merge.

Second, while "k-quality" and "k-combined-1" have very close sweeping times, the latter achieves 19.8% more merges. Furthermore, the decrease in the "Height" column reveals that the height cuts indeed lead to merges. Although "s-quality" is more effective than the two above methods, it is less efficient due to the larger cut sizes.

For the methods with s (excluding "s-quality"), we observe that "s-height $N = 1$" is the fastest and produces a good number of merges. Increasing the number of height cuts to two triples the run time without gaining many more merges. Comparing it with "s-combined-1", an improvement on the merges is shown by the latter. This indicates that maintaining one height cut and one quality cut works better than two height cuts. For "s-combined-2", the number of merges is between the two above methods, but with lower run time. Furthermore, the numbers of generated cuts and kept cuts are even comparable to "s-height $N = 1$". That is, even though we keep three cuts for those nodes with height larger than 350, on average we compute only a few percent more cuts than we do in the case of one cut per node.

The "Height" values of the three methods confirm the assumption made in Section III: most merges produced by the height heuristic come from cuts close to the PIs. When the two heuristics are combined, a significant increase on the "Height" value is observed. In Figure 2, we show the number of merges found by "s-height $N = 1$" and "s-combined-2" on nodes within different height ranges. The plot is normalized to "k-quality" and has bin size of 50, i.e., a point at $(2, 1)$ indicates that the method finds the same number of merges as "k-quality" for nodes with height from 100 to 149. Obviously, the height heuristic works better on smaller height nodes, while the quality heuristic catches more on larger height ones. The combined heuristic takes the advantage of both and produces an even better profile on nodes with larger heights. Note that although the height heuristic works worse on the nodes with larger heights, it can still get more merges. This may be due to the fact that in this benchmark set, a large percent of equivalences are located at lower heights.

In our setup, cut sweeping is intended for usage as a fast method. Thus we consider "s-height $N = 1$" and "s-combined-2" to be the best variants. Compared to BDD sweeping, those two variants are faster because they create fewer BDD nodes than BDD sweeping, but are less effective since BDD sweeping may keep BDDs

[2]Detailed results for first and second parts can be found at http://eces.colorado.edu/~hassanz/sweeping-study

TABLE I
RESULTS OF CUT SWEEPING, BDD SWEEPING AND SAT SWEEPING ON THE HWMCC'10 SET. BY DEFAULT, $k = 8$ AND $s = 250$.

Method	Final ($\times 10^6$)	Merges ($\times 10^5$)	Time (s)	Generated ($\times 10^7$)	Kept ($\times 10^7$)	Height
k-quality $N = 5$	6.82	2.62	123.26	5.83	1.92	10.20
k-combined-1 $N = 5$	6.75	3.14	129.64	5.84	1.90	8.57
s-quality $N = 5$	6.71	3.31	536.75	7.63	2.32	12.11
s-height $N = 1$	6.55	4.07	58.99	1.07	0.54	3.19
s-height $N = 2$	6.51	4.20	181.52	2.18	0.99	2.94
s-combined-1 $N = 2$	6.48	4.42	181.21	2.29	1.02	12.86
s-combined-2	6.52	4.28	74.64	1.10	0.54	12.72
BDD Sweeping	6.34	5.61	112.74	–	–	–
SAT Sweeping	6.10	6.37	2149.4	–	–	–

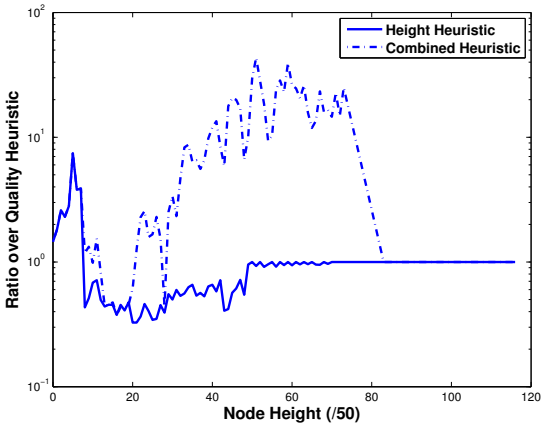


Fig. 2. Number of merges on nodes within different height ranges

that exceed the threshold.

B. Combined Sweeping Methods

In this section, we show experimental evidence that supports the guidelines of combining sweeping methods presented in Section IV. In particular, we try several possibilities of allotting the budget to the different sweeping algorithms with the purpose of identifying empirically if they should be combined, and if so, in which way. In what follows we use the "s-height $N = 1$" variant of cut-sweeping since it is the fastest, and we simply refer to it as cut sweeping.

In our combined approach, SAT sweeping is always run last since it is the only complete method of the three, and should thus be given whatever time is left to find equivalences not discovered by the other methods. Also, the time not used by any of the preceding methods is passed to SAT sweeping. For instance, if cut sweeping is given a time budget of 4 seconds and only uses 3 of them, SAT sweeping gets to run for one extra second.

We compare the reduction measured in terms of the number of AIG nodes removed, and the total time spent in sweeping. Data are aggregated over the 818 benchmarks. The base case for our comparisons is the pure SAT sweeping case in which SAT sweeping gets the whole budget. The time budget used in our study is 10 seconds.

We consider the following policies: (a) allocating the budget to two methods, (b) allocating it to three methods, and (c) allocating the whole budget to SAT sweeping. For (a) and (b), we consider all the different permutations of assigning integer time values to each method, such that they sum up to 10 seconds. Note that if a sweeping algorithm times out, what it has achieved thus far is used in what follows. In all cases, a set of light-weight sequential simplification algorithms are applied before sweeping. This set of algorithms includes cone-of-influence reduction, stuck-at-constant latch detection, and equivalent latch detection. The total number of AIG nodes for all 818 benchmarks measured after the sequential simplification step is 6.1M.

Results are presented in Table II. The first column lists the methods, where the number before each sweeping method indicates the number of seconds given to it. The second column shows the number of AIG nodes removed. The third column shows the total time spent in sweeping. The methods are listed in order of decreasing reduction. The last row is for pure SAT sweeping. We only show the best three setups in terms of reduction for each of the possible orders of the method sequences.

Several observations can be made. First, when it comes to running two methods in sequence, BDD sweeping combined with SAT sweeping outperforms cut sweeping combined with SAT sweeping. The method that achieves maximum reduction (8 seconds of BDD sweeping followed by 2 seconds of SAT sweeping) removes 56K more nodes than pure SAT sweeping (7.7% more reduction). Second, more reduction is achievable by running three methods in sequence. As suggested in Section IV, ordering the methods by increasing effort

TABLE II
EFFECT OF BUDGET ALLOCATION ON REDUCTION.

Method	Reduction	Total Sweeping Time (s)
4 Cut, 5 BDD, 1 SAT	801,932	518
2 Cut, 5 BDD, 3 SAT	801,137	516
6 Cut, 3 BDD, 1 SAT	801,119	522
4 BDD, 1 Cut, 5 SAT	794,052	517
8 BDD, 1 Cut, 1 SAT	793,921	515
7 BDD, 2 Cut, 1 SAT	793,814	519
8 BDD, 2 SAT	793,226	500
7 BDD, 3 SAT	793,068	503
5 BDD, 5 SAT	792,797	508
1 Cut, 9 SAT	772,563	512
6 Cut, 4 SAT	771,070	513
3 Cut, 7 SAT	769,483	511
10 SAT	736,594	619

TABLE III
EFFECT OF SWEEPING ON ic3'S PERFORMANCE.

Seed Index	Solves (No Sweeping)	Solves (With Sweeping)	Runtime (s) (No Sweeping)	Runtime (s) (With Sweeping)
1	693	698	96,297	91,762
2	689	699	95,629	90,341
3	691	699	95,050	92,714
4	696	697	93,691	91,141
5	693	698	95,007	89,656
6	690	695	96,270	91,559
7	693	699	94,784	92,056
8	690	701	94,351	90,837
9	693	693	95,491	92,847
10	690	693	95,124	93,048
Average	691.8	697.2	95,169	91,596

(or equivalently by increasing degree of completeness) achieves more reduction than otherwise. Here, the best method (4 seconds, 5 seconds, and 1 second for cut, BDD and SAT sweeping, respectively), has an edge of 65K nodes over pure SAT sweeping (about 8.9% more reduction). Third, in terms of sweeping time, it is clear that a large drop occurs ($>$ 100 seconds) when two or three methods are combined versus pure SAT sweeping, which is due to the often smaller time needed by BDD and cut sweeping to discover equivalences than SAT sweeping. Given an overall model checking budget, smaller sweeping time allows more time for the model checking engine, which is desirable.

The question of whether such difference has a considerable effect on the performance of the model checking engine is answered in the next section.

C. *Effect on* ic3

The recently developed model checking algorithm, ic3 [1], has been regarded as the best standalone model checking algorithm developed up till now [16]. As the interaction of combinational simplification methods with different model checking algorithms has been studied in the past, we here aim to study how they interact with ic3. In particular, we would like to empirically find out if ic3 benefits from preprocessing the netlist with a simplification algorithm or not, and if it does, how sensitive it is to the magnitude of reductions achieved through simplification.

In the first experiment, we compare two runs of ic3, one that is preceded by SAT sweeping, and one that is not. The experimental setup is as follows. A total timeout of 10 minutes is used. The budget for SAT sweeping is 10 seconds. The light-weight sequential simplification algorithms referred to in Section V-B are applied once in the no-sweeping case, and twice (before and after sweeping) in the sweeping case. We compare the number of solves, and the aggregate runtime among all benchmarks.

It is important to note that the ic3 algorithm has a random flavor. In particular, the order by which generalization (dropping literals) is attempted is randomized. Also, since the algorithm is SAT-based, randomization occurs in the SAT solver decisions. To have reliable experimental results, each experiment is repeated with 10 different seeds, and the results are averaged over the different seeds.

Results are shown in Table III. The first column shows the seed index, the second and third columns show the number of solves without and with sweeping, and the fourth and fifth columns show the aggregate runtime without and with sweeping.

The results confirm a positive effect of sweeping on the performance of ic3. On average, five more solves are achieved with sweeping, and the aggregate runtime drops by 3.8%.

The enhancement in the performance of ic3 in presence of sweeping can be attributed to two factors. First, reduction in the number of gates caused by sweeping can result in the reduction in the SAT solver time. Second, simplification often results in dropping of latches (e.g., if it merges the next-state functions of two latches). For example, for the benchmark set used in our experiments, sweeping reduces the aggregate number of latches from 279,161 to 269,091 (3.6% decrease). This reduces the amount of work done by ic3 in generalization of counterexamples to induction.

We now repeat the previous experiment, but this time we compare the number of solves and the aggregate runtime between pure SAT sweeping and the empirically optimum combined sweeping scheme of Section V-B. The purpose of this experiment is to understand how

TABLE IV
OPTIMUM SWEEPING VERSUS PURE SAT SWEEPING.

Seed Index	Solves (Pure SAT Sweeping)	Solves (Optimum Sweeping Scheme)	Runtime (s) (Pure SAT Sweeping)	Runtime (s) (Optimum Sweeping Scheme)
1	698	696	91,762	91,567
2	699	696	90,341	91,113
3	699	697	92,714	92,373
4	697	702	91,141	90,478
5	698	697	89,656	90,327
6	695	699	91,559	90,606
7	699	697	92,056	91,498
8	701	699	90,837	92,228
9	693	696	92,847	91,252
10	693	697	93,048	92,663
Average	697.2	697.6	91,596	91,411

sensitive `ic3` is to the magnitude of reductions.

Results are shown in Table IV, where the second and third columns compare the number of solves for pure SAT sweeping and the optimum sweeping scheme, and the fourth and fifth columns compare the total runtime.

As the results indicate, `ic3` does not benefit much from the better reduction achieved by the combined sweeping scheme. The lack of performance enhancement on `ic3` can be attributed to the small improvement in reduction the combined sweeping approach achieves over pure SAT sweeping. In particular, while pure SAT sweeping removes 737K nodes out of the total 6.1M nodes in the 818 benchmarks (12.1% reduction), the combined approach removes 802K nodes (13.2% reduction); a mere 1.1% improvement. This suggests that `ic3` has a small sensitivity to the magnitude of reduction.

VI. CONCLUSION

In this paper, we presented an empirical study of the different sweeping methods proposed in the past. We have shown that a combined sweeping approach outperforms any of the sweeping methods alone. We have also proposed a BDD-based cut sweeping method that is more effective than the original cut sweeping. Finally, we have studied the effect of sweeping on the new model checking algorithm, `ic3`, and investigated the causes of the better performance it experiences with sweeping. The goal of this analysis is to help designers of model checkers to make decisions regarding the incorporation of sweeping methods and to provide a deeper understanding of how sweeping methods interact with `ic3`.

ACKNOWLEDGEMENTS

We thank Aaron Bradley who motivated this work and contributed to many discussions. We also thank the reviewers for their insightful comments regarding cut sweeping's pruning heuristics that prompted us to try the "combined-2" heuristic.

REFERENCES

[1] A. R. Bradley, "SAT-based model checking without unrolling," in *Proc. Int. Conf. on Verification, model checking, and abstract interpretation*, 2011, pp. 70–87.

[2] H. Mony, J. Baumgartner, V. Paruthi, R. Kanzelman, and A. Kuehlmann, "Scalable automated verification via expert-system guided transformations," in *Formal Methods in Computer-Aided Design*, 2004, pp. 159–173.

[3] R. Brayton and A. Mishchenko, "ABC: An academic industrial-strength verification tool," in *Computer Aided Verification*, 2010, pp. 24–40.

[4] G. Cabodi, S. Nocco, and S. Quer, "Benchmarking a model checker for algorithmic improvements and tuning for performance," in *Proc. Hardware Verification Workshop*, 2010.

[5] P. Bjesse and A. Boralv, "DAG-aware circuit compression for formal verification," in *Proc. Int. Conf. on Computer-aided design*, 2004, pp. 42–49.

[6] A. Mishchenko, S. Chatterjee, and R. Brayton, "DAG-aware AIG rewriting a fresh look at combinational logic synthesis," in *Proc. Design Automation Conference*, 2006, pp. 532–535.

[7] A. Kuehlmann and F. Krohm, "Equivalence checking using cuts and heaps," in *Proc. Design Automation Conference*, 1997, pp. 263–268.

[8] A. Kuehlmann, "Dynamic transition relation simplification for bounded property checking," in *Proc. Int. Conf. on Computer-aided design*, 2004, pp. 50–57.

[9] N. Eén, "Cut sweeping," Cadence Design Systems, Tech. Rep., 2007.

[10] A. Mishchenko, S. Chatterjee, and R. Brayton, "FRAIGs: A unifying representation for logic synthesis and verification," EECS Dept., UC Berkeley, ERL, Tech. Rep., Mar 2005.

[11] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *TACAS*, 1999, pp. 193–207.

[12] G. Cabodi, M. Murciano, S. Nocco, and S. Quer, "Stepping forward with interpolants in unbounded model checking," in *Proc. Int. Conf. on Computer-aided design*, 2006, pp. 772–778.

[13] A. Mishchenko, S. Chatterjee, R. Brayton, and N. Een, "Improvements to combinational equivalence checking," in *Proc. Int. Conference on Computer-aided design*, 2006, pp. 836–843.

[14] A. Biere and K. Claessen, "Hardware model checking competition," in *Hardware Verification Workshop*, 2010.

[15] F. Somenzi, *CUDD: CU Decision Diagram Package*, University of Colorado at Boulder, ftp://vlsi.colorado.edu/pub/.

[16] R. Brayton, N. Een, and A. Mishchenko, "Continued relevance of bit-level verification research," in *Proc. Usable Verification*, Nov. 2010, pp. 15–16.

Enhancing ABC for LTL Stabilization Verification of SystemVerilog/VHDL Models

Jiang Long, Sayak Ray, Baruch Sterin, Alan Mishchenko, Robert Brayton
Berkeley Verification and Synthesis Research Center (BVSRC)
Department of EECS, University of California, Berkeley
{jlong, sayak, sterin, alanmi, brayton}@eecs.berkeley.edu

***Abstract*—We describe a tool which combines a commercial front-end with a version of the model checker, ABC, enhanced to handle a subset of LTL properties. Our tool, VeriABC, provides a solution at the RTL level and produces models for synthesis and formal verification purposes. We use Verific (a commercial software) as the generic parser platform for SystemVerilog and VHDL designs. VeriABC traverses the Verific netlist database structure and produces a formal model in the AIGER format. LTL can be specified using SVA 2009 constructs that are processed by Verific. VeriABC traverses the resulting SVA parse trees and produces equivalent LTL formulae using the F,G, Until and X operators. The model checker in ABC has been enhanced to handles LTL stabilization properties, an important subset of LTL. The paper presents VeriABC's implementation strategy, software architecture, tool flow, environment setup for formal verification, use model, the specification of properties in SVA and translation into LTL. Finally the properties are translated into safety properties that can be verified by the ABC model checker.**

I. Introduction

We present an integrated tool flow for liveness model checking using industry hardware description languages (HDLs) and SystemVerilog Assertions: (i) VeriABC: a front-end to read in hardware models expressed in HDLs, and (ii) capability of model checking a subset of liveness properties. VeriABC is able to read in hardware models expressed in SystemVerilog or VHDL. SystemVerilog and VHDL languages are the most popular HDLs being used in industry today for digital designs. VeriABC generates a formal model in the AIGER[2] format and relies on a commerical front-end, Verific, to build a generic parser platform for HDLs. This allows down-stream tool flows in synthesis and verification. A version of ABC was enhanced from a safety-only verification engine to allow both safety and liveness verification. Our current version supports a particular subset of liveness properties called *stabilization properties* or *generalized fairness constraints* (defined in Section IV).

In a typical use model, a user will develop a hardware design in SystemVerilog or VHDL, and specify its correctness requirements in the property specification language *SystemVerilog Assertion* (SVA). SVA has been adopted into IEEE SystemVerilog standard and is supported by major commercial tools in simulation, synthesis and verification. The SVA language in SystemVerilog 2009 standard contains liveness constructs that allow full specification of liveness properties as those defined in LTL formulas. In our framework, a user can specify both safety properties and liveness properties (stabilization properties, to be precise). In this paper, we detail its liveness capabilities.

After reading in a design, VeriABC bit-blasts it into a bit-level netlist and converts the SVA stabilization properties into an intermediate LTL representation. Then the LTL properties are folded into the bit-level netlist in an appropriate way (using an extended *liveness-to-safety* conversion, explained later in Section IV). The resulting bit-level netlist represents a formal model of the design, represented as an and-inverter graph (AIG). And-inverter graphs are concise representations of finite state machines. The AIGER[2] format is a prevalent format for AIG representation. supported by various academic tools and used in annual hardware model checking competitions. Since our liveness verification is based on liveness-to-safety conversion, eventually the safety verification backend in ABC[1] is called which works on the bit-level netlist produced by the VeriABC front-end. We have used this methodology to verify liveness in the context of compositional verification of deadlock freedom of micro-architectural communication fabrics. Preliminary experimental results are encouraging.

A. Related work

Although parsing and elaborating RTL languages are a standard practice for commercial EDA products, it is a daunting task for academics due to language complexity and continuous language evolution over the years. Although, *vl2mv*[12] was an academic tool that attempted to treat a significant subset of the Verilog language for synthesis and verification purposes, it was not maintained and language support was not complete. Tools like ABC[1], VIS[9] parse subsets of Verilog language too strict and not applicable in a broad setting. Freely accessible tools like icarus[3] contain Verilog languages front-end but are not up-to-date with newer SystemVerilog features.

Our choice of a commercial and stable front-end Verific, allows academics to get around the language barrier to access real-world designs.

Liveness-to-safety conversion was first proposed in [7], [24]. They demonstrated that verification problems for any ω-regular property can be converted into a verification problem of an equisatisfiable safety problem. Their algorithm has been deployed successfully in industrial setups and used to verify liveness properties of microprocessor designs [6]. Our liveness-to-safety conversion algorithm for stabilization is essentially an extension of the algorithm proposed in [24] and is broader than discussed in [6].

B. Contribution

As illustrated in Figure 1, we combine a commercial front-end, Verific, with a version of our model checker, ABC, enhanced to handle a subset of LTL properties. Our tool processes the Verific output, conducts various modeling procedures on the design, compiles SVA into LTL formulas, then the enhanced ABC processes the LTL formulas for liveness model checking. We detail the software architecture, tool flow, formal model construction, SVA compilation and downstream LTL modeling checking.

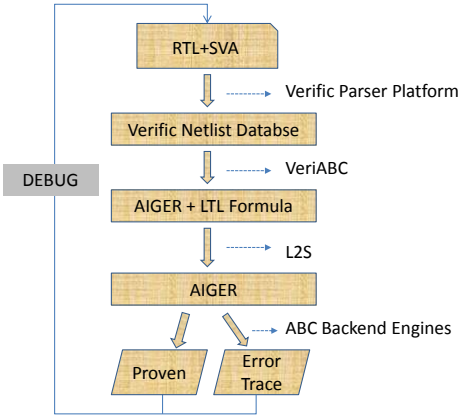


Fig. 1. Complete Tool Flow

C. Organization of the Paper

We first discuss the capabilities of the Verific parser platform. In Section III we describe the architecture, formal modeling of VeriABC and translation of SVA into LTL. In Section IV the stabilization properties are described in further detail. Section V describes the liveness-to-safety conversion for stabilization properties. Experimental results are presented in Section VI.

II. Background: Verific Parser Platform

Verific Design Automation[4] builds SystemVerilog and VHDL Parser Platforms which enable its customers to develop advanced EDA products quickly and at low cost. Verific's Parser Platforms are distributed as C++ source code or library and build on all 32 and 64 bit Unix, Linux, and Windows operating systems. Applications vary from formal verification to synthesis, simulation, emulation, virtual prototyping, in circuit debug, and design-for-test. We chose Verific as our front-end for its commercial success and stability in supporting the latest language constructs in SystemVerilog.

Figure 2 shows the architecture of the Verific parser front-end. The main commands we use in Verific library are *analyze* and *elaborate*. *Analyze* creates parse-trees and performs type-inferencing to resolve the meaning of identifiers. The Parser/Analyzer supports the entire SystemVerilog IEEE 1800,

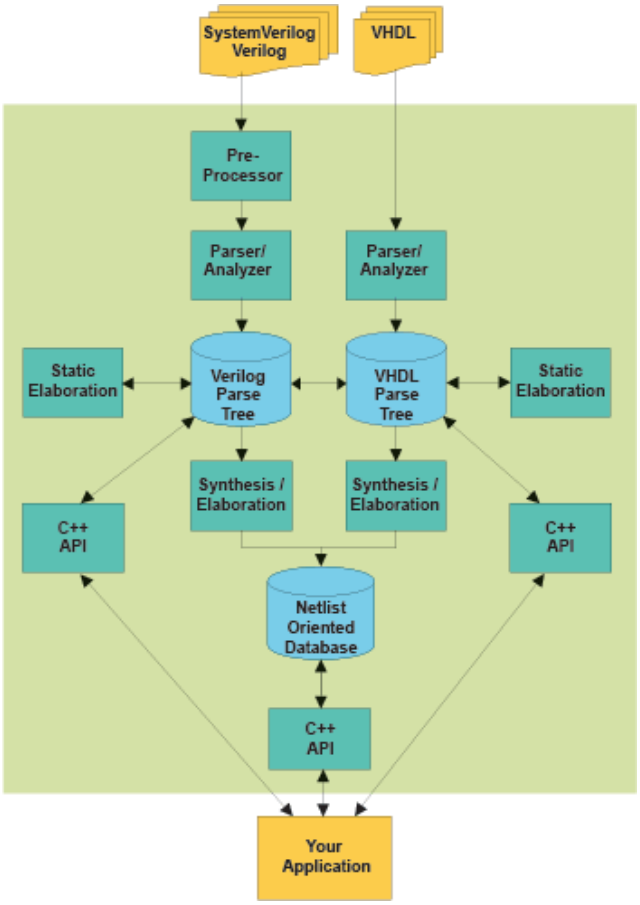


Fig. 2. Verific Parser Flow

VHDL IEEE 1076-1993, and Verilog IEEE 1364-1995/2001 languages, without any restrictions. The resulting parse tree comes with an extensive API.

Elaborate supports both static elaboration and RTL elaboration. Static elaboration elaborates the entire language, and specifically binds instances to modules, resolves library references, propagates defparams, unrolls generate statements, and checks all hierarchical names and function/task calls. The result after static elaboration is an elaborated parse tree, appropriate for simulation like applications. RTL elaboration is limited to the synthesizable subset of the language. In addition to the static elaboration tasks for this subset, it generates sequential networks through flipflop and latch detection, and Boolean extraction. The result after RTL elaboration is a netlist database, appropriate to applications such as logic synthesis and formal verification. This netlist database is the starting point of VeriABC and we utilize Verific provided C++ APIs to access the database.

A. Verific Netlist Database Structure

In this Section, we use Verilog terminology to present Verific's netlist database structures. The netlist database is rather intuitive and adheres to the module definitions. Shown in Table I, there is a one-to-one correspondence between the C++ API class definitions and Verilog constructs.

A *Netlist* corresponds to *module* definitions in Verilog while an *Instance* object corresponds to module instantiation,

Verific Database C++ API Class	Verilog RTL Objects
Netlist	*Module definition*
Instance	*Module instantiation*
Port	*Module port declarations*
Net	*wire/reg/assign*
PortRef	*Port to Net connectivity*

TABLE I
VERIFIC NETLIST OBJECTS

after the module's parameters have been characterized. An $Instance$ is a thin copy of the $Netlist$ plus a pointer to its parent netlist. A $Netlist$ contains a set of $Ports$, $Nets$ and $Instances$ for its internal logic structure. A $Port$ corresponds to the Verilog port definitions which can be $input$, $output$ or $inout$. A Net is a named component, intuitively a $wire$. $PortRef$ contains the connectivity between a $Port$ and a Net. The direction of the $PortRef$ can be in, out, or $inout$ depending on the type of $Port$ it contains. Using these C++ objects, the Verific netlist database defines a directed hyper-graph and encapsulates the following types of information:

Design Hierarchy
: Design hierarchy is captured as an instance tree by the parent pointers in the $Instance$ with a top-level netlist as the root.

Unique Hierarchical Name
: Following the design hierarchy through the instance tree, each internal object is assigned a unique hierarchical name.

Connectivity
: A directed hyper-graph is defined through $Port$, Net and $PortRef$: $Port$ being the node, Net being the edge, and $PortRef$ containing the connectivity and direction information between pairs of a $Port$ and a Net. As an edge in the hyper-graph, a Net can be referenced in multiple $PortRef$ objects.

Logic Definition
: At the leaf of the design hierarchy, a $Netlist$ of primitive operator types such as and, or, $adder$, flipflop, latches etc defines the basic logic operators.

Recursively, the functional behavior of the design is captured through the directed hierarchical hyper-graph with basic logic operators at its leaf level.

III. VERIABC

VeriABC traverses the above netlist database and transforms it into a monolithic AIG which can be treated as a directed acyclic graph (DAG). The AIG contains primary input, primary output, register nodes and and-inverter nodes. Each named $Port$ and Net in the Verific netlist has a mapped node in the AIG graph. Additional book-keeping information such as hash tables are created that map the hierarchical name to the corresponding AIG node. The down-stream model checker ABC then reads in this AIG to conduct formal analysis.

A. Architecture

A hyper-graph is rather hard to traverse and conduct analysis/transformation at the same time.

As shown in Figure 3, we employ a two phase approach. First we construct an intermediate netlist graph, a DAG with extra annotated node types representing logic structure and connectivity of the flattened design. In addition to the simple node types in and-inverter graphs, extra node types contain annotations for language constructs such as tri states, flipflops and latches etc. For example, flipflops contain set/reset pins and driving d-pins; latches contain additional gated-clock definitions. By language definition, a design can specify any signal for the clock and reset signals. Design behavior is defined by event-driven semantics. Further analysis needs be done to determine if there is an AIG representation that can capture the original design semantics. The intermediate netlist is a DAG for which various traversal algorithms can be conducted for the later analysis. For this step, VeriABC only traverses the design hierarchy and hyper-graph in the Verific netlist database to gather information and construct the intermediate DAG representation without conducting any design style checking or transformation.

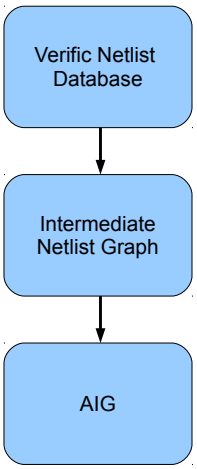


Fig. 3. VeriABC Architecture

B. Formal Modeling

The end result of VeriABC is an AIG model that is consistent with the original RTL design semantics. AIG is recognized as a finite state machine model. Compared to the event-driven semantics in HDL language, the execution semantics of an AIG is synchronous with an implicit universal clock that ticks at every step of the execution. The register loads in its driver value at the beginning of each step. The semantics inferred from the Verific netlist structure is more complex, such as its flip-flop can have arbitrary reset logic and clock network. The task in formal modeling is to transform the above design components into AIG registers with additional glue logic so that it maintains the consistency. In its simplest form, the following check determines if the design can be readily represented by an AIG.

- All registers are clocked by the same primary input signal which does not fanout to other nodes.
- Reset/Set signals are primary inputs

- No combinational loops
- No multiple drivers per node

More complex design modeling and transformations can be achieved by identifying certain patterns by traversing the intermediate netlist graph. Our current implementation supports the above form and produces a design style summary for debugging purposes. Though capacity and performance depends on the type of individual transformation and analysis, for the above ones, the transformation and design style checking is very fast, as it conducts only a few traversals of the intermediate netlist graph. After design style checking, a final traversal of the intermediate netlist graph generates an AIGER file representing the formal model.

C. Commands Implemented

VeriABC implementation utilizes the Tcl command interface shipped with the Verific library distribution. The following is the list of commands implemented to manage the environment setup for formal verification.

veriabc_analyze
: This command constructs the intermediate netlist graph in Figure 3 and conducts design style checking.

veriabc_set_reset
: Although a *reset* signal can be automatically detected in certain situations, this command provides the user with the option to specify the length of the reset sequence and phase of the *reset* condition. A user can also specify the initial value of the registers through a VCD waveform or textual file. In the generated formal model, the initial value of the registers will be valued at the end of the reset sequence and the reset condition will be disabled after reset.

veriabc_set_clock
: A user can specify the clock periods and relationships in the situation when multi-clocks are in the design. A phase-counter network will be created in the formal model to generate the corresponding clock signals.

veriabc_set_cutpoint
: This command will prune the cone-of-influence at cutpoint signal and treat it as free input.

veriabc_set_constant
: This command sets either an input or a cut-point signal to a constant value.

veriabc_set_assume
: This constrains the design signal to be a constant.

veriabc_write
: write out the final formal model in AIGER format.

The above commands give the user flexibility to model the environment with constraints and conduct design abstractions during verification.

D. SVA to LTL Compilation

In SystemVerilg 2009 standard, a rich set of LTL operators are added into SVA language. The SVA properties shown in Figure 4 are the templates for matching the basic LTL operators used in the set of stabilization properties. For stabilization properties, in our current implementation, we restrict the p and q to be Boolean expressions which seems to be sufficient in practice. The SVA verification directive $assume$ is used to specify a fairness constraint, while $assert$ is used to specify the target LTL properties.

```
property Until(p,q);
  p s_until q;
endproperty

property GF(p);
  always (s_eventually p);
endproperty

property FG(p);
  s_eventually (always p);
endproperty

property X(p);
  s_nexttime (p);
endproperty
```

Fig. 4. SVA Liveness Template

Verific also processes SVA constructs into the netlist database structure, essentially a corresponding parse tree is integrated into the netlist. We conduct traversals of the parse tree, identify specific liveness constructs and map them into the corresponding LTL formula. At the end of the procedure, along with the AIGER file generated, a separate file containing the LTL formulas are generated indicating target liveness assertions and fairness constraints. The support signals referred to in the LTL formulas are named output signals in the AIGER file. Although we currently only support stabilization properties in LTL, the full LTL language using X, F, G, U operators can be specified fully and translated through SVA constructs. In doing so, this completes the formal model generation and SVA compilation at the RTL level.

IV. Liveness Model Checking in VeriABC

In the FSM modeling formalism, the most intuitive notion of stabilization states that the system will always reach a particular state and stay there forever, no matter which state the system started from or which path it took. Relaxing this notion a bit, stabilization means that the system will eventually reach and stay within a given subset of states. Also, stabilization may denote conditions on the input and output signals of a system when it attains a stable state. Applications of stabilization properties have been demonstrated in [14] and [18], to name a few. We review some basic temporal logic terminology and formally define stabilization properties using LTL below.

A. LTL, model checking and stabilization property

Familiarity is assumed with LTL, basic model checking algorithms, and related terminology like Kripke structures and

Büchi automata. For further details, see [13]. In our current context, we use LTL properties $\mathbf{GF}p$ and $\mathbf{FG}p$, and thus overview their semantics here: let π be a path in some Kripke structure K; $\pi \models_K \mathbf{G}p$ means property p will hold on every state along π; $\pi \models_K \mathbf{F}p$ means the property p will hold eventually on some state along π; $\pi \models_K \mathbf{GF}p$ means p will hold along π infinitely often, and $\pi \models_K \mathbf{FG}p$ means p will hold eventually on π forever. Since temporal operators $\mathbf{F}$ and $\mathbf{G}$ are dual (i.e. $\mathbf{F}p \equiv \neg\mathbf{G}\neg p$), operators $\mathbf{FG}$ and $\mathbf{GF}$ are also dual (i.e. $\mathbf{FG}p \equiv \neg\mathbf{GF}\neg p$).

Definition 1 (GF-atom): Any LTL formula of the form $\mathbf{GF}p$ or $\mathbf{FG}p$, where p is some atomic proposition or some Boolean formula involving atomic propositions only, will be called a 'GF-atom'.

Stabilization properties are defined as the family of LTL formulas that are Boolean combinations of GF-atoms. Formally:

Definition 2 (Stabilization Property): The set of stabilization properties is the syntactic subset of LTL defined as follows:

- any GF-atom is a stabilization property
- if ϕ is a stabilization property, then so is $\neg\phi$
- if ϕ and ψ are stabilization properties, then so are $\phi \wedge \psi$ and $\phi \vee \psi$

Example 1: $\mathbf{FG}p$, $\mathbf{GF}p \Rightarrow \mathbf{GF}q$, $\mathbf{FG}p \wedge \mathbf{FG}q \Rightarrow \mathbf{FG}r$, and $\mathbf{FG}p \Rightarrow \mathbf{FG}q \vee (\mathbf{FG}r \wedge \mathbf{GF}s)$ are stabilization properties where p, q, r, and s are atomic propositions or Boolean formulas involving atomic propositions only and $a \Rightarrow b$ is the usual shorthand for $\neg a \vee b$. However, $\mathbf{G}(r \Rightarrow \mathbf{F}g)$ is an LTL liveness property but not a stabilization property.

Needless to say, these are all liveness properties. But not all of them specify so-called system stabilization directly. Properties like $\mathbf{FG}p$ and $\mathbf{FG}p \wedge \mathbf{FG}q \Rightarrow \mathbf{FG}r$ (or its generalization $\wedge_{i=1}^{k}\mathbf{FG}p_i \Rightarrow \mathbf{FG}q$) are perhaps the most elementary stabilization properties. $\mathbf{FG}p$ means that the system eventually will reach a state from where p will always hold, i.e. the system will eventually 'stabilize' at p. $\mathbf{FG}p \wedge \mathbf{FG}q \Rightarrow \mathbf{FG}r$ means that if the system stabilizes at p and also at q (at perhaps some other time), then it will stabilize eventually at r. Hence, semantics of these properties are close to the intuitive notion of stabilization. [14] demonstrates the use and significance of stabilization properties in the context of biological system analysis. However, our definition of stabilization captures a broader family of specifications. It includes $\mathbf{FG}p \Rightarrow \mathbf{FG}q \vee (\mathbf{FG}r \wedge \mathbf{GF}s)$ which may look contrived, but for example, [18] uses many such complicated stabilization properties for compositional deadlock analysis of micro-architectural communication fabrics. On the other hand, our definition includes many properties not intended to specify so-called stabilization behavior. For example, $\mathbf{GF}p$ or $\mathbf{GF}p \Rightarrow \mathbf{GF}q$.

The main motivation behind considering this broader subset of LTL is that we offer a short-cut L2S conversion, avoiding Büchi automaton construction, in a uniform way (due to the duality between $\mathbf{FG}$ and $\mathbf{GF}$ operators). The most significant applications of this class that we have encountered is "stabilization verification", and hence the name is coined for the family. (This name was inspired by [14]). Thus the L2S conversion proposed here may be applied for proving properties beyond the context of stabilization verification (eg. $\mathbf{GF}p \Rightarrow \mathbf{GF}q$).

The class of LTL properties defined as stabilization properties in this paper is a very important class of temporal properties extensively studied in the literature. It is related to so-called *fairness* specifications. Operators $\mathbf{GF}$ and $\mathbf{FG}$ are often called *infinitary operators* [19] and symbols F^{∞} and G^{∞} are used respectively instead [15]. The class itself (i.e. Boolean combination of GF-atoms) has been called *general fairness constraints* [16], [19]. As shown in [16], various notions of fairness like *impartiality* [21], *weak fairness* [20](also called *justice* [21]), *strong fairness* [20] (also called *compassion* [21]), *generalized fairness* [17], *state fairness* [22] (also known as *fair choice from states* [23]), *limited looping fairness* [5], and *fair reachability of predicate* [23] can be expressed by stabilization properties. These properties are used to exclude "unfair" counterexamples in liveness verification in both linear time and (fair) branching time paradigms. For liveness verification, we usually have a liveness property (the actual proof obligation) along with a set of fairness constraints. Liveness properties may not be stabilization properties. In that case we may need to construct the product of the system and the Büchi automaton of the (negation of the) liveness property before performing the L2S conversion. Interestingly, for many interesting applications as in [14] and [18], the liveness verification obligations fall entirely in the family of stabilization properties. For these applications, the simple L2S scheme proposed in this paper works. Note that some liveness properties like $\mathbf{G}(request \Rightarrow \mathbf{F}grant)$ are not stabilization properties, but also have a direct L2S conversion [24]. It is, therefore, an interesting question that under what more general conditions there exists a direct L2S conversion.

V. L2S Conversion for stabilization properties

It is important to understand that any counterexample to a liveness property (which must be an infinite trace) can be seen as a "lasso" like configuration with a finite handle and a finite loop. Therefore a liveness counter-example is a lasso which does not satisfy the property on the loop but satisfies all imposed fairness constraints on the loop.

In general, a liveness problem is converted to a safety problem by adding a loop-detection logic and property-detection logic on top of the product of the FSM of the original system and the Büchi automaton of the property to be verified. The loop-detection logic consists of a set of shadow registers, comparator logic, and an 'oracle'. The oracle saves the system state in the shadow registers at a non-deterministically chosen time. In all subsequent time frames, the current state of the system is compared to the state in the shadow registers. Whenever these two states match, the system has completed a loop. The non-deterministic nature of the oracle allows all such loops to be explored. The property verification logic checks if any of the liveness conditions are violated in any such loop and all fairness conditions always hold in the loop. This check is done as a safety obligation. For a more detailed exposition, see [24].

As mentioned, for some simple properties L2S conversion can be achieved while avoiding explicit Büchi automata con-

struction. This is done by adding more functionality to the property detection logic. As presented in [24], these properties are $\mathbf{F}p, \mathbf{GF}p, \mathbf{FG}p, p\mathbf{U}q, , \mathbf{G}(r \Rightarrow \mathbf{F}q)$, and $\mathbf{F}(p \wedge \mathbf{X}q)$ (Table 1 of [24]). This approach, reviewed in Figure 5, depicts an L2S converted circuit for verifying the LTL property $\mathbf{F}p$.

In the next paragraph, we describe how this construction verifies $\mathbf{F}p$. In Section V-A we explain how to extend the ideas of Figure 5 for stabilization properties. Instead of presenting the liveness-to-safety conversion through Kripke structure-based representations (i.e. through explicit state machines based representations), we present the idea in terms of an actual circuit construction (i.e. through symbolic representation of the state space). Also, although we do not discuss it further, the same mechanism handles fairness constraints, which are always stabilization properties, so they just entail adding additional logic to the circuit for the monitor. For Kripke structure-based descriptions of liveness-to-safety conversion, see [24].

In Figure 5, `save` represents an additional primary input added to the circuit. This plays the role of the 'oracle'. When `save` is asserted for the first time, the current state of the circuit is saved in the set of shadow registers, and register `saved` is set. `saved` thus remembers that input `save` has been asserted and allows any further activity on `save` to be ignored. For subsequent time frames, `saved` enables the equality detection between the current state of the circuit and the state in the shadow registers. Clearly, signal `looped` is asserted iff the system has completed a loop. Signal `live` remembers if the signal p has ever been asserted. The safety property that the circuit verifies is, therefore, `looped` $\Rightarrow$ `live`. (In general this would be `looped` & `fair` $\Rightarrow$ `live`.) The block marked with "$\Uparrow$" represents this logical implication - the direction of the arrow distinguishes the antecedent signal from the consequent signal of the implication.

A. L2S *for stabilization properties*

In [24], the authors show how to do the L2S conversion for $\mathbf{GF}p$ and $\mathbf{FG}p$, which are GF-atoms. We demonstrate how to extend this to any Boolean combination of GF-atoms using an example, omitting a formal proof of correctness.

Consider a simple stabilization property ϕ of the form $\mathbf{FG}a \Rightarrow \mathbf{FG}b + \mathbf{FG}c$. An L2S converted circuit for this is shown in Figure 6. (For simplicity, we do not show any fairness constraints in the example.) Note that, signal `live` in Figure 5 monitors if signal p has ever been asserted from the very initial time frame. But for verifying $\mathbf{GF}p$, we need to monitor whether signal p has been asserted between the time when `saved` is set and the time when `looped` becomes true. Using this fact, the duality between $\mathbf{FG}$ and $\mathbf{GF}$ operators, and the Boolean structure $X_a \Rightarrow X_b + X_c$ of the given formula, we can derive the circuit of Figure 6. Logic that captures the Boolean structure of ϕ is marked with a dotted triangle in Figure 6. Hence, for any arbitrary stabilization property, we need to create monitors for individual GF-atoms and a crown of combinational logic on top of these monitors that captures the Boolean structure of the property. We can formulate the following theorem.

Theorem 1: For any stabilization property, the given procedure finds one counter-example if one exists.

(Proof Sketch) Any stabilization property can be transformed into another stabilization property with $\mathbf{GF}$ operators only. Let f be the Boolean structure in the negation of the given stabilization property. The procedure described above will create a monitor that will search for a lasso-loop where f is violated inside the loop. Since the procedure implicitly enumerates all possible cycles in the state space, it will detect a violating cycle if one such exists.

VI. Experimental Results

We implemented our L2S scheme for general stabilization properties in ABC and experimented with several designs of communication fabrics from industry. Our objective was to verify all stabilization properties defined for every structural primitive of the xMAS framework [18]. The properties, though local to each component, are verified in the context of the whole design in order to avoid explicit environmental modeling. BLIF models of the communication fabrics were generated by the xMAS compiler [11] from high-level C++ models. The L2S monitor logic was then created by ABC on these BLIF models. The xMAS compiler also generates SMV models from C++ models so that the LTL encoding of the stabilization properties can be verified directly on the SMV models using the NuSMV model checker.

We found that the ABC based L2S implementation has much better scalability than NuSMV. NuSMV can solve only toy designs while on the large designs of interest, it fails to produce a result. On the other hand, our tool works well even on large designs. For most cases, it produces a result almost immediately. For a few cases, initial trials could not produce a proof, but with the latest version of ABC using simplification, abstraction, speculative reduction, and property directed reachability (PDR) analysis [8], the proofs were completed. This observation supports the premise that the use of highly developed safety techniques can pay off for liveness verification.

Experimental results are shown below. Among all the local properties that the xMAS compiler generated, we provide results for the most challenging one. Call this property ψ; it is defined for a FIFO buffer, and has the following LTL form

$$\psi := \mathbf{FG}(\neg a) \Rightarrow \mathbf{FG}(\neg b) \vee \mathbf{FG}(c)$$

where a, b, and c are appropriate design signals (i.e. interface signals of a FIFO buffer). Table 1, 2, and 3 compare the performance of ABC with NuSMV on small examples. These examples are instances of communication fabrics or sub-modules thereof, and are explained in full detail in [10]. *simple_credit* and *simple_vc* (Table 1 and 2, respectively) are designs corresponding to Figure 4 and 5 of [10], and *simple_ms* (Table 3) is a much simpler version of the design shown in Figure 6 of [10]. Note from the table, how performance of NuSMV degrades even for small designs. For large designs, NuSMV could not finish for any single instance of ψ.

Since ψ is defined for a FIFO buffer and the xMAS compiler created one instance of ψ for each FIFO buffer, the

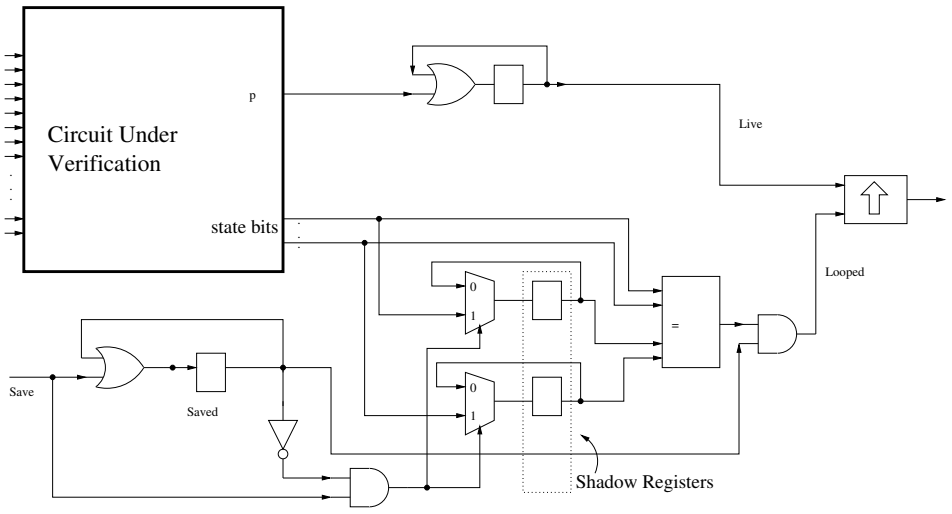


Fig. 5. Liveness-to-safety transformation for $\mathbf{F}p$

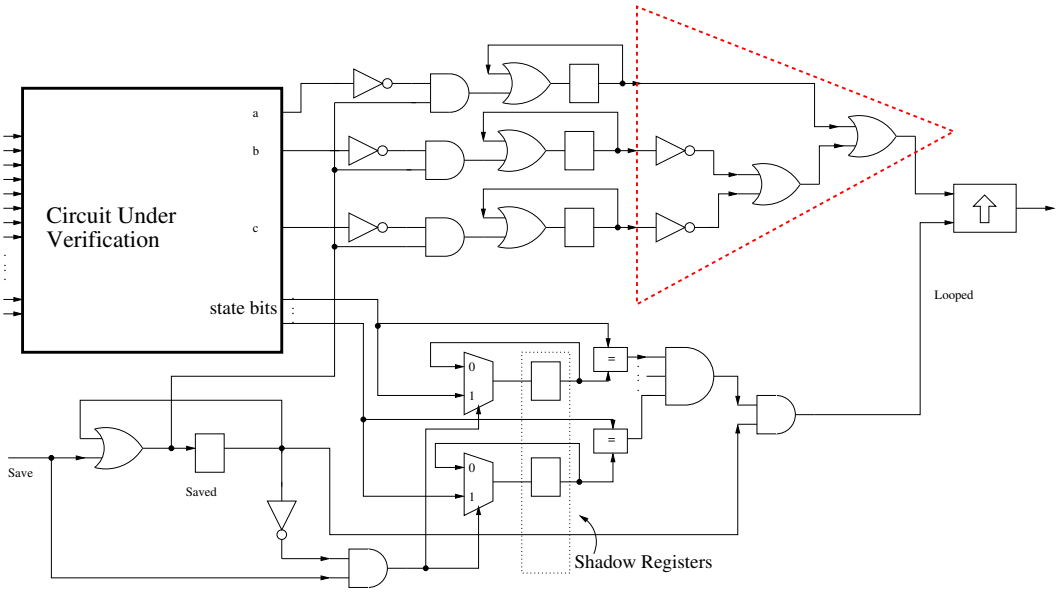


Fig. 6. L2S for stabilization property $FGa \Rightarrow FGb + FGc$

Prop #	ABC (sec)	NuSMV (sec)
0	0.25	0.115
1	0.05	0.14
2	0.02	0.09

TABLE II
SIMPLE_CREDIT

Prop #	ABC (sec)	NuSMV (sec)
0	0.09	33.23
1	0.07	31.8
2	0.06	39.57
3	0.03	16.46
4	0.5	41.37
5	0.03	16.89

TABLE III
SIMPLE_VC

Prop #	ABC (sec)	NuSMV (sec)
0	0.03	431.5
1	0.12	379.59
2	0.8	471.36
3	0.8	385.67

TABLE IV
SIMPLE_MS

number of ψ instances is the same as the number of FIFO buffers. For example, the designs corresponding to Table 1, 2, and 3 above have 3, 6, and 4 FIFO buffers, respectively.

We also experimented on two large communication fabrics of practical interest [10], [18]. One has 20 buffers and the other has 24 buffers. 19 out of 20 of the first design and 23 out of 24 from the second design were proved by ABC by a light-weight interpolation engine within a worst case time of 5.83 seconds (most were proved in less than a second). Light-weight interpolation could not prove one instance from each design. These were proved using advanced techniques from ABC's arsenal of safety verification algorithms. For example, ABC took a total of 217.2 seconds to prove one of these harder properties. In this time span, ABC first did some preliminary simplification, then it tried interpolation, BMC, simulation and PDR in parallel for a time budget of 20 seconds. But this attempt failed and it moved on to further simplification by reducing the design using localization abstraction and speculation. It ran interpolation, BMC, simulation, BDD-based reachability and PDR engines in parallel both after abstraction and speculation, using an elevated time budget of 100 seconds and 49 seconds respectively. The iteration after abstraction could not prove the property, but the iteration after speculation managed to prove it with the PDR engine, which produced the final proof in 7 seconds.

VII. Conclusion & Future Work

We have developed a tool, VeriABC, which allows us to access real industrial designs written in SystemVerilog or VHDL and to process them into the AIGER format. The result can be used for synthesis and verification using a tool like ABC. We described how the RTL processing is done using the commercial front-end, Verific. SVA assertions are also processed by Verific, and VeriABC creates a separate file of equivalent LTL formulas. We showed an application of this to property checking, where ABC was enhanced to convert a subset of LTL into a circuit structure, thus effectively allowing liveness checking in ABC.

The use of a stable, supported and complete language processing tool like Verific, allows academics access to real industrial designs, without going through the hassle and daunting task of building their own equivalent tool. Liveness property checking is a growing interest in industry, and our enhanced ABC with a front end that automatically converts to a circuit structure for liveness checking, can use the advanced safety property methods of ABC.

In the future, the development of VeriABC will allow us to extract higher level constructs from SystemVerilog and VHDL by accessing Verific's parse trees. These constructs can be passed on using an extended AIGER format to an enhanced ABC, which will use this information in synthesis and verification.

VIII. Acknowledgment

This work has been supported in part by SRC contract 2057.001 and our industrial sponsors: Abound Logic, Actel, Altera, Atrenta, IBM, Intel, Jasper, Magma, Oasys, Real Intent, Synopsys, Tabula, and Verific.

References

[1] ABC - a system for sequential synthesis and verification. Berkeley Verification and Synthesis Research Center, http://www.bvsrc.org.
[2] Aiger, http://fmv.jku.at/aiger/.
[3] Icarus verilog, http://iverilog.icarus.com.
[4] Verific Design Automation: http://www.verific.com.
[5] K. Abrahamson. Decidability and expressiveness of logics of processes. In *PhD Thesis, University of Washington*, 1980.
[6] J. Baumgartner and H. Mony. Scalable liveness checking via property-preserving transformations. In *DATE*, pages 1680–1685, 2009.
[7] A. Biere, C. Artho, and V. Schuppan. Liveness checking as safety checking. In *In FMICS02: Formal Methods for Industrial Critical Systems, volume 66(2) of ENTCS*. Elsevier, 2002.
[8] A. Bradley. Sat-based model checking without unrolling. 2011.
[9] R. Brayton, G. D. Hachtel, A. Sangiovanni-vincentelli, F. Somenzi, A. Aziz, S. tsung Cheng, and S. Edwards. Vis : A system for verification and synthesis. pages 428–432. Springer-Verlag, 1996.
[10] S. Chatterjee and M. Kishinevsky. Automatic generation of inductive invariants from high-level microarchitectural models of communication fabrics. In *CAV*, pages 321–338, 2010.
[11] S. Chatterjee, M. Kishinevsky, and U. Ogras. Modeling communication micro-architectures (with one hand tied behind your back). *Intel Technical Report*, 2009.
[12] S.-T. Cheng and R. K. Brayton. Compiling verilog into automata, 1994.
[13] E. Clarke, O. Grumburg, and D. Peled. Model checking. 2000.
[14] B. Cook, J. Fisher, E. Krepska, and N. Piterman. Proving stabilization of biological systems. 2011.
[15] E. A. Emerson. Handbook of theoretical computer science (vol. b). chapter Temporal and modal logic, pages 995–1072. MIT Press, Cambridge, MA, USA, 1990.
[16] E. A. Emerson and C.-L. Lei. Modalities for model checking: branching time logic strikes back. *Sci. Comput. Program.*, 8:275–306, June 1987.
[17] N. Francez and D. Kozen. Generalized fair termination. In *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '84, pages 46–53, New York, NY, USA, 1984. ACM.
[18] A. Gotmanov, S. Chatterjee, and M. Kishinevsky. Verifying deadlock-freedom of communication fabrics. 2011.
[19] R. Hojati, R. K. Brayton, and R. P. Kurshan. Bdd-based debugging of design using language containment and fair ctl. In *Proceedings of the 5th International Conference on Computer Aided Verification*, CAV '93, pages 41–58, London, UK, 1993. Springer-Verlag.
[20] L. Lamport. "sometime" is sometimes "not never": on the temporal logic of programs. In *Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '80, pages 174–185, New York, NY, USA, 1980. ACM.
[21] D. J. Lehmann, A. Pnueli, and J. Stavi. Impartiality, justice and fairness: The ethics of concurrent termination. In *Proceedings of the 8th Colloquium on Automata, Languages and Programming*, pages 264–277, London, UK, 1981. Springer-Verlag.
[22] A. Pnueli. On the extremely fair treatment of probabilistic algorithms. In *Proceedings of the fifteenth annual ACM symposium on Theory of computing*, STOC '83, pages 278–290, New York, NY, USA, 1983. ACM.
[23] J. P. Queille and J. Sifakis. Fairness and related properties in transition systems a temporal logic to deal with fairness. In *Acta Informat*, pages 195–220, 1983.
[24] V. Schuppan and A. Biere. Efficient reduction of finite state model checking to reachability analysis. *Int. J. Softw. Tools Technol. Transf.*, 5(2):185–204, 2004.

On Incremental Satisfiability and Bounded Model Checking

Siert Wieringa*
Aalto University, Finland
siert.wieringa@aalto.fi

Abstract

Bounded Model Checking (BMC) is a symbolic model checking technique in which the existence of a counterexample of a bounded length is represented by the satisfiability of a propositional logic formula. Although solving a single instance of the satisfiability problem (SAT) is sufficient to decide on the existence of a counterexample for any arbitrary bound typically one starts from bound zero and solves the sequence of formulas for all consecutive bounds until a satisfiable formula is found. This is especially efficient in the presence of incremental SAT-solvers, which solve sequences of incrementally encoded formulas. In this article we analyze empirical results that demonstrate the difference in run time behavior between incremental and non-incremental SAT-solvers. We show a relation between the observed run time behavior and the way in which the activity of variables inside the solver propagates across bounds. This observation has not been previously presented and is particularly useful for designing solving strategies for parallelized model checkers.

*Financially supported by the Academy of Finland project 139402

1 Introduction

Model checking is a formal verification technique revolving around proving temporal properties of systems modelled as finite state machines. A property holds for the model if it holds in all possible execution paths. If the property does not hold this can be witnessed by a *counterexample*, which is a valid execution path for the model in which the property does not hold. Because the model has a finite number of states any infinite execution of the system includes a loop, and can thus be represented by a finite sequence of execution steps. Bounded Model Checking (BMC) [1] is a symbolic model checking technique in which the existence of a counterexample consisting of a bounded number of execution steps is represented by the satisfiability of a propositional logic formula. It thus allows the use of decision procedures for the propositional satisfiability problem (SAT) for model checking. Despite the theoretical hardness of SAT [7] such decision procedures, called SAT-solvers [9, 14, 17], have become extremely efficient. BMC is popular as a technique for refuting properties, and although BMC based techniques can be used for proving properties we do not consider such techniques here.

A typical BMC encoding will have semantics such that if there exists a counterexample of

length k then there also exists a counterexample of any length greater than k. Thus in principle solving a single propositional logic formula is sufficient to decide on the existence of a counterexample for any arbitrary finite bound. However, one typically starts to solve the formula corresponding to bound zero and then solves sequentially each consecutive bound until a counterexample is found. We will refer to this as the standard sequential search strategy. This strategy has the nice property that it always finds a counterexample of minimal length. As with every bound the representing formula grows larger it also avoids solving unnecessarily large formulas. Importantly, the performance of this strategy benefits greatly from the availability of *incremental* SAT-solvers. Incremental SAT-solvers can solve sequences of formulas that share large parts in common efficiently in a single solver process, allowing reuse of information between formulas.

2 Motivation

Automated SAT based planning is a problem closely related to BMC. It deals with the same sequences of parameterized formulas, except that the satisfiability of a formula now corresponds to the existence of a plan of a bounded length. In [15] evaluation strategies for planning were suggested that are more opportunistic than the standard sequential search strategy. They suggest to spend some amount of the total solving effort at attempting to solve formulas for bounds ahead of the currently smallest unsolved one. It was inspired by the empirical observation that if a plan exists then amongst the smallest satisfiable formulas in the sequence there are typically formulas that are much easier to solve than the largest unsatisfiable ones. A more opportunistic search strategy may reduce the total time required to find a satisfiable formula by skipping over hard instances. Such strategies are natural for environments in which multiple computing nodes are available in parallel, where one may define some nodes to use a more opportunistic strategy than others.

The observation on the empirical hardness of the smallest satisfiable formulas compared to the largest unsatisfiable ones can also be made for BMC. We attempted to implement opportunistic strategies in our parallelized BMC framework Tarmo [18]. This however turned out to be less efficient then we would have expected, with performance degrading for many benchmarks. In this article we evaluate the performance of the incremental solver and compare it against that of solving each bound separately. The purpose of this study is not to illustrate that incremental solvers are more effective for BMC than non-incremental ones, as that is well known, but to understand when and how opportunistic strategies can be applied. This is done by comparing against non-incremental solver run times because solvers applying opportunistic strategies benefit less from the incremental interface of the solver, as the problem is no longer introduced one bound at a time.

3 Preliminaries

The majority of modern SAT-solvers are based on the *Davis Putnam Logemann Loveland* (DPLL) procedure [8]. The DPLL-procedure is a backtracking search procedure for SAT that builds a partial assignment by iteratively deciding on a *branching variable* to be assigned a value in the partial assignment. When the par-

tition of the search space defined by the partial assignment is without solutions the algorithm backtracks. In addition to this modern SAT-solvers typically employ conflict clause learning [17]. Such solvers record a new *conflict clause* whenever they are forced to backtrack. They then backtrack non-chronologically to a decision point at which the conflict clause was still satisfied.

The performance of the DPLL-procedure depends heavily on its branching variable decisions. A commonly used decision heuristic for clause learning SAT-solvers is the *Variable State Independent Decaying Sum (VSIDS)* heuristic first presented in the solver Chaff [14]. The idea of the heuristic is to favor variables that are included in recently derived conflict clauses. For each variable an initially zero value called the *activity* is maintained. Whenever a conflict clause is learnt the activity of all variables that occur in the clause is increased. Periodically the activity of all variables is divided by a constant.

All results presented in this article were obtained using the SAT-solver MiniSAT 2.2.0 [9][1]. The solver core was not modified but a number of small modifications[2] were made in auxiliary routines such as the file parser in order to read incremental SAT sequences from disk. When employing BMC typically the SAT-solver will not read the formula sequence from disk but it will rather be integrated into a BMC engine that is generating formulas for new bounds on the fly. We use sequences stored on disk as this is convenient for testing the performance of the SAT-solver independently. As our sequences are streamable one can also use them as an interface between a BMC engine and a SAT-solver without the need for integrating them into one application. The input sequences used were the same as the benchmarks used for experimental results presented in [18]. These sequences were generated with the current state-of-the-art encoding for model checking of linear time temporal logic properties with past (PLTL) [2] as implemented in the model checker NuSMV 2.4.3 [6].

[1] Available from `http://www.minisat.se`

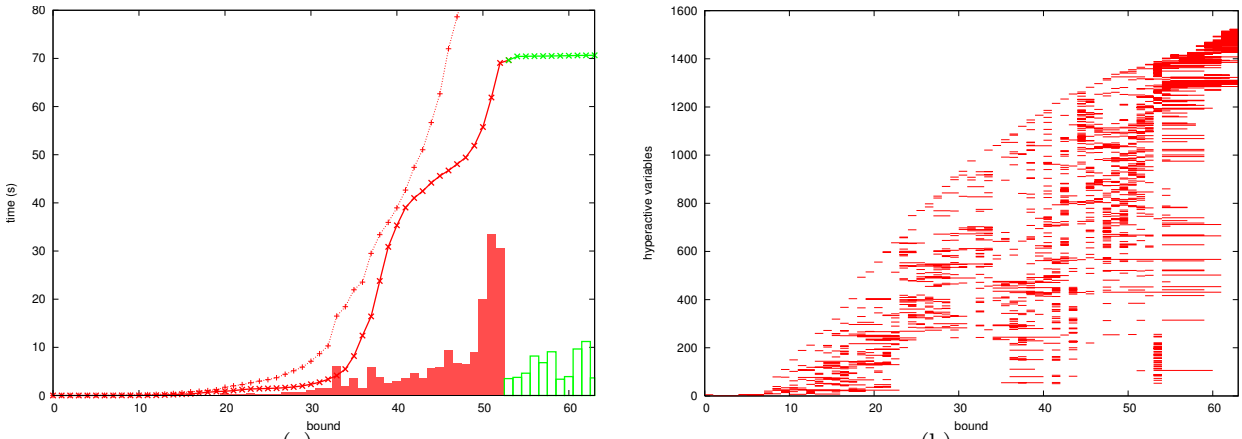
[2] Available from `http://users.ics.tkk.fi/swiering`

4 Run time

In our experiments we have studied the behavior of SAT-solvers on problem sequences from BMC regarding both the run time and variable activity. A selection of the results is presented in the figures in this article. For each benchmark there are two subfigures, a run time graph labeled (a) and a variable activity graph labeled (b). In this section we will focus only on the run time graphs, which have bounds on the x-axis and time on the y-axis. For each bound a vertical bar displays the time it took to solve the formula corresponding to that single bound using the SAT-solver in non-incremental fashion. If the solver found unsatisfiable a solid red bar is drawn, if the solver found satisfiable only the outline of the bar is drawn in green.

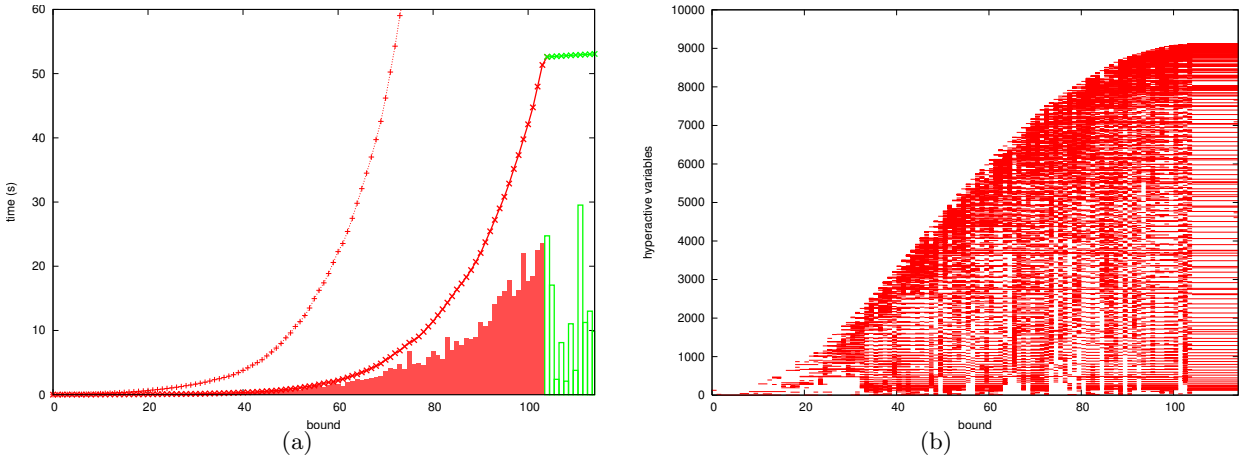
The incremental solver solves using the standard sequential strategy and whenever it completes a bound it reports the run time up to that point. The points in the graphs marked with crosses (x) and connected by the thick line represent these run times.

The thin dotted line connecting the plusses (+) is representing the cumulative run time of the non-incremental solver, i.e. for each bound k the value displayed is the sum of all run times of the non-incremental solver tests from bound 0 to k. This demonstrates the time required



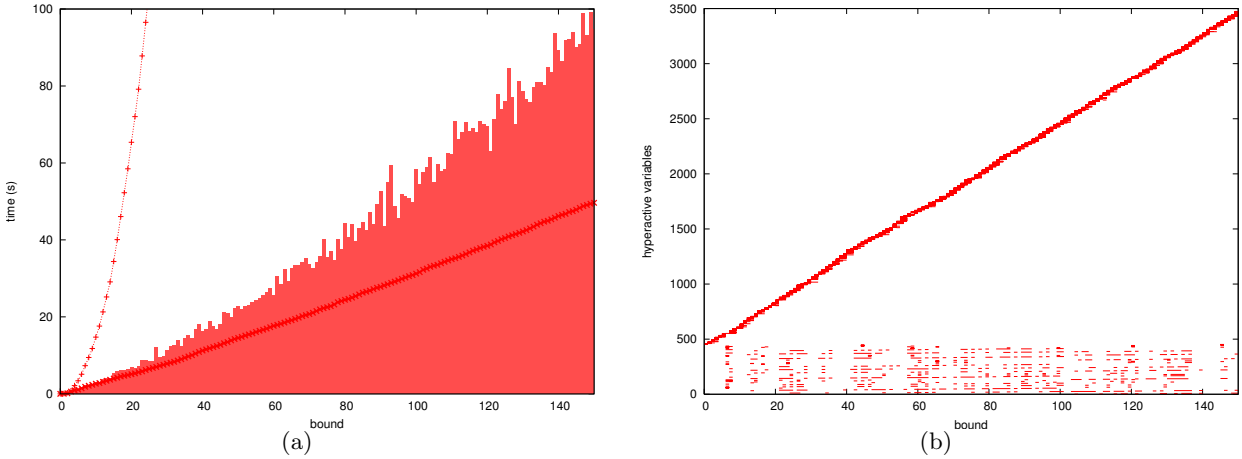
(a) (b)

Figure 1: Benchmark irst.dme6 from [3]



(a) (b)

Figure 2: Benchmark bc57sensors.p2neg from [3]



(a) (b)

Figure 3: Benchmark eijk.S1238.S from [3]

for the standard sequential strategy using a non-incremental solver. This line is intentionally not influencing the range of the y-axis, as it typically grows so large that it would make the other results hard to see.

From Fig. 1(a) it can be seen that the shortest counterexample for benchmark `irst.dme6` is of length 53. The run times of the non-incremental SAT-solver clearly show the behavior that inspired the opportunistic strategies of [15], i.e. the run time of the non-incremental solver for small satisfiable formulas is much smaller than that of the largest unsatisfiable ones. It may be observed from Fig. 2(a) that for benchmark `bc57sensors.p2neg` the run times for the two smallest satisfiable formulas corresponding to bounds 104 and 105 are relatively large. An easy satisfiable formula can however be found a little further ahead at bound 106, thus the use of an opportunistic strategy could possibly be beneficial.

Note that all results presented in this article demonstrate that the use of the incremental solver is crucial when performing the standard sequential strategy. Fig. 3(a) presents run time behavior for the benchmark `eijk.S1238.S` which is the encoding of model checking a property that holds on all execution paths of the model. This implies that the formulas are unsatisfiable for all bounds. Here, the crucial role that incremental SAT solving often plays in solving BMC is even clearer. Whereas a non-incremental solver would take about 100 seconds to find that bound 150 alone is unsatisfiable, the incremental solver finds this result for all bounds from 0 to 150 sequentially in half that time. This is typical behavior for many benchmarks corresponding to model checking a property that holds. It seems that in these cases the solver learns that the property holds for all short execution paths in a way that is easy to update when the bound on the length of the execution paths is extended. The solver can be thought of as having tuned itself towards verifying the property holds in the exact same way over and over.

Another way to look at the result presented in Fig. 3(a) is that by using the standard sequential strategy we are aiding the solver in proving the unsatisfiability of the formula corresponding to the counterexample of length 150, the largest bound tested here. By forcing it through the sequence of formulas we force a direction on the search that is natural to our problem description, and apparently this is helpful for the SAT-solver. For benchmarks with this kind of run time behavior there is clearly no hope for any opportunistic strategies.

An incremental solver can be started from any arbitrary bound, and it is possible to proceed by increasing the bound by more than one every time a formula is solved. Using bound increments larger than one is one of the simple strategies we have tried in our experiments. This strategy should still be considered opportunistic because of the "missing information" it causes for the solver, leading it further away from the efficiency of incremental solving, and further towards non-incremental behavior. Given the small margin of error available for opportunistic approaches for satisfiable benchmarks, and no chance of any performance improvement for many unsatisfiable benchmarks, we need to be careful when applying these approaches. They are however amongst the most natural ways of diversifying search strategies amongst nodes in an environment with parallel computation resources.

5 Parallel SAT solving

There are two common architectures for parallel SAT-solvers [11]. The first is the classic divide-and-conquer approach in which the formula is split into multiple disjoint subformulas each of which are then solved on a different computing node [4, 19]. The second approach is the so called portfolio approach [10]. The basic idea is that every computing node is running a SAT-solver that is attempting to solve the same formula. As modern SAT-solvers make some decisions randomly their run time varies greatly between runs. This makes the portfolio approach surprisingly efficient as it is able to decide the satisfiability of the formula as soon as the fastest solver finishes. Further diversification may be achieved by using different parameters on different computation nodes. Obviously opportunistic search strategies provide means for diversification when we are considering solving incrementally encoded SAT formulas in a parallel environment.

The current implementation of our parallelized BMC framework Tarmo can be seen as a parallelized incremental SAT-solver using the portfolio approach. Each computing node is running an incremental SAT-solver in the conventional sequential fashion. The novelty of Tarmo is that it allows sharing of conflict clauses between SAT-solvers even if they are working on different bounds. The solvers operate otherwise independently, i.e. if one solver solves a formula this does not stop the other solvers from attempting to solve that same formula. This choice was made after observing that interrupting a solver to make it "catch-up" with another breaks its ability to benefit from incremental SAT to the full extent. As we made this observation in an environment where clause sharing takes place it seems that the interrupted solver is missing more information than just conflict clauses. This was one of the reasons to look at the way the activity of variables propagates across bounds on the incremental SAT-solver runs.

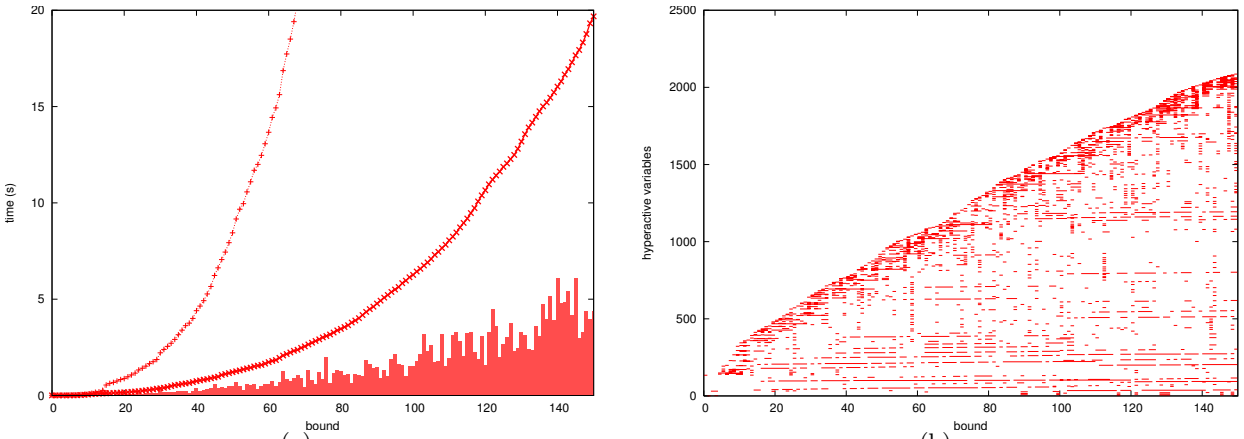
6 Variable activity

To obtain data on the activity of variables the SAT-solver was modified to print the activity of all variables after each bound it completed. For each bound we are interested in which variables are the most active, and especially in whether this activity remains high across several bounds. We consider a variable *hyperactive* if its activity is within the highest 2% of variables with non-zero activity.

The graphs labeled (b) in this article visualize the hyperactive variables. All variables that are hyperactive for at least one bound are represented by an integer value on the y-axis of the graph. The variables are sorted on the y-axis by their index such that if we define $y(v)$ as the integer on the y-axis corresponding to the variable with index v then for any $v' > v$ we have $y(v') > y(v)$.

Just like in the run time graphs the values on the x-axis of the graph represent bounds. If a variable was hyperactive starting from bound k up to but not including bound $k' > k$ then a horizontal line was drawn in the graph from bound k to k' at the y position corresponding to that variable. In other words for all variables v and all bound intervals $[k, k')$ on which v is hyperactive a line was drawn from $(k, y(v))$ to $(k', y(v))$.

One may observe that generally variables with larger indices become active later. This is because in the solver each newly introduced variable is given a larger index than all existing vari-



(a) (b)

Figure 4: Benchmark abp4.ptimoneg from [2]

ables, and for each bound a set of new variables is created. For each bound a subset of the new variables becomes hyperactive quickly, as the solver runs into conflicts on the newly added clauses.

The hyperactive variables in the satisfiable benchmarks `irst.dme6` and `bc57sensors.p2neg` are displayed in Fig. 1(b) and Fig. 2(b). Note that although for each bound some of the new variables become hyperactive all these variables tend to remain hyperactive throughout the whole process. This means that whenever a bound is added the solver still runs into new conflicts regarding variables that represent the state at smaller timepoints. We say that the activity of variables is *bound global*.

For the benchmark `eijk.S1238.S` the activity graph looks very different. For each bound the solver creates conflict clauses including the new variables, thus creating a new set of hyperactive variables, but there are only very few variables for which hyperactivity is maintained. This is in line with observation on the run time behavior of this benchmark which also indicate that hardly any work has to be performed to find unsatisfiability. We say that the activity of variables is *bound local*.

We have generated graphs like the ones presented in this paper for a large set of benchmarks[3]. We observe that on benchmarks with a bound global variable activity the run time of the non-incremental SAT-solver for the largest bound solved is smaller than the time spent for the incremental solver to get to the same bound and solve it. For benchmarks with a bound local variable activity this is never the case and thus a opportunistic heuristic will not improve performance.

Although we expect all hard satisfiable benchmarks to have a bound global variable activity it is not the case that all unsatisfiable benchmarks have a bound local variable activity. The benchmark `abp4.ptimoneg` represented in Fig. 6 is an example of an unsatisfiable benchmark with a bound global variable activity. Apparently the correctness of the property is not implied within a short number of execution steps here, and the incremental solver needs to evaluate large portions of the search space for every bound. Note also that for this benchmark an opportunistic ap-

[3]Available from `http://users.ics.tkk.fi/swiering`

proach may help to find unsatisfiable formulas at larger bounds faster.

7 Conclusions

In this article we have shown a relation between the run time of the standard sequential strategy for bounded model checking and the activity of decision variables in solvers employing this strategy. We can use this observation in a SAT-solver to predict during the search whether a more opportunistic strategy could be beneficial for the search. This is especially useful for parallel solvers in which different threads may be executing different strategies.

It is also easy to envision how these techniques could be useful for model checkers that use a combination of truly different model checking techniques such as PdTrav [5]. One could easily engineer a system which would do BMC for some amount of time, after which the variable activity could play a role in the decision on how to continue. If the variable activity appears to be bound local then the property is likely to hold for the model and thus we may want to start doing a complete model checking technique based on for example k-induction [16], Craig interpolation [13] or BDDs [12] to prove this.

Another observation we made is that for deciding the satisfiability of the last formula in an incrementally encoded sequence of formulas it can sometimes be faster to solve all formulas in the sequence. This raises the question whether other applications of SAT solving that currently rely on solving a single SAT formula could benefit from the use of incremental problem encodings. Such encodings allow enforcing a search direction on the SAT-solver that is natural to the application and therefore possibly beneficial to the solver. Tolerance to bad choices for such an incremental encoding could be achieved by doing this in a parallel environment with some opportunistic nodes.

References

[1] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In Rance Cleaveland, editor, *TACAS*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer, 1999.

[2] Armin Biere, Keijo Heljanko, Tommi A. Junttila, Timo Latvala, and Viktor Schuppan. Linear encodings of bounded LTL model checking. *Logical Methods in Computer Science*, 2(5), 2006.

[3] Armin Biere and Toni Jussila. Hardware model checking competition 2007 (HWMCC07). Organized as a satellite event to CAV 2007, Berlin, Germany, July 3-7, 2007.

[4] Max Böhm and Ewald Speckenmeyer. A fast parallel SAT-solver - efficient workload balancing. *Ann. Math. Artif. Intell.*, 17(3-4):381–400, 1996.

[5] Gianpiero Cabodi, Paolo Camurati, Luz Garcia, Marco Murciano, Sergio Nocco, and Stefano Quer. Trading-off SAT search and variable quantifications for effective unbounded model checking. In Alessandro Cimatti and Robert B. Jones, editors, *FMCAD*, pages 1–8. IEEE, 2008.

[6] Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia,

Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. NuSMV 2: An opensource tool for symbolic model checking. In Ed Brinksma and Kim Guldstrand Larsen, editors, *CAV*, volume 2404 of *Lecture Notes in Computer Science*, pages 359–364. Springer, 2002.

[7] Stephen A. Cook. The complexity of theorem-proving procedures. In *STOC*, pages 151–158. ACM, 1971.

[8] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5:394–397, July 1962.

[9] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

[10] Youssef Hamadi, Saïd Jabbour, and Lakhdar Sais. ManySAT: A parallel SAT solver. *JSAT*, 6(4):245–262, 2009.

[11] Antti Eero Johannes Hyvärinen, Tommi A. Junttila, and Ilkka Niemelä. Partitioning search spaces of a randomized search. In Roberto Serra and Rita Cucchiara, editors, *AI*IA*, volume 5883 of *Lecture Notes in Computer Science*, pages 243–252. Springer, 2009.

[12] Kenneth L. McMillan. *Symbolic model checking*. Kluwer, 1993.

[13] Kenneth L. McMillan. Applications of Craig interpolants in model checking. In Nicolas Halbwachs and Lenore D. Zuck, editors, *TACAS*, volume 3440 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 2005.

[14] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *DAC*, pages 530–535. ACM, 2001.

[15] Jussi Rintanen, Keijo Heljanko, and Ilkka Niemelä. Planning as satisfiability: parallel plans and algorithms for plan search. *Artif. Intell.*, 170(12-13):1031–1080, 2006.

[16] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a SAT-solver. In Warren A. Hunt Jr. and Steven D. Johnson, editors, *FMCAD*, volume 1954 of *Lecture Notes in Computer Science*, pages 108–125. Springer, 2000.

[17] João P. Marques Silva and Karem A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Trans. Computers*, 48(5):506–521, 1999.

[18] Siert Wieringa, Matti Niemenmaa, and Keijo Heljanko. Tarmo: A framework for parallelized bounded model checking. In Lubos Brim and Jaco van de Pol, editors, *PDMC*, volume 14 of *EPTCS*, pages 62–76, 2009.

[19] Hantao Zhang, Maria Paola Bonacina, and Jieh Hsiang. PSATO: A distributed propositional prover and its application to quasigroup problems. *J. Symb. Comput.*, 21(4):543–560, 1996.

Keyword Index

Author Index