Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Special section on Advances in Reachability Analysis and Decision Procedures

Contributions to abstraction-based system verification

Michael Huth[1], **Orna Grumberg**[2]

[1] Department of Computing, Imperial College London, South Kensington campus, London, SW7 2AZ, United Kingdom
[2] Computer Science Department, TECHNION - Israel Institute of Technology, Technion City, Haifa 32000, Israel

January 4, 2009

Abstract. Reachability analysis asks whether a system can evolve from legitimate initial states to unsafe states. It is thus a fundamental tool in the validation of computational systems – be they software, hardware or a combination thereof. We recall a standard approach for reachability analysis, which captures the system in a transition system, forms another transition system as an over-approximation, and performs an incremental fixed-point computation on that over-approximation to determine whether unsafe states can be reached. We show this method to be sound for proving the absence of errors, and discuss its limitations for proving the presence of errors, as well as some means of addressing this limitation.

We then sketch how program annotations for data integrity constraints and interface specifications – as in Bertrand Meyer's paradigm of Design by Contract – can facilitate the validation of modular programs., e.g. by obtaining more precise verification conditions for software verification supported by automated theorem proving.

Then we recap how the decision problem of satisfiability for formulae of logics with theories – e.g. bit-vector arithmetic – can be used to construct an over-approximating transition system for a program. Programs with data types comprised of bit-vectors of finite width require bespoke decision procedures for satisfiability. Finite-width data types challenge the reduction of that decision problem to one that off-the-shelf tools can solve effectively, e.g. SAT solvers for propositional logic. In that context, we recall the Tseitin encoding which converts formulae from that logic into conjunctive normal form – the standard format for most SAT solvers – with only linear blow-up in the size of the formula, but linear increase in the number of variables.

Finally, we discuss the contributions that the three papers in this special section make in the areas that we sketched above.

Key words: bounded reachability – abstraction – memory models – asynchronous systems – decision diagrams – decision problems – bit-vector arithmetic

1 Introduction

We are pleased to introduce three outstanding papers that report advances in reachability analysis and in decision procedures. These papers were published in a conference version in the proceedings of the *Thirteenth International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (TACAS 2007), held as part of the *European Joint Conferences on Theory and Practice of Software* (ETAPS) which ran from 24 March until 1 April 2007 in Braga, Portugal. These papers were then rewritten and extended as journal papers, by our invitation. The criteria for inviting papers were that papers were amongst the top papers published in the TACAS 2007 proceedings, and that the papers reflected well on the intent of this journal: software tools for technology transfer. So we very much hope that this special section will be enjoyed by the readership of this journal.

The purpose of this introductory paper is to put the papers of this special section and their contributions into an appropriate context and perspective, and to provide some background – in an informal tutorial style – that will make the technical aspects of these papers even more accessible than they already are. Given that these papers draw from a very wide range of techniques and results, we can naturally only select some key issues in our tutorial-style discussion and have to refer to the existing literature for some technical aspects of these papers.

The tutorial part of this introductory paper begins in Section 2, and the discussion of the special-section papers can be found in Section 5.

2 Verifying safety through reachability analysis

The safety of programs is a concern of growing significance. More and more, our daily lives and interactions are controlled or assisted by devices that have computer programs embedded in them – whether we like it or not. The reliable functioning of these programs, therefore, becomes ever more important. Many program errors can be avoided by detecting them at compile time. Typed programming languages, e.g., help to shield against run-time errors that result from dynamic type mismatches. Program analysis tools, e.g., can alert to problematic aspects of programs such as a portion of code that can never be reached in program execution. But some program errors are subtler or challenging to detect, e.g.

- the permanent blocking of a computation thread
- the corruption of structured data

Two instances of the latter are the inadvertent creation of a cycle in an acyclic list, and the violation of an important data integrity constraint such as that account balances be no less than the allowed overdraft limit.

It turns out that many pratically relevant program errors can be abstractly characterized as being able to reach system states that we deem to be dangerous or undesirable. For example, whenever an account balance falls below an allowed overdraft limit we can assume that this stems from some portion of code that is able to change the account balance. We can therefore insert "assert" statements at all such locations in order to check whether the integrity constraint is being preserved:

```
assert accountBalance >= allowedOverdraft;
```

where `accountBalance` and `allowedOverdraft` are program variables already declared and managed for an appropriate type such as `double`. Programs execute such assert statements by evaluating their Boolean expressions, here

```
accountBalance >= allowedOverdraft
```

If the Boolean expression evaluates to `true`, the statement has no effect other than its mere evaluation. If it evaluates to `false`, an error state has been reached. Most research on program verification concerns itself with this type of program error, and many approaches to program verification utilize the above technique of defining error states by annotating programs with assert statements.

2.1 Programs as formal transition systems

A prominent formal model of a computer program, then, is a transition system consisting of

- a set of system states S,
- a transition relation $R \subseteq S \times S$ where $(s, s') \in R$ models that the program may – if in system state s – have the atomic effect of changing that system state to s' at run-time,
- a subset I of S that models those system states in which the program may start executing, and
- a subset E of S that models those system states that the verifier deems to be an error.

Set E may reflect a specific kind of error, such as accessing any dangling program pointers. Alternatively, a verifier may simply bundle all relevant error types together into this set. Relation R may be non-deterministic if the program constructs contain, e.g., a pseudo-random choice operator. In addition, set S is often unbounded, since the program may support unbounded data types, an unbounded number of threads, etc.

2.2 Sound reachability analysis through abstraction

Given such a formal model of the program and its error states, how can we check its safety, i.e. that no error state can ever be reached? Conceptually, this is very easy. Let

$$\mathsf{next}(X) = \{s' \in S \mid \exists s \in X \colon (s, s') \in R\} \tag{1}$$

be the set of system states that can be reached from any state in $X \subseteq S$ in one R-step. Then the program can never reach an error state from any initial state if, and only if

$$E \cap \bigcup_{n \geq 0} \mathsf{next}^n(I) = \{\} \tag{2}$$

where

$$\begin{aligned} \mathsf{next}^0(X) &= X \\ \mathsf{next}^{n+1}(X) &= \mathsf{next}(\mathsf{next}^n(X)) \qquad (n \geq 0) \end{aligned} \tag{3}$$

Equation (2) stipulates that if the program begins in a state from I, then no finite sequence of R-steps can reach an error state in E. This is so since $\mathsf{next}^n(X)$ captures those states that can be reached from states in X by an R-path of length n. Thus this correctly captures the desired safety property.

The problem with this notion, though, is that S and R may be infinite or very large and so an incremental computation of $\bigcup_{n \geq 0} \mathsf{next}^n(I)$ may be infeasible. One way of addressing this is by means of abstraction [11, 9, 12, 13]. Suppose that there is another, more abstract, *finite* set of states S^α and a relation $\rho \subseteq S \times S^\alpha$ such that

$$\forall s \in S \, \exists t \in S^\alpha \colon s \rho t \tag{4}$$

where we write $s\rho t$ for $(s, t) \in \rho$. Then we can make S^α into an abstract program model by setting

$$\begin{aligned} R^\alpha &= \{(t, t') \in S^\alpha \times S^\alpha \mid \exists s\rho t, s'\rho t' \colon (s, s') \in R\} \\ I^\alpha &= \{t \in S^\alpha \mid \exists s \in I \colon s \rho t\} \\ E^\alpha &= \{t \in S^\alpha \mid \exists s \in E \colon s \rho t\} \end{aligned} \tag{5}$$

So abstract states are initial (respectively, error states) if they are related to some concrete initial (respectively, error) state via ρ. Similarly, a transition between abstract states is one that corresponds to a concrete transition where the concrete source and target states relate to the abstract source and target states via ρ. Since S^α is finite, we can now verify

$$E^\alpha \cap \bigcup_{n \geq 0} \mathsf{next}^n(I^\alpha) = \{\} \tag{6}$$

by incrementally computing the finite set $\bigcup_{n\geq 0} \mathsf{next}^n(I^\alpha)$, where the definition of next is as before but now in (S^α, R^α). This turns out to be a sound method for showing that the program cannot reach any error states: If (6) holds, we are assured that (2) holds as well – provided that ρ is such that is satisfies (4).

The proof of this important fact is instructive as it reveals that the abstract program model was defined expressly in order to secure this fact:

> Using proof by contradiction, assume that (2) is false. Then there is an R-path $s_0 R s_1 \ldots s_{n-1} R s_n$ from an initial state $s_0 \in I$ to some error state $s_n \in E$. By (4) there is some $t_0 \in S^\alpha$ with $s_0 \rho t_0$. By definition of I^α, we get $t_0 \in I^\alpha$. By induction on n and by the definition of R^α, we get an R^α-path $t_0 R^\alpha t_1 \ldots t_{n-1} R^\alpha t_n$ with $s_i \rho t_i$ for all $0 \leq i \leq n$. We merely show the induction base case for this claim: *from (4) we infer the existence of some $t_1 \in S^\alpha$ with $s_1 \rho t_1$. But then $(s_0, s_1) \in R$ and $s_0 \rho t_0$ imply $(t_0, t_1) \in R^\alpha$*. In particular, $t_n \in E^\alpha$ since $s_n \in E$ and $s_n \rho t_n$. To summarize, we constructed an R^α-path from an element in I^α to an element in E^α, contradicting (6).

Condition (4) is often referred to as saying that relation ρ is "left total". It holds automatically for many methods of synthesizing abstract program models, e.g., for predicate abstraction [15,1], where each state t^α is a bit-vector modelling which set of fixed predicates a concrete program state s satisfies – and so every concrete state naturally has such a corresponding bit-vector and ρ is then the graph of a total function.

2.3 Counter-example guided abstraction-refinement

Unfortunately, we cannot always infer that the program can reach error states if the abstract program model can reach an abstract error state: the non-emptiness of (6) does generally not imply the non-emptiness of (2). The reason is that, from $(t, t') \in R^\alpha$, $s \rho t$, and $s' \rho t'$ we cannot conclude that $(s, s') \in R$ and so not every R^α-step has a corresponding R-step for a designated concrete successor state. So an abstract counter-example to the safety of the program (an R^α-path from set I^α into set E^α) may be spurious in that it has no corresponding concrete R-path from set I into set E. It is thus no surprise that techniques have been developed that attempt to

- either verify that the abstract counter-example has indeed a corresponding concrete counter-example (e.g. [8]), or
- failing that, the spurious nature of the abstract counter-example can be used to refine set S^α into a larger set S^{α_1} and to redefine ρ over $S \times S^{\alpha_1}$ (e.g. [8,10,14]).

The hope is then that either there are no abstract counter-examples in the more concrete program model $(S^{\alpha_1}, R^{\alpha_1}, I^{\alpha_1}, E^{\alpha_1})$, or it has a non-spurious counter-example. In general, one may have to iterate this process to compute a sequence of abstract but increasingly more concrete program models

$$\mathcal{M}^{\alpha_k} = (S^{\alpha_k}, R^{\alpha_k}, I^{\alpha_k}, E^{\alpha_k}) \qquad (k \geq 1) \tag{7}$$

that all over-approximate the concrete program model (S, R, I, E) so that

- there is some $k \geq 1$ for which $\mathcal{M}^{\alpha_k}$ has no path from an initial state to an error state – and so (2) holds
- there is some $k \geq 1$ for which $\mathcal{M}^{\alpha_k}$ has a path from an initial state to an error state, and that path has a corresponding path from an initial state to an error state in (S, R, I, E) – and so (2) does not hold
- or for all $k \geq 1$ the abstract program model $\mathcal{M}^{\alpha_k}$ has spurious abstract counter-examples but none of them correspond to concrete error paths – and so the computation diverges.

This iterative process, over-simplified above, is known as "counter-example guided abstraction-refinement" (CEGAR) and many verification tools use this scheme for the verification of the unreachability of error states (e.g. [8, 10,14]). This process is not confined to verifying safety properties. For example, the paper *"An Abstraction-Based Decision Procedure for Bit-Vector Arithmetic"* in this special section uses a similar process to decide whether a formula of some logic is satisfiable.

2.4 Programming by contract

Computer programs are not written as monolithic instruction sets but are composed out of instantiations of parameterized modules. This facilitates code re-use – e.g., the use of library calls – and makes code development more scalable, reliable, and maintainable. Thus, assert statements that check for program errors will be placed within such modules. But module boundaries and module composition offer additional opportunities for specialized assert statements.

We now illustrate this idea in a style similar to that of the Java Modeling Language (JML) (with home page at `www.eecs.ucf.edu/~leavens/JML/`) on a simple example, adapted from [18]. In doing so we will deliberately over-simplify some of the issues. Consider the declaration of a class in some object-oriented programming language

```
public class Person {
 invariant !name.equals("") & this.weight >= 0;
 private String name;
 private int weight;
... }
```

It states that an object of type `Person` has two private fields `name` and `weight`. It also declares an *invariant*, namely that the name not be the null string and that the weight be non-negative. These are sensible constraints on such data for persons. Invariants are meant to hold after an object of that type has been constructed, and before and after each non-constructor method call that manipulates objects of that type. In particular, the invariant can be violated during the execution of such methods. Invariants are thus an effective means of expressing that data integrity constraints are enforced at the appropriate program states. It should be clear that we could check compliance with an invariant by writing corresponding assert statements and placing them around method and constructor boundaries.

Invariants are not the only application of placing assert statements around method boundaries. Bertrand Meyer's notion of Design by Contract [19] suggests that each method has an explicit contract: a specification of what the caller of a method has to ensure prior to a call (often called a "*pre-condition*"), and a specification of what the callee will guarantee upon termination of that method (often called a "*post-condition*").

For example, consider the interface specification

```
requires kgs >= 0;
ensures weight == \old(weight) + kgs;
public void addkgs(int kgs) { ... };
```

for method `addkgs` within class `Person`. That method takes as input an integer variable `kgs`, returns nothing (as the return type is `void`), and has its computation as a side effect on the state of an object of type `Person`.

The contract for this method consists of two parts. The keyword `requires` indicates a pre-condition, in this case that the method can only be called with input values `kgs` that are non-negative. The keyword `ensures` indicates a post-condition, in this case that the value of the field `weight` in the object of type `Person` equals the sum of the value it had prior to the method call (indicated by the wrapper `\old`) with the input value `kgs`. The semantics of this contract is usually that of *partial correctness*: the post-conditions only have to be true if the method actually returns (it could diverge).

It is pretty intuitive to see how such annotations would expand into assert statements: all pre-conditions have to be enforced prior to the execution of the first statement in the method body; and all post-conditions have to be enforced prior to any method return points. At run-time one can therefore use these interface specifications for testing. But these interface specifications can also be used in static reasoning where, in contrast, pre-conditions are assumed to be true: For each method, we can statically generate verification conditions that imply that all post-conditions hold upon method return, under the assumption that all pre-conditions and all class invariants hold prior to the method call. For example, if we implement the method as in

```
requires kgs >= 0;
ensures weight == \old(weight) + kgs;
public void addkgs(int kgs) {
                  weight = weight - kgs;
}
```

we can then derive from this implementation and its contract a verification condition that captures that the implementation honors the contract. Here, this verification condition is that the quantifier-free formula of first-order logic with arithmetic

$$\texttt{kgs} \geq 0 \wedge (\texttt{weight} = \texttt{old}(\texttt{weight}) - \texttt{kgs}) \rightarrow (\texttt{weight} = \texttt{old}(\texttt{weight}) + \texttt{kgs}) \quad (8)$$

be valid, i.e. universally true. This is thus a task we could delegate to a theorem prover for that logic. Such a prover would certainly detect that the formula is not valid, take for example $\texttt{kgs} = 2$ and $\texttt{weight} = 67$. In fact the formula is true only when the value of `kgs` is negative (in which case the contract is mute) or zero (in which case the mistake of using subtraction instead of addition has no effect).

Another important interface specification is that of "*frame conditions*". For a method, we often want to say that it may only change the state of certain variables. For the method above we could state

```
modifies weight;
public void addkgs(int kgs) { ... };
```

saying that method `addkgs` may modify the value of variable `weight` (but is not required to do so), and that that method must not modify the value of any other program variable. In particular, this states that method `addkgs` must not change the value of private field `name`.

We could achieve a similar effect for `weight` with the specification `ensures weight = \old(weight)` but this would not capture that any other variables won't change, and we may not even know the exact set of other variables since we may not know in which context method `addkgs` operates. Thus the frame condition expressed through the keyword `modifies` adds extra strength to interface specifications. But that strength also makes it very difficult to reason about frame conditions in a modular manner.

Finally, one may need the ability to check constraints for each atom of a composite data object. For example, the code fragment

```
Object[] a;
...
ensures \forall int i;
            0 <= i < a.length; a[i] != null;
public void someMethod() { ... };
```

uses a specification pragma for universal quantification. Method `someMethod` has as post-condition, saying that all elements of array `a` be non-null. A pragma for existential quantification (`\exists`) is written in the same style. Enforcing such constraints is more involved. In this example, a dynamic check during testing needs to traverse the entire array and verify that all array elements have non-null object references. A static check of verification conditions may require the use of theorem provers that can deal with quantified formulae of first-order logic with arithmetic. Such provers are known to often struggle when reasoning about quantifiers.

Annotation languages such as the one sketched above have been used for the static verification of programs (e.g. JML and ESC/Java [6], and Spec# [2]), and for the automated generation of unit tests (e.g. [20]). Another application of annotation languages is that they can be used to provide a mapping between some abstract system model and concrete program code. Such an application can be seen in the paper *"A Low-Level Model and the Accompanying Reachability Predicate"* in this special section that uses an abstract yet formal model of low-level memory.

3 Decision procedures for logics

As we have already touched upon, logics are natural formal languages for expressing program state and program behavior. For example, a bit-vector over two variables x_1 and x_2 may describe the abstract state of a computer program where truth of x_1 encodes that $y < z + 1$ is true, and truth of x_2 represents that $y \neq 0$ is true, and where these predicates may stem from control expressions in said computer program. Given an atomic program statement such as $y := y + z$; we then want to understand its effect *as a transducer of abstract states.*

3.1 Satisfiability checks generate abstract transitions

For example, if the current abstract state is 01, saying that x_1 is false and x_2 is true, is 11 a possible abstract successor state of 01 after that assignment executed? Let us write y' and z' to represent the values of these variables after that assigment. Then 11 is an abstract successor state of 01 if, and only if, the quantifier-free formula of first-order logic given by

$$\begin{aligned}\phi_1 = \neg(y < z + 1) \wedge (y \neq 0) \wedge (y' < z' + 1) \wedge (y' \neq 0) \\ \wedge (z' = z) \wedge (y' = y + z) \end{aligned} \quad (9)$$

is satisfiable. These six conjuncts encode the abstract state 01 (first two conjuncts), the putative abstract successor state 11 (next two conjuncts), and the effect of the assigment (the last two conjuncts). This formula is satisfiable, e.g., choose $y = -1$, $z = z' = -10$, and $y' = -11$. So we can conclude that there is an abstact transition from 01 to 11 for the transducer $y := y + z$.

Now what if we refine our abstract model to introduce a third variable x_3 whose truth represents the truth of $z \geq 0$? From the abstract state 011, then, execution of $y := y + z$ cannot lead to any abstract state $11b$ for $b \in \{0, 1\}$ since the formula

$$\phi_2 = \phi_1 \wedge (z \geq 0) \wedge (z' \geq 0) \quad (10)$$

is unsatisfiable. To see this, we have $y + z = y' < z + 1$ and so $y+z < z+1$ which implies $y < 1$. Since $y \neq 0$ this renders $y < 0$ and so from $z + 1 \leq y < 0$ we infer that $z < -1$ which contradicts the conjunct $z \geq 0$ in (10).

Needless to say, such manual derivations are error-prone and won't scale to real programs. Therefore we want fully automated techniques for deciding such problems. But note that our reasoning above implicitly assumed a "background theory", meaning that we interpreted the constants 0 and 1, equality $=$ and the strictly less than relation $<$ according to rules that we would expect to hold, e.g. that $<$ is transitive, that $0 < 1$, etc. So we need to consider decision problems for logics enriched with appropriate theories.

If the decision problem of satisfaction is undecidable for a logic – and it is for many – we want automated techniques that approximate such decisions soundly. Suppose we could not decide whether a formula such as (10) is satisfiable. For verifying the unreachability of error states it is then always sound to state that a formula *is* satisfiable even though we do not know this and it may be false. The reason is that this finds all abstract transitions between all abstract states of the computer program that stem from satisfiable formulae, and may just add some extra, spurious transitions for unsatisfiable formulae. So this *over*-approximates the abstract transition system obtained by always correctly guessing satisfiability. If on that over-approximating abstract model no error state can be reached from an initial state, then this is surely true in the abstract model it over-approximates – as we discussed and proved already in the previous section. By the same token, the latter would then guarantee that the computer program itself cannot reach such error states, since any path in the computer program has a corresponding path in that abstract model by construction – again, by the same argument as already made in the previous section.

3.2 Satisfiability checks for bit-vector arithmetic

The logic of our example was the quantifier-free fragment of first-order logic with a theory of ordered arithmetic. This seemed entirely appropriate for program variables y and z that have integral or real numbers as values. But most practical programming langugages declare integer or real types with a fixed precision. This creates friction with the clean and idealized world of a mathematical

logic. For example [17], consider the formula

$$\neg(x = y) \wedge \neg(x = z) \wedge \neg(y = z) \tag{11}$$

in the logic aforementioned. This formula is clearly satisfiable, e.g. take $x = 0$, $y = 1$, and $z = 2$. But what if these variables are declared as bit-vectors of width 1, i.e. as single bit flags? Then the formula becomes unsatisfiable. This illustrates the challenges faced by decision procedures for formulae with bit-vector variables of fixed width.

A naive Boolean encoding of abstract states needs to be enriched with constraints that accurately capture the behavior of constants, functions, and relations within their finite width context-of-use. A technique known as "bit blasting" [3] translates in this manner a formula of bit-vector arithmetic into a formula of propositional logic such that the latter is satisfiable if, and only if, the fomer is. Thus one can reduce the decision problem for formulae over bit-vector arithmetic to that of formulae of propositional logic.

3.3 Relevant concepts from propositional logic

Propositional logic is generated by the grammar

$$\phi ::= x \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \tag{12}$$

where x is from an unbounded set of Boolean variables, $\neg$ denotes negation, $\wedge$ conjunction, and $\vee$ disjunction. The decision problem for this logic, whether a formula is satisfiable, univerally referred to as SAT, is hard. All known algorithms that decide this problem have exponential worst-case running times in the size of the input formula. At the same time, a history of SAT solver building and fine-tuning has resulted in today's sophisticated SAT solvers that can solve this problem on many if not most formulae that reflect encodings of practical applications such as planning problems in artificial intelligence, equivalence checking of hardware circuits, etc. But there are exceptions such as checking the satisfiability of some unsatisfiable formulae. For example, one can encode the question of whether n pigeons can fit into $n-1$ pigeon holes without any pigeons sharing holes as a formula of propositional logic. Such encodings are unsatisfiable by construction yet modern SAT solvers struggle to show this for moderate sizes of n for such encodings (see e.g. the discussion in [17]).

Most SAT solvers operate on formulae in conjunctive normal form (CNF), generated by the grammar

$$\begin{aligned} L &::= x \mid \neg x \\ D &::= L \mid L \vee D \\ C &::= D \mid D \wedge C \end{aligned} \tag{13}$$

as expressions C. In particular, such formulae are in negation normal form (NNF): only variables are negated. For example $\neg x_1 \wedge (x_2 \vee x_1) \wedge (x_1 \wedge x_3 \wedge \neg x_2)$ is in CNF.

Fortunately, there is a technique for efficiently encoding formulae in CNF: Tseitsin's encoding [21] takes a formula ϕ of n variables and computes a formula $T(\phi)$ in CNF with $n+p$ variables where p and the size of $T(\phi)$ are linear in the size of ϕ. The price we pay are the extra p new variables, whose values for satisfying assignments also do not really interest us. What we gain is that ϕ is satisfiable if, and only if, $T(\phi)$ is satisfiable.

For example (adapted from [17]) let $n = 3$ and $p = 2$, where the NNF $\phi = \neg x_1 \vee (x_2 \wedge x_3)$ gets a new variable for each binary operator: a_1 for $\vee$ and a_2 for $\wedge$. We then stipulate that a_1 is true if, and only if, ϕ is true; a_2 is true if, and only if, $x_2 \wedge x_3$ is true; and a_1 is true (since ϕ needs to be made true). Each stipulation is easily transformed into a short CNF – e.g. the first stipulation turns into $(a_1 \vee x_1) \wedge (a_1 \vee \neg a_2) \wedge (\neg a_1 \vee \neg x_1 \vee a_2)$ – and the conjunction of these three CNFs is the desired Tseitin encoding of ϕ.

For a CNF ϕ that is unsatisfiable, an *unsatisfiable core* is any subset of its clauses C that, seen as a CNF in its own right, is again unsatisfiable. Naturally, one is interested in a small subset of such clauses and many SAT solvers that operate on CNFs will compute such a small unsatisfiable core when they check the satisfiability of an unsatisfiable input formula. Furthermore, resolution-based methods can then be used on an unsatisfiable core to procude a formal proof that this core, and therefore also ϕ, is unsatisfiable. This complements thus the ability to find a satisfying assignment if a formula is satisfiable. The latter is very easy to do for SAT solvers. The former proof of unsatisfiability, though, requires more work as it needs to find such a formal proof.

4 Bounded reachability analysis

Bounded reachability is a technique that attempts to detect errors in systems by unfolding the system from its initial states a finite, bounded number of times. This technique thus computes an *under-approximation* of the set of states reachable from the initial states. Specifically, let (S, R, I, E) be our system model as above. We can then define the set of states reachable from the initial states through method `reachability` whose body performs a least fixed-point computation, where `union` denotes set union, `I` denotes I, and `next(X)` denotes the set $\mathsf{next}(X) = \{s' \in S \mid \exists s \in X : (s, s') \in R\}$ already defined earlier on:

```
reachability() {
 Reach = I;
 repeat {
  Cache = Reach;
  Reach = Reach union next(Reach);
 } until (Cache == Reach)
 return Reach;
}
```

If S is infinite, then this computation may well diverge. The computation, as defined in this method, is also not the most efficient one for finite sets S, but we here focus on conceptual clarity. We can, similarly, write a method `boundedReachability` that takes as input a finite bound and computes a bounded version of the above fixed-point:

```
boundedReachability(B : int) {
 Reach = I;
 for (i = 1; 0 < i <= B) {
  Reach = Reach union next(Reach);
 }
 return Reach;
}
```

Whenever the bound is non-negative we have that the set of states returned by `boundedReachability(B)` is contained in the set of states `reachability()`. This is useful since it gives us a sound way of exposing errors. Let `E` represent the set E of error states, let `and` denote set intersection, and let {} represent the empty set in method `errorFound`:

```
errorFound(B : int) {
 return ( E and boundedReachability(B) != {} )
}
```

which returns a Boolean value. When this method returns `true`, we know that there is some state $s \in S$ that is reachable from some initial state $i \in I$ such that $s \in E$. Therefore, we have found an error. It is a different matter, though, to then expose a witness path from i to s. The method `errorFound`, as written, is of little help here since it just intersects the bounded reachability set with the set of error states. Such Boolean approaches to error detection forget the temporal structure of paths and thus may have to do some post-processing so that useful diagnostic information can be regenerated. For example, knowing that $s \in E$ is reachable from I, one could do a backwards search in (S, R) from s and stop as soon as some element of I has been reached.

The advantage of using bounded reachability analysis is that we can unfold the system incrementally and expose errors as soon as they occur in the unfolding of the system. For example, concurrency errors such as deadlocked states typically will happen on a relatively short transition path in (S, R) from I, and so a finite, incremental unfolding may mean that the system size stays manageable for tools until errors are found. A full system check, on the other hand, may never complete as tools may not be able to handle the size of the full system.

The disadvantage of bounded reachability analysis seems to be that when method `errorFound` returns `false` we only know that there was no error within the first B unfoldings. But we do not know what happens in subsequent unfoldings of the system. The error may just be one unfolding away. It is therefore not suprising that research has tried to address this shortcoming.

One approach, for example, is to establish the diameter d of the directed graph (S, R) – e.g. by an instrumented bounded reachability analysis [16][page 183]. Then $B = d$ as bound will guarantee that the method `boundedReachability(B)` computes the same set as does method `reachability()`, since there is no cycle in (S, R) larger than d. This technique is effective, e.g., for reasoning about some protocols. So domain-specific knowledge may be exploitable to compute graph diameters.

Equally, knowledge of the structure of a system may guide the modeling and implementation of bounded reachability analysis. For example, consider bounded deadlock detection in a system that is *globally asynchronous and locally synchronous*. We may abstractly represent such a system via a set of asynchronous events $\mathcal{E} = \{\alpha_1, \alpha_2, \ldots, \alpha_K\}$. The overall system model is then

$$\begin{aligned} S &= S_1 \times S_2 \times \cdots \times S_K \\ S_i &= \{0, 1, \ldots, n_i - 1\} \\ I &= \text{whatever the system demands} \\ E &= \{s \in S \mid \forall s' \in S \colon (s, s') \notin R\} \end{aligned} \tag{14}$$

So a state is a tuple $(s_1, s_2, \ldots, s_k)$ where each local state s_i corresponds to event $\alpha_i \in \mathcal{E}$ and is from a state space of finite size $n_i > 0$. The initial states are those required by the application. The error states are those that do not have a successor state (those states that deadlock). Finally, the transition relation R is defined as follows: we have $(s, s') \in R$ if, and only if, for all $1 \leq i \leq K$ we either have that the local state s_i for event α_i equals s'_i, or s'_i is the result of the synchronous effect of α_i on local state s_i. For example, that effect may be the parallel execution of several assignments to local variables (not modeled above). Abstractly, one can write the entire transition relation R as a disjunction of conjunctions $\bigvee_{\alpha_i \in \mathcal{E}} \bigwedge_r \mathsf{synchrEffect}_r^{\alpha_i}$ where r ranges over the resources that are affected by α_i in local state s_i and each constraint $\mathsf{synchrEffect}_r^{\alpha_i}$ accurately captures how this effect can transform resource r in local state s_i.

The challange and opportunity here is to exploit the logical structure of this formula, its two-level layer, so that the locality of interactions can lead to more compact representations of the bounded reachability analysis. Such a development is at the heart of the paper *"Decision-diagram-based Techniques for Bounded Reachability Checking of Asynchronous Systems"* featured in this special section.

5 The special section

All papers in this special section demonstrate advances in reachability analysis for computer systems or in the efficiency of decision procedures that reason about important data types for programs. We discuss briefly the contributions of each of these papers:

"An Abstraction-Based Decision Procedure for Bit-Vector Arithmetic" deals with the decision problem for quantifier-free formulae over bit-vector arithmetic for vectors of bounded width. That is to say, given a formula from that logic, is this formula satisfiable? The decision problem for these formulae often has to be implemented as part of a verification tool, whether it reasons about software which – in part – manipulates data structures built out of bit vectors (e.g. code for 64-bit integers) or whether it reasons about hardware structures that manipulate bit-vectors to realize desired functionality at low system levels (e.g. checking the equivalence of a higher-level hardware program and its realization at the register transfer level). Efficient techniques for deciding this problem are therefore desirable and will have high impact on improving such verification tools.

The approach taken in this paper is as follows: from an initial formula an under-approximating formula is computed by translating it into propositional logic – as familiar from "bit blasting" – but where some bit-vector variables are represented with fewer Boolean variables than the width of the bit-vector variable. If that under-approximation is satisfiable (which can be checked with any SAT solver), the initial formula is satisfiable, too. If the under-approximation is unsatisfiable, a proof of unsatisfiability can be generated from an unsatisfiable core (where the SAT solver needs to support such a proof generation feature). The terms participating in that proof are then used to construct an over-approximation of the initial formula. If that over-approximation is unsatisfiable (checked by any SAT solver), then so is the initial formula. Otherwise, a satisfying model of the over-approximation (revealed by the SAT solver) can guide the refinement of the under-approximation by increasing the number of Boolean variables that are used for representing some bit-vector variables. This CEGAR-like process is completely automated, guaranteed to terminate, and renders a correct implementation of the above decision procedure.

The avdantage of this technique over, say, conventional "bit blasting" is that it can be much more efficient than conventional methods when satisfiable bit-vector formulae have solutions that are representable via a small number of bits. It also is expected to work well for unsatisfiable bit-vector formulae that have a proof of unsatisfiability involving only a relatively small number of terms. The paper describes this approach in detail and its experimental results corroborate that this technique can dramatically improve performance for formulae of the aforementioned types and for other benchmarks in software verification.

Other points of interest in this paper are that

- the approach can work on formulae represented as Boolean circuits, as opposed to representations in conjunction normal form, and this has advantages when dealing with specific bit-vector operators such as ITE
- the approach does not rely on a chosen set of operators for bit arithmetic but only requires that variables have finite width and that formulae have propositional encodings
- it offers translations from the Boolean encodings of the over- and under-approximations into conjunctive normal form and so generalizes, in a way, Tseitin's encoding to the class of bit-vector formulae.

We are pleased to report that this approach has already been adopted by Synopsys (`www.synopsys.com`), a company that – according to its web site – "provides tools and services for digital system-on-chip design", in one of their tools called Hector – an equivalence checker between RTL and C++.

The authors of **"A Low-Level Model and the Accompanying Reachability Predicate"** note a genuine need to transfer existing methods and tools for verifying object-oriented programs written in a high-level programming language such as Java, C++ or C# into the realm of systems software. The latter, though, is mostly written in low-level languages – such as C – that have access to and exploit low-level features of computer memory. The authors point out that already the verification of programs written in a high-level language such as Java is challenging since the verfication conditions generated for such programs need to capture – e.g. – the heap shape of dynamically allocated data structures, arithmetic constraints on values of numeric program variables or interface specifications for method boundaries.

Verification conditions are often written as formulae of some logic. The encoding of dynamically linked data structures and their state therefore needs to rely on the ability to state that one node of the heap is reachable by another node in the heap through links of perhaps a specific kind. Transitive closure operators naturally capture such reachability predicates. But it is well known that such operators and precidates are not expressible in first-order logic (which allows for the quantification over program varibles but not over relations on such variables).

The authors thus consider a first-order logic extended with a reachability predicate and relevant theories that has already been used in the verification of Java-like programs. In this paper they adapt this logic and the process of verification-condition generation to obtain relatively exact verification conditions that allow the verification of a practically important wide range of programs for system software.

The logic used for higher-order languages matches well the encapsulation of low-level memory models provided in such Java-like languages, since heap nodes of interest are captured directly by object references, and thus as variables in the logic. Many linked data structures in system software, however, are lists whose nodes are continous segments of memory, and heap nodes of

interest are often only found within such list nodes by performing a contant shift of bytes in that segment of memory. For each such constant shift in the program's data structures, the authors define a correponding reachability predicate that accurately models this constant shift.

A challenge in this approach to the generation of verification conditions of heap-manipulating programs remains: Program effects that change the heap shape or state need to be reflected in corresponding updates of the impacted reachability predicates so that they still accurately model the current state of the heap. Additionally, the generation of verification conditions needs a formal abstract memory model so that all operations on memory, be they access-related (such as a write operation) or management-related (such as the freeing of a segment of memory), get an operational semantics based on this abstract model. This is a delicate task as there is an inherent trade-off between the scalability of program verification and the degree of precision of this abstract memory model.

In this paper the authors describe how one can automatically compute the precise updates for the low-level reachability predicates, and how theorem provers for first-order logic can reason about such predicates soundly. They also design an annotation language that uses familiar constructs from design by contract but also allows the use of the low-level reachability predicates and a special contract pragma for memory management. The prototype implementation HAVOC takes a C program with such annotations and translates this into an annotated BoogiePL program. The Boogie verifier next generates the verification conditions whose validity then implies partial correctness of the BoogiePL program. Finally, the verification condition is then checked for validity with the Z3 theorem prover.

The authors validate their approach on a representative set of small to moderately sized C programs. This shows that, in principle, one may obtain strong verification engines for systems software based on automated theorem proving of first-order logic. Two promising avenues for future work are the development of techniques that can infer annotations – thus alleviating the annotation burden put on programmers or verifiers, and the extension of these low-level reachability predicates to data structures that are more complex than linked lists – such as trees.

"Decision-diagram-based Techniques for Bounded Reachability Checking of Asynchronous Systems", last but not least, develops new approaches to bounded reachability analysis, specifically for finding deadlocks in systems that exhibit global asynchronicity but local synchronicity. The authors model such systems abstractly by a set $\mathcal{E}$ of (asynchronous) events such that each event $\alpha \in \mathcal{E}$ is in turn modeled by a synchronous composition of smaller components. Such models then have states that are tuples over $\mathcal{E}$ as an index set. Entries of these tuples represent states for the corresponding events. The transition relation of such systems is then an asynchronous composition of locally synchronous compositions of events. States and transitions alike are thus amenable to symbolic representations. For example, the transition relation is logically in disjunctive normal form. And so one can compute bounded reachability sets symbolically for these kinds of systems.

One approach to symbolic representations is to express the constraint returned by method `errorFound(B)`, discussed earlier, through the satisfiability of a (huge) formula of propositional logic, and to then use a SAT solver to detect whether that formula is actually satisfiable (and thus to detect an error state reachable from initial states) [4]. This approach, as pointed out by the authors, is widely considered to be instrumental in leveraging the strength of bounded reachability analysis. But there has also been some work in the literature that demonstrated that alternative approaches, such as representing bounded reachability analysis through Binary Decision Diagrams (e.g. [5]), are a viable alternative to SAT-solver-based techniques in the verification of systems that are synchronous in nature.

The authors remark, though, that the current thinking is that such alternative techniques based on decision diagrams cannot be competitive with those based on SAT solvers in the verification of *asynchronous* systems. We hasten to add that the latter type of system is of growing importance, e.g., in the hardware industry where the paradigm shift to multi-core computing means that many hitherto synchronous systems are now in part asynchronous.

In this paper, the authors adapt the existing body of work on saturation [7] – an alternative to conventional breadth-first search – and on various forms of decision diagrams to demonstrate convincingly that techniques for bounded reachability analysis based on decision diagrams can indeed generally perform as well, and in some cases even better, than SAT-solver-based techniques. The experimental evidence that they supply is based on benchmarks for deadlock detection in Petri nets, and all benchmarks do contain deadlocks.

The authors overcome interesting technical challenges in transferring these existing techniques into their asynchronous setting. For example,

- they need to bound the symbolic traversal in the nested fixed-point comuputations of the usual Saturation algorithm
- they trade the avdantages of a more advanced iteration order in the Saturation algorithm over breadth-first-search approaches against the overhead resulting from more expensive symbolic data structures – demonstrating that this trade off is often effective in both time and memory consumption

– and they can ensure the termination of the Saturation algorithm that is adapted to a type of decision diagram that does not encode distance information.

As future work, the authors plan to research whether the Bounded-Saturation algorithm of their article can be applied or adapted to problems other than reachability checking, e.g. to model checking temporal logics that can express behavior of interest for such glocally asynchronous and locally synchronous systems. They also mean to investigate to what extent the event locality of such systems can be exploited by the SAT-solver-based approaches to bounded reachability analysis.

Acknowledgments

Daniel Kroening kindly pointed out to us that the approach in the paper *"An Abstraction-Based Decision Procedure for Bit-Vector Arithmetic"* has been adopted in a Synopsys tool.

References

1. T. Ball and S. K. Rajamani. The SLAM project: debugging system software via static analysis. In *Conference Record of the 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–3, Portland, Oregon, 16-18 January 2002. ACM Press.
2. M. Barnett, K. Rustan M. Leino, and W. Schulte. The Spec# programming system: An overview. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, volume 3362 of *Lecture Notes in Computer Science*, Springer, 2005.
3. S. Berezin, V. Ganesh, and D. Dill. A decision procedure for fixed-width bit-vectors. Technical report, Computer Science Department, Stanford University, April 2005.
4. A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. In *Proc. of the Fifth Int'l Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer Verlag, 1999.
5. R. E. Bryant. *Graph-Based Algorithms for Boolean Function Manipulation.* IEEE Transactions on Computers 35(8): 677–691, 1986.
6. L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. Rustan M. Leino, and E. Poll. *An overview of JML tools and applications.* Int'l Journal on Software Tools for Technology Transfer, 7(3):212-232, June 2005.
7. G. Ciardo, G. Luettgen, and R. Siminiceanu. Saturation: an efficient iteration strategy for symbolic state-space generation. In *Proc. of the Ninth Int'l Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 328-342, Springer-Verlag, 2001.
8. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Proc. of the 12th Int'l Conference on Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169, Berlin, Germany, 2000. Springer Verlag.
9. E. M. Clarke, O. Grumberg, and D. E. Long. *Model checking and abstraction.* ACM Transactions on Programming Languages and Systems, 16(5):1512–1542, 1994.
10. B. Cook, A. Podelski, and A. Rybalchenko. Abstraction Refinement for Termination. In *Proc. of the 12th Int'l Symposium on Static Analysis*, volume 3672 of *Lecture Notes in Computer Science*, pages 87–101, London, England, 7-9 September 2005. Springer Verlag.
11. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs. In *Proc. of the Fourth ACM Symp. on Principles of Programming Languages*, pages 238–252. ACM Press, 1977.
12. D. Dams. *Abstract interpretation and partition refinement for model checking.* PhD thesis, Technische Universiteit Eindhoven, The Netherlands, 1996.
13. D. Dams, R. Gerth, and O. Grumberg. *Abstract interpretation of reactive systems.* ACM TOPLAS, 19:253–291, 1997.
14. A. Dimovski, D. R. Ghica, and R. Lazić. Data-Abstraction Refinement: A Game Semantic Approach. In *Proc. of the 12th Int'l Symposium on Static Analysis*, volume 3672 of *Lecture Notes in Computer Science*, pages 102–117, London, England, 7-9 September 2005. Springer Verlag.
15. S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *Proc. of the 9th Intl. Conference on Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83, Haifa, Israel, June 1997. Springer Verlag.
16. D. Jackson. *Software Abstractions: Logic, Language, and Analysis.* The MIT Press, 2006.
17. D. Kroening and O. Strichman. *Decision Procedures: An Algorithmic Point of View.* Texts in Theoretical Computer Science. An EATCS Series, Springer Verlag, 2008.
18. G. T. Leavens and Y. Cheon. Design By Contract with JML. Tutorial paper available at `ftp://ftp.cs.iastate.edu/pub/leavens/JML/`, 2003.
19. B. Meyer. Design by Contract. In B. Meyer and D. Mandrioli, editors, *Proc. of Advances in Object- Oriented Software Engineering*, pages 1–50, Prentice-Hall, 1991.
20. R. P. Tan and S. H. Edwards. *Experiences evaluating the effectiveness of JML – JUnit testing.* ACM SIGSOFT Software Engineering Notes 29(5): 1–4, 2004.
21. G. Tseitin. On the complexity of proofs in propositional logics. In J. Siekmann and G. Wrightson, editors *Automation of Reasoning: Classical Papers in Computational Logic 1967–1970.* Volume 2., Springer-Verlag (1983), originally published in 1970.