Why Waste a Perfectly Good Abstraction?

Arie Gurfinkel and Marsha Chechik

Department of Computer Science, University of Toronto,
Toronto, ON M5S 3G4, Canada.
Email: {arie,chechik}@cs.toronto.edu

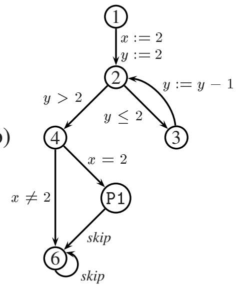
Abstract. Software model-checking based on the CEGAR framework can be made more precise by separating non-determinism from the lack of information due to abstraction. The two can be modeled individually using four-valued Belnap logic. In addition, this logic allows reasoning about negations effectively and thus enables checking of full CTL. In this paper, we present YASM – a new symbolic software model-checker. Preliminary experience with YASM shows that our implementation can effectively construct and analyze Belnap models without a substantial overhead when compared to its classical counterparts.

1 Introduction

Symbolic software model-checking, pioneered by the Microsoft's SLAM [1] project, is a technique that works directly on code and checks the program by combining automated *predicate abstraction* [13] with *counterexample-guided abstraction refinement* (CEGAR) [6]. The approach is divided into three phases: abstraction, model-checking, and refinement. During the abstraction phase, a theorem-prover is typically used to construct, using a list of predicates, a finite model that approximates the program being verified. The model is analyzed by the model-checker, and counterexamples generated by it are used to find additional predicates, if necessary. The process continues until either the property is successfully proved or disproved, or resources are exhausted.

For example, suppose our goal is to verify whether the line labelled `P1` can be reached in the (deterministic) C program shown in Figure 1(a). This can be expressed in CTL as $AG(pc \neq \texttt{P1})$. Figure 1(c)-(e) gives a series of predicate programs which are automatically constructed while checking this property. The abstraction in Figure 1(c) is just the control-flow graph of the program, where the symbol '$*$' indicates that the condition was abstracted away, and its value is not known. '$*$' is thus interpreted as "either true or false", and treated as a non-deterministic choice during model-checking. Verifying $AG(pc \neq \texttt{P1})$ on this abstraction yields false. It is possible to resolve non-determinism so as to reach the line labeled `P1`, i.e., by exiting the `while` and entering the `if` statement. We then check the feasibility of this execution in the concrete program, with the goal of replacing the undesired non-determinism. Specifically, a predicate $x = 2$ is needed to determine whether the control flow enters the `if` statement. The new abstraction is shown in Figure 1(d): $x = 2$ becomes true during initialization, is not affected by the body of the loop, and is checked in the condition of the `if` statement. Now, the analysis yields that the property is violated if the loop terminates, and the condition $y \leq 2$ of the loop is added to the list of predicates, yielding an abstraction

(a)
```
void main (void) {
1:  int x = 2;
    int y = 2;
2:  while (y <= 2)
3:     {y = y - 1;}
4:  if (x == 2)
5:     {P1:}
6:}
```

(b)


(c)
```
void main (void) {
  while (*)
    {}
  if (*)
    {P1:}
}
```

(d)
```
void main (void) {
  (x=2) := T;
  while (*)
    {(x=2) := (x=2);}
  if (x=2)
    {P1:}
}
```

(e)
```
void main (void) {
  (x=2) := T;
  (y <= 2) := T;
   while (y<=2)
     {(y<=2) := (y<=2)? T : *;
      (x=2) := (x=2);}
   if (x=2)
     {P1:}
}
```

Fig. 1. A simple C program (a), its control-flow graph (b) and its predicate abstractions: (c): no predicates; (d): after adding $x = 2$; (e): after adding $y \leq 2$.

(a)
```
int x;
x = 0;
if (x > 0)
  {x++}
else
  {x--}
P1:
```

(b)
```
if (*)
  {}
else
  {}
P1:
```

(c)
```
int x = 0;
if (fopen (...) != NULL)
{  if (x > 0)
     {x++;}
   else
    {x--}
   P1:
}
```

(d)
```
if (NONDET)
{  if (*)
     {}
   else
    {}
   P1:
}
```

Fig. 2. (a): a C program where line `P1` is not reachable; (b): abstraction of (a) without predicates; (c): a non-deterministic C program; (d) abstraction of (c).

in Figure 1(e). The statement $\mathtt{y} = \mathtt{y} - 1$ is abstracted as follows: if $y \leq 2$ is true, then decrementing y leaves it as true; otherwise, its value is unknown. The predicate $x = 2$ is not affected. The predicate program in Figure 1(e) is sufficient to determine that the loop does not terminate, and thus the property $AG(pc \neq \mathtt{P1})$ holds.

Now consider an example in Figure 2(a). Here, the property $\varphi = AG(pc \neq \mathtt{P1})$ fails in the concrete program. However, existing techniques (i.e., SLAM or BLAST) will not be able to determine this from the abstract program in Figure 2(b). To find a possible counterexample to φ, e.g., an execution which passes through the `else` part of the `if` statement, these techniques need to add another predicate, $x > 0$, and repeat the refinement and the model-checking phases. On the other hand, a human can easily determine that φ fails just by looking at the abstract program in Figure 2(b): line `P1` is reachable along *every* path, regardless of which of these are feasible. Thus, the abstraction in Figure 2(b) is conclusive for φ, and we will use it in our analysis. Specifically, given an abstraction and a property $AG(pc \neq x)$ for some line x, we first attempt to prove it directly, just like other approaches. If the proof fails, we then attempt to prove its negation, i.e., that $pc = x$ is *always* reachable, without considering which abstract executions are possible. If this proof fails as well, we gather additional predicates and proceed to the refinement phase.

So far, we have assumed that programs are deterministic. This assumption is unrealistic even for sequential programs, e.g., because of user input or other external factors, such as presence or absence of files that the program attempts to use. For example, consider the program in Figure 2(d) which abstracts the one in Figure 2(c). Here, the computation not leading to `P1` occurs when the file cannot be opened; since this behaviour is controlled by the environment, there exists a concrete execution leading to `P1`. Thus, we can conclude $AG(pc \neq \mathtt{P1})$ fails, without any further refinements or analysis of the feasibility of this execution.

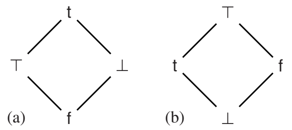


Fig. 3. Belnap logic: (a) truth order; (b) information order.

In this paper, we present an approach to proving truth and falsity of reachability properties. It is based on treating unknowns resulting from abstraction, '$*$', differently from unknowns resulting from the environment, non-determinism, as shown in the above example. Our approach is similar to the one taken by Reps and Sagiv [27] in the sense that it uses a logic with additional truth values (we use Belnap logic [4] which is an extension of Kleene logic [22] used in [27]) enabling us to perform both checks during a single analysis phase. The analysis yields one of the following answers: (1) the property holds; (2) the property fails (as in the model in Figure 2(d)); (3) the value of the property depends the resolution of '$*$', and thus the abstraction needs to be further refined.

We also present an implementation of this approach via a symbolic software model-checker YASM[1]. Although similar approaches have been studied theoretically, e.g. see [8, 12], we believe this to be the first efficient implementation with performance that is comparable to SLAM and BLAST. The implementation makes use of a number of ideas from existing CEGAR approaches, which we have generalized for our purposes. In particular, our implementation is applicable to programs with non-deterministic control-flow, and is not restricted to reachability analysis.

The rest of this paper is organized as follows. After giving the necessary background in Section 2, we describe, in Section 3, the process of creating and interpreting abstractions of programs we want to check. We discuss three abstract semantics: over-approximation, under-approximation and exact approximation, used in YASM. In Section 4, we describe model-checking of the models constructed via the exact approximation and the use of counterexamples for conclusiveness, generated by the model-checker, for computing refined abstractions. Exact approximations enable the use of effective techniques for improving the speed and the precision of the analysis. We discuss a few of them in Section 5. We describe the tool and give its performance data in Section 6, and conclude in Section 7 with a comparison of our approach with related work, a summary of the paper, and a discussion of future research directions.

2 Background

In this section, we review multi-valued logics and define multi-valued Kripke structures and their abstractions.

Logics. Boolean logic **2** is a set $\{\mathsf{t}, \mathsf{f}\}$ together with the truth ordering relation $\sqsubseteq$, s.t. $\mathsf{f} \sqsubseteq \mathsf{t}$. Conjunction $\wedge$ and disjunction $\vee$ represent meet and join with respect to the truth ordering. Additionally, a negation operator is defined as $\neg\mathsf{t} \triangleq \mathsf{f}$ and $\neg\mathsf{f} \triangleq \mathsf{t}$. Kleene logic [22] **3** extends **2** with an additional element $\bot$, representing "unknown" information. In this paper, $\bot$ is used to represent '$*$', discussed in Section 1. The truth ordering

[1] YASM stands for a **Y**et **A**nother **S**oftware **M**odel-checker.

of the logic is extended as $\mathsf{f} \sqsubseteq \bot$ and $\bot \sqsubseteq \mathsf{t}$, and negation as $\neg\bot = \bot$. We define an additional ordering $\preceq$, that relates values based on the amount of *information*; thus $\bot \preceq \mathsf{t}$ and $\bot \preceq \mathsf{f}$, so that $\bot$ represents the least amount of information. Belnap logic [4] $\mathbf{4}$ extends $\mathbf{3}$ with an additional element $\top$. The truth ordering is extended so that $\mathsf{f} \sqsubseteq \top$ and $\top \sqsubseteq \mathsf{t}$, and negation as $\neg\top = \top$, i.e., $\top$ is equivalent to $\bot$ with respect to this ordering. Finally, the information ordering is extended by making $\top$ be the largest element, i.e., $\mathsf{f} \preceq \top$ and $\mathsf{t} \preceq \top$. This makes $\mathbf{4}$ into the smallest structure containing $\mathbf{2}$ that is a complete distributive lattice under both truth and information orderings. The truth and information orderings of $\mathbf{4}$ are shown in Figure 3.

Temporal Logic. Temporal logic properties are specified in propositional μ-calculus $L_\mu(AP)$ [23]. Properties are often expressed in CTL, which is a subset of L_μ [7].

Definition 1. *Let* Var *be a set of variable names, and* AP *be a set of atomic propositions. The logic* $L_\mu(AP)$ *is the set of formulas defined as:*

$$\varphi ::= Z \mid p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \Diamond\varphi \mid \mu Z \cdot \varphi(Z)$$

where $p \in AP$, $Z \in \mathsf{Var}$, *and* $\varphi(Z)$ *is syntactically monotone in* Z.

We use the following syntactic abbreviations:

$$\varphi \vee \psi = \neg(\neg\varphi \wedge \neg\psi) \qquad \Box\varphi = \neg\Diamond\neg\varphi \qquad \nu Z \cdot \varphi(Z) = \neg\mu Z \cdot \neg\varphi(\neg Z)$$

The semantics of L_μ is defined with respect to an $\mathcal{L}$-valued Kripke structures.

Definition 2. *An* $\mathcal{L}$*-valued Kripke structure over a set of atomic propositions* AP *is a tuple* $K = \langle S, \mathcal{L}, R, I\rangle$, *where* S *is a set of states,* $\mathcal{L} \in \{\mathbf{2}, \mathbf{3}, \mathbf{4}\}$ *is a logic,* $R : S \times S \to \mathcal{L}$ *is a transition relation, and* $I : AP \to [S \to \mathcal{L}]$ *is an interpretation of atomic propositions that assigns to each atomic proposition a mapping from states to values in* $\mathcal{L}$.

We often refer to $\mathcal{L}$-valued Kripke structures simply as Kripke structures when $\mathcal{L}$ is irrelevant or clear from the context. For a transition relation $R : S \times S \to \mathcal{L}$, we define the *preimage* of $Q : S \to \mathcal{L}$ w.r.t. R, $\mathit{pre}[R] : [S \to \mathcal{L}] \to [S \to \mathcal{L}]$, as

$$\mathit{pre}[R](Q) \triangleq \lambda s \in S \cdot \bigvee_{t \in S} R(s,t) \wedge Q(t)$$

$\mathit{pre}[R](Q)$ is a set of states that have an R-successor in Q. A dual of *pre* is *wp*:

$$\mathit{wp}[R](Q) \triangleq \neg\mathit{pre}[R](\neg Q)$$

$\mathit{wp}[R](Q)$ is a set of states whose R-successors are all in Q.

The semantics of L_μ formula φ in a Kripke structure K, written $||\varphi||^K_\sigma$, is defined inductively on the structure of the formula, where $\sigma : \mathsf{Var} \to \mathcal{L}^S$ is an object assignment for free variables:

$$\begin{array}{llll}
||p||^K_\sigma & \triangleq I(p) & ||z||^K_\sigma & \triangleq \sigma(z) \\
||\varphi \wedge \psi||^K_\sigma & \triangleq ||\varphi||^K_\sigma \wedge ||\psi||^K_\sigma & ||\neg\varphi||^K_\sigma & \triangleq \neg||\varphi||^K_\sigma \\
||\mu x \cdot \varphi||^K_\sigma & \triangleq \mathrm{lfp}^{\sqsubseteq}\left(\lambda S \cdot ||\varphi||^K_{\sigma[x \mapsto S]}\right) & ||\Diamond\varphi||^K_\sigma & \triangleq \mathit{pre}[R](||\varphi||^K_\sigma)
\end{array}$$

where $\text{lfp}^{\sqsubseteq} f$ is the $\sqsubseteq$-least fixpoint of f. For a closed L_μ formula φ, $||\varphi||^K_\sigma = ||\varphi||^K_{\sigma'}$ for any σ and σ', written as $||\varphi||^K$. For a Kripke structure K and a state s, we write $K, s \models \varphi$ to mean $||\varphi||^K(s) = \mathsf{true}$. Note that when K is $\mathbf{4}$-valued, $K, s \not\models \varphi$ does not mean that $K, s \models \neg\varphi$, i.e., proving that φ is false is not the same as failing to prove that φ is true. Finally, we define $\Box L_\mu$ and $\Diamond L_\mu$ to be subsets of L_μ, where the modal operations are just $\Box$ and $\Diamond$, respectively, and negation is allowed only at the level of atomic propositions.

3 Program Abstraction

In this section, we show how programs are approximated by Boolean programs and present three approximation semantics.

3.1 Programs

Operations. Let V denote the set of program variables. A program is built out of operations Ops of which there are two kinds: (1) an assignment $l := e$, where l is a variable from V and e is an expression over program variables, and (2) an operation $\textit{assume}(e)$, where e is a boolean expression. Assume operations are used to model conditional branches. We also use an operation *skip* as a syntactic abbreviation for $\textit{assume}(\mathtt{t})$.

Programs as Control Flow Graphs. A *Control Flow Graph* CFG is a structure $G = \langle \textit{Loc}, \delta \rangle$, where *Loc* is a finite set of locations, and $\delta : \textit{Loc} \times \textit{Loc} \to \mathbf{2}$ is a transition relation. A program is modeled by a labeled CFG $\langle G, \tau \rangle$, where τ labels each edge of the CFG G with an operation from Ops. A CFG corresponding to the program in Figure 1(a) is shown in Figure 1(b).

Programs as Kripke Structures. A *state* is a type-correct valuation of all program variables. We use S to denote the set of all states, and $s(x)$ to denote the value of the variable x in s. Each operation *op* corresponds to a transition relation $\mathsf{S}(op)$ defined as:

$$\mathsf{S}(op)(s,t) \Leftrightarrow t = \begin{cases} s & \text{if } op \text{ is } \textit{assume}(e) \text{ and } s \models e \\ s[l \mapsto s.e] & \text{if } op \text{ is } l := e \end{cases}$$

Finally, a program $Prg = \langle G, \tau \rangle$ corresponds to a Kripke structure $K_{\textit{Prg}} \triangleq \langle \textit{Loc} \times S, \mathbf{2}, R_{\textit{Prg}}, I_{\textit{Prg}} \rangle$, where R and I are defined as:

$$\begin{aligned} R_{\textit{Prg}}(\langle l,s\rangle,\langle k,t\rangle) &\triangleq \delta(l,k) \wedge (\mathsf{S}(\tau(l,k)))(s,t) \\ I_{\textit{Prg}}(pc = j)(\langle l,s\rangle) &\triangleq (l = j) \\ I_{\textit{Prg}}(e)(\langle l,s\rangle) &\triangleq s \models e \end{aligned}$$

and e is a boolean expression.

The semantics of L_μ is extended to programs in the obvious way: a program *Prg* satisfies φ iff the corresponding Kripke structure $K_{\textit{Prg}}$ satisfies φ.

3.2 Boolean Programs

Boolean Operations. Let $P = \{p_1, \ldots, p_n\}$ be a set of quantifier-free first-order boolean predicates over program variables V. A Boolean (or Predicate) program [2] is a program constructed out of Boolean operations $BOps$. As before, the operations are divided into two kinds: (1) a parallel assignment $p_1 := e_1, \ldots, p_n := e_n$, and (2) an operation *assume*(e). We refer to elements of a parallel assignment as *updates*, e.g., $p_1 := e_1, p_2 := e_2$ consists of two updates, for predicates p_1 and p_2, respectively. The expressions on the right-hand-side of the assignment and in the argument of the assume operation are *partial boolean expressions* with the following grammar:

$$pb\_expr ::= * \mid \mathit{choice}(bool\_expr, bool\_expr) \mid \neg pb\_expr \mid bool\_expr$$

Intuitively, $*$ stands for an unknown expression, and *choice*(a, b) – for an expression that evaluates to true when a is true, to false when b is false, and whose value is unknown otherwise. In Boolean programs, we use *skip* as a syntactic abbreviation for a parallel assignment $p_1 := \mathit{choice}(p_1, \neg p_1), \ldots, p_n := \mathit{choice}(p_n, \neg p_n)$. As before, a Boolean program is a CFG whose edges are labeled with operations from $BOps$.

Syntactic Abstraction. We now show how a Boolean program *BPrg* is used to approximate a program *Prg* by describing the behavior of *Prg* using a finite set of predicates. We present the approximation in a bottom-up fashion, starting with approximation of expressions, and ending with approximation of programs.

A partial boolean expression pe approximates a boolean expression e (denoted as $pe \preceq e$), if (a) pe is a boolean expression logically equivalent to e, (b) pe is the $*$ expression, (c) pe is of the form *choice*(a, b) and a logically implies e, and b logically implies $\neg e$. For example, $y > 1$ is approximated by *choice*$(y > 0, \mathsf{f})$. Note that from the perspective of the approximation, $*$ is equivalent to *choice*(f, f).

The approximation is extended to the assume operations in the obvious way: *assume*$(pe) \preceq$ *assume*(e) iff $pe \preceq e$, e.g., *assume*$(y > 1)$ is approximated by *assume*$(\mathit{choice}(y > 0, \mathsf{f}))$. A single update $p{:=}\mathit{choice}(a, b)$ approximates an assignment $l{:=}e$ if *choice*(a, b) approximates the weakest pre-condition of the predicate p with respect to the assignment. In other words, a approximates the condition under which p becomes true after the assignment, and b approximates the condition under which p becomes false. For example, a program assignment $y{:=}y-1$ is approximated by $(y > 0){:=}\mathit{choice}(y > 0, \mathsf{f})$. Finally, a parallel assignment A approximates an assignment $l := e$ if all update operations of A approximate $l := e$. For example, $y := y - 1$ is approximated by $(y > 0) := \mathit{choice}(y > 0, \mathsf{f}), (x = 2) := \mathit{choice}(x = 2, \neg(x = 2))$.

We say that a Boolean program $\mathit{BPrg} = \langle G, \tau_B \rangle$ approximates a program $\mathit{Prg} = \langle G, \tau \rangle$ if each operation of *BPrg* approximates the corresponding operation of *Prg*. Since we have not yet given an operational semantics to Boolean programs, we call this approximation a *syntactic predicate abstraction*. There are standard techniques to compute such abstractions [1, 13].

3.3 Three Semantics of Boolean Programs

In order to evaluate temporal properties on Boolean programs, we must equip them with Kripke semantics. The only difficulty is to find a proper way to model the partial expres-

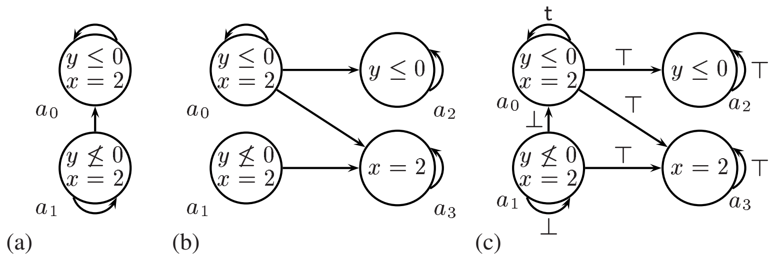


Fig. 4. Fragment of a transition relation: (a) over-approximation, (b) under-approximation, and (c) exact approximation.

sions, i.e., $*$ and *choice*(a, b). In this section, we describe three choices for this approximation: (a) an *over-approximating* semantics where "unknown" is modeled as a non-deterministic choice between true and false – this is the semantics used by most existing model-checkers such as SLAM [2] and BLAST [20]; (b) an *under-approximating* semantics where "unknown" is modeled by a partial assignment; and (c) the *exact* semantics that uses Belnap logic to combine over- and under-approximation – this is the semantics used by our model-checker YASM. The three semantics are illustrated on a parallel assignment $A : (y > 0) := \mathit{choice}(y > 0, \mathsf{f}), (x = 2) := \mathit{choice}(x = 2, \neg(x = 2))$ that approximates $y := y - 1$ using predicates $y > 0$ and $x = 2$.

Over-Approximation. In this case, a state is a boolean valuation of predicates, i.e., it is an element of $\mathbf{2}^P$. Each operation *bop* in BOp is associated with a transition relation $\mathsf{O}(\mathit{bop}) \subseteq \mathbf{2}^P \times \mathbf{2}^P$, such that abstract states a and b are not connected if we can conclude from the boolean operation that there is no transition between the corresponding states of the concrete program. That is, if the current state a does not satisfy the precondition for p to become false, a has a successor, b, where p is true.

Formally, the semantics of an update operation is

$$\mathsf{O}(p := \mathit{choice}(q, r))(a, b(p_i)) \Leftrightarrow (b(p_i) = \mathsf{t} \text{ and } a \not\models r) \text{ or } (b(p_i) = \mathsf{f} \text{ and } a \not\models q)$$

and semantics of a parallel assignment is the conjunction of all of its updates:

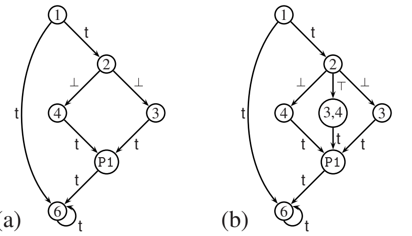
$$\mathsf{O}(\{p_i := \mathit{choice}(q_i, r_i)\}_{i=1}^n)(a, b) \Leftrightarrow \bigwedge_{i=1}^n (\mathsf{O}(p_i := \mathit{choice}(q_i, r_i))(a, b(p_i)))$$

For our running example, a part of a transition relation $\mathsf{O}(A)$ is shown in Figure 4(a). Note that a state a_1 corresponding to concrete states where $\neg(y \leq 2)$ and $x = 2$ has two outgoing transitions, to states a_0 and a_1, indicating that it is possible for $y \leq 2$ to non-deterministically become true or false in the next state. That is, the fact that the value of $y \leq 2$ is unknown in the next state is modeled by non-determinism.

Finally, the semantics of the assume operator is:

$$\mathsf{O}(\mathit{assume}(e))(a, b) \Leftrightarrow a \not\models \neg e \text{ and } \mathsf{O}(\mathit{skip})(a, b)$$

Under-Approximation. In this case, a state is a partial valuation of predicates, i.e, an element of $\mathbf{3}^P$, or a "tri-vector" [1]. Each operation $\mathit{bop} \in BOp$ is associated with a transition relation $\mathsf{U}(\mathit{op}) \subseteq \mathbf{3}^P \times \subseteq \mathbf{3}^P$, such that each predicate p is true in the next state, b, only if the current state, a, satisfies a precondition for p to become true.



(a) (b)

(c)

```
int x;
INIT:
x = 10;
P1:while(x > 0)
  x = x - 1;
END:
```

(d)

```
int x = 4;
if(x > 0){
  x=(int)(sqrt(x)+x^3);
  if(x < 212){
    x=(int)((3*x)/13);
    if(x*x > 100){
      ; }}}
END:
```

Fig. 5. (a) a Kripke structure corresponding to the Boolean program in Figure 2(d); (b) improving precision of `if`; (c) and (d) two example programs.

Formally, the semantics of an update operation is

$$\mathsf{U}(p := \mathit{choice}(q,r))(a, b(p_i)) \Leftrightarrow (b(p_i) = \mathsf{t} \text{ and } a \models q) \text{ or } (b(p_i) = \mathsf{f} \text{ and } a \models r) \text{ or } (b(p_i) = \bot)$$

and semantics of a parallel assignment is the conjunction of all of its updates:

$$\mathsf{U}(\{p_i := \mathit{choice}(q_i, r_i)\}_{i=1}^{n})(a,b) \Leftrightarrow \bigwedge_{i=1}^{n} \big(\mathsf{U}(p_i := \mathit{choice}(q_i, r_i))(a, b(i))\big)$$

For our running example, a part of a transition relation $\mathsf{U}(A)$ is shown in Figure 4(b). Here, state a_1 has a single outgoing transition to state a_3 indicating that in the next state the value of $y \leq 2$ is unknown, and $x = 2$ remains true.

Finally, the semantics of the assume operator is:

$$\mathsf{U}(\mathit{assume}(e))(a,b) \Leftrightarrow a \models e \text{ and } \mathsf{U}(\mathit{skip})(a,b)$$

Exact Approximation. This approximation combines the over- and under-approximating semantics in a single $\mathbf{4}$-valued model. Thus, the states are partial valuations of predicates, i.e., elements of $\mathbf{3}^P$. Each operation *bop* in BOp is associated with a $\mathbf{4}$-valued transition relation $\mathsf{E}(\mathit{bop}) : \mathbf{3}^P \times \mathbf{3}^P \to \mathbf{4}$. Intuitively, a transition is t if it appears both in the over- and the under-approximations, $\bot$ if it appears only in the over-approximation, and $\top$ if it appears only in the under-approximation. For our running example, a part of a transition relation $\mathsf{E}(A)$ is shown in Figure 4(c).

To define the semantics formally, we first introduce a function *eval*():

$$\mathit{eval}(\varphi, a) \triangleq \begin{cases} \mathsf{t} & \text{if } a \models \varphi \\ \mathsf{f} & \text{if } a \models \neg\varphi \\ \bot & \text{if } a \not\models \varphi \text{ and } a \not\models \neg\varphi \end{cases}$$

Then, the semantics of an update is

$$\mathsf{E}(p := \mathit{choice}(q,r))(a, v) = \begin{cases} \mathit{eval}(q,a) & \text{if } v = \mathsf{t} \\ \mathit{eval}(r,a) & \text{if } v = \mathsf{f} \\ \top & \text{if } v = \bot \end{cases}$$

and the semantics of a parallel assignment is the conjunction of all of its updates:

$$\mathsf{U}(\{p_i := \mathit{choice}(q_i, r_i)\}_{i=1}^{n})(a,b) \Leftrightarrow \bigwedge_{i=1}^{n} \big(\mathsf{U}(p_i := \mathit{choice}(q_i, r_i))(a, b(i))\big)$$

Finally, the semantics of the assume operation is:

$$\mathsf{E}(\mathit{assume}(e))(a,b) \triangleq \mathit{eval}(e,a) \wedge \mathsf{E}(\mathit{skip})(a,b)$$

The semantics of operations is extended to Boolean programs in an obvious way. Thus, each semantics associates a Boolean program with a Kripke structure. For example, the Kripke structure corresponding to the boolean program in Figure 2(d) is shown in Figure 5(a). The following theorem shows that over- and under-approximating semantics preserve universal and existential fragments of L_μ, respectively, whereas the exact semantics preserves full L_μ.

Theorem 1. *Let Prg and $BPrg$ be a program and its Boolean abstraction, respectively. Then, for an abstract state, a, and a corresponding concrete state, s, the following holds:*

$$\begin{array}{l} 1.\ \forall\varphi \in \Box L_\mu \cdot \mathsf{O}(BPrg), a \models \varphi \Rightarrow Prg, s \models \varphi \\ 2.\ \forall\varphi \in \Diamond L_\mu \cdot \mathsf{U}(BPrg), a \models \varphi \Rightarrow Prg, s \models \varphi \\ 3.\ \forall\varphi \in L_\mu \quad \cdot \mathsf{E}(BPrg), a \models \varphi \Rightarrow Prg, s \models \varphi \end{array}$$

The over-approximating semantics has been used in standard software model-checking tools to prove truth of AG properties. Our approach uses the exact semantics, which enables us to prove truth and falsity of such properties. We discuss it in the next section.

4 Abstract Model-Checking

To model-check a temporal logic formula φ in a state s and location l of a Boolean program *BPrg*, we use the techniques of Section 3 to construct a $\mathbf{4}$-valued Kripke structure K_{BPrg} and then compute the value of $||\varphi||_{K_{BPrg}}(\langle l, s\rangle)$. The latter step involves multi-valued model-checking, e.g., using the algorithm implemented in χChek [5]. In Section 4.1, we describe how to perform model-checking with Belnap logic using standard BDD packages.In Section 4.2, we show how to find additional predicates to refine the abstraction if the result of model-checking a formula φ is inconclusive.

4.1 Model-Checking

A symbolic multi-valued model-checking algorithm depends on an efficient representation and manipulation of Belnap functions, i.e., functions from some set S into $\mathbf{4}$. These functions are represented in YASM as BDDs using the ideas below. First, any set S can be encoded by r boolean variables $v_1, \ldots, v_r$, for a sufficiently large r. Thus, we only need to find a representation for Belnap functions whose domain is $\mathbf{2}^r$. Second, any Belnap function $f : \mathbf{2}^r \to \mathbf{4}$ can be represented by a pair of boolean functions $\langle f_\top, f_\bot\rangle$ over $\mathbf{2}^r$ [16], where $f_\top$ is $\lambda x \cdot f(x) \sqsupseteq \top$ and $f_\bot$ is $\lambda x \cdot f(x) \sqsupseteq \bot$. With this decomposition, $f(x)$ is equivalent to $(f_\top(x) \wedge \top) \vee (f_\bot(x) \wedge \bot)$. The following equivalences enable the direct computation of conjunction and disjunction of Belnap functions: (a) $f \wedge g = \langle f_\top \wedge g_\top, f_\bot \wedge g_\bot\rangle$; (b) $f \vee g = \langle f_\top \vee g_\top, f_\bot \vee g_\bot\rangle$; (c) $\neg f = \neg\langle f_\top, f_\bot\rangle = \langle \neg f_\bot, \neg f_\top\rangle$. Thus, we can represent each Belnap function by a pair of BDDs, one for each boolean function in the decomposition. Third, a pair of boolean functions $\langle f, g\rangle$ over variables $v_1, \ldots, v_r$ is represented by a single boolean function h with a new boolean variable z, using the encoding $h = z \wedge f \vee \neg z \wedge g$. So, Belnap functions can be represented and manipulated as standard BDDs at the expense of one additional variable. Furthermore, as many other symbolic model-checkers, we use the control-flow graph to partition the transition relation and its pre-image computation.

4.2 Abstraction Refinement

Whenever model-checking φ is inconclusive, our model-checker produces a behaviour of the system explaining why this is the case [15, 14]. So, we can use any of the existing techniques, e.g., [21], to determine whether this trace is feasible and use it to obtain additional predicates to refine the abstraction. However, in the multi-valued framework, checking feasibility of the trace is not necessary: we know exactly which part is inconclusive, and concentrate the refinement on it.

For example, consider the program in Figure 2(c), its abstraction in Figure 2(d), and the corresponding Kripke structure in Figure 5(a). Since the abstraction is built without any predicates, the Kripke structure is essentially equivalent to the CFG of the program. In this abstraction, $||EF(pc = \mathtt{P1})||(1)$ is inconclusive, i.e., $\perp$, which is exemplified by a path 1, 2, 4, P1 with an $\perp$-transition between states 2 and 4. In this example, the $\perp$-transition is the result of executing a boolean operation *assume*$(*)$ that syntactically abstracts *assume*$(y > 2)$ in the concrete program. Thus, the value of the predicate $y > 2$ is required to make the proof conclusive, which is done by refining the abstraction with this predicate, and repeating model-checking on the refined program.

In general, a path exemplifying why the result of model-checking is inconclusive always contains at least one $\perp$-transition. Suppose such a transition is between states $\langle k, a\rangle$ and $\langle l, b\rangle$, where k and l are program locations, and a and b are boolean valuations of predicates. By construction, this transition corresponds to a boolean operation *bop* such that $\mathsf{E}(bop)(a, b) = \perp$. If *bop* is a parallel assignment, then by the definition of the exact semantics there exists a predicate p and an update of the form $p :=$ *choice*(q, r) in *bop* such that $a \not\models q$ and $a \not\models r$. The idea is to refine the update, by strengthening the expressions q and r by the precondition for p to become true after execution of the concrete operation *op* corresponding to *bop*. This is done by refining the Boolean program with the predicate corresponding to the weakest precondition of p with respect to *op*, i.e., *wp*$[op](p)$.

For example, suppose we have model-checked a Boolean program with a single predicate $x > 0$, and the cause of inconclusiveness is a $\perp$-transition between states $a = \{(x > 0) \mapsto \mathtt{f}\}$ and $b = \{(x > 0) \mapsto \mathtt{t}\}$. Further, assume that the corresponding boolean operation is $(x > 0) :=$ *choice*$(x > 0, \mathtt{f})$, which is the result of abstracting the concrete operation $x := x + 1$. We can then refine the Boolean program by adding the predicate *wp*$[x := x + 1](x > 0) = (x > 1)$.

The above approach to abstraction-refinement is not limited to reachability properties. Our multi-valued model-checker can provide explanations to inconclusiveness of arbitrary CTL properties in the form of *proofs* [15], which we mine for additional predicates.

5 Exploiting Exact Approximations

The use of precise (Belnap) abstractions, described in Section 3, opens way to creating a number of techniques for improving the speed and the precision of the analysis. In this section, we discuss two of them.

Reusing Results of Previous Abstractions. One of the obvious limitations of the CEGAR framework is the fact that intermediate results are not shared between successive abstraction-refinement iterations. For example, consider applying the framework to check whether the property $EF(pc = \texttt{END})$ holds in location `INIT` in the program in Figure 5(c).

In the first iteration, we conclude that reachability of `END` depends on the value of $x > 0$, which is added to the list of predicates. During the model-checking phase of the second iteration, it is proved that `END` is reachable from `P1` if $x \leq 0$, i.e., $||EF(pc = \texttt{END})||(\langle \texttt{P1}, \langle \mathsf{f} \rangle \rangle)$ holds in the corresponding Kripke structure. Additionally, during the refinement phase, a new predicate $x > 1$ is added. The third iteration once again reproves that `END` is reachable from `P1` provided that $x \leq 0$ or $x \leq 1$, adding a predicate $x > 2$. The process continues, repeating the work done at the previous iterations, until all predicates of the form $x > i$, where $0 \leq i \leq 10$, are added, and termination of the loop is established.

To reuse results of previous computations, we must first identify which results are preserved between iterations. For a program *Prg*, let K_P be a Kripke structure abstracting *Prg* using predicates $P = \{p_1, \ldots, p_n\}$, constructed during an iteration i, and let $K_{P'}$ be a Kripke structure constructed during the $i+1$ iteration using predicates $P' = P \cup \{p_{n+1}\}$. From the construction of the abstractions, $||\varphi||_{K_P}(\langle l, u\rangle) \preceq ||\varphi||_{K_{P'}}(\langle l, v\rangle)$ for a formula φ and those states $\langle l, v\rangle$ of K_P and $\langle l, u\rangle$ of $K_{P'}$, where $v(i) = u(i)$ for all $i \leq n$ (i.e., v and u agree on the values of the first n predicates). In particular, if the concretization of v is not empty, then, if φ is either t or f in $\langle l, u\rangle$, it is correspondingly t or f in $\langle l, v\rangle$. This allows us to use $||\varphi||_{K_P}$, the result of model-checking φ on K_P, to help compute $||\varphi||_{K_{P'}}$ Formally, we define D_P as follows:

$$D_P(\langle l, \langle u_1, \ldots, u_{n+1}\rangle\rangle) = \begin{cases} \mathsf{t} & \text{if } ||\varphi||_{K_P}(\langle l, \langle u_1, \ldots, u_n\rangle\rangle) = \mathsf{t} \\ \mathsf{f} & \text{otherwise} \end{cases}$$

The function D_P is an under-approximation of $||\varphi||_{K_{P'}}$: for any state s of $K_{P'}$ with a non-empty concretization, $D_P(s) \sqsubseteq ||\varphi||_{K_{P'}}(s)$. If φ is computed using a least fixpoint, e.g., $\varphi = EF\psi$, D_P can be used as the starting point in computing φ on $K_{P'}$. For formulas that are computed using a greatest fixpoint, an over-approximation with respect to the truth ordering can be constructed and used in a similar manner. For a formula $\varphi = EF\psi$, the above optimization computes, at iteration $i+1$, reachability of states proved to satisfy $EF\psi$ at iteration i. In our example, this results in $EF(pc = \texttt{END})$ during the first and the second iteration, $EF(pc = \texttt{END} \vee (pc = \texttt{P1} \wedge x \leq 0))$ during the third, $EF(pc = \texttt{END} \vee (pc = \texttt{P1} \wedge x \leq 1))$ during the fourth, etc. In the standard approach, we would have been checking $EF(pc = \texttt{END})$ from scratch after each refinement.

Handling Conditional Statements. In this paper, every abstract state corresponds to a unique control flow location, which simplifies construction of the abstract model but limits its precision. Recall that the goal of checking our program in Figure 2(d) and its corresponding Kripke structure in Figure 5(a) was to establish whether the location $pc = \texttt{P1}$ is reachable. Intuitively, model-checking begins by labeling node P1 by t, i.e., P1 is reachable from itself, and then propagates this labeling along the edges of the Kripke structure. Thus, in the second iteration, nodes $pc = 4$ and $pc = 3$ are labeled by t, i.e., $pc = \texttt{P1}$ is definitely reachable from these nodes. In the third iteration, we run into a problem. We would like to conclude that $pc = \texttt{P1}$ is reachable from the node $pc = 2$ – it is reachable from both branches of the if-statement. However, according to our algorithm, the value at $pc = 2$ is obtained by (a) propagating the labeling of $pc = 3$ through the $(2, 3)$-edge, (b) propagating the labeling of $pc = 4$ through the $(2, 4)$-edge, and (c) taking a disjunction of (a) and (b). Thus, after the third iteration, the node $pc = 2$ is labeled with $(\bot \wedge \mathsf{t}) \vee (\bot \wedge \mathsf{t}) = \bot$, allowing us to conclude only that $pc = \texttt{P1}$ is $\bot$-reachable from $pc = 2$.

The cause of this problem is that in our abstract domain, we cannot express that one of the branches of the if-statements is taken, although we do not know which. One solution is to increase the abstract domain to include a state corresponding to several control flow locations. Figure 5(b) shows a possible abstraction with an additional abstract state $pc = (3, 4)$, corresponding to the set of program states in which the control location is either 3 or 4. It has a t-transition to $pc = \texttt{P1}$ since all states corresponding to it have a transition there, and has a $\top$-transition from $pc = 2$, indicating that the execution of the if-statement *definitely* results in the control passing to either location 2 or 3. In this case, after the second iteration of the model-checking algorithm, nodes P1, 3, 4, and $(3, 4)$ are labeled with t, and the third iteration results in the desired result: $(\bot \wedge \mathsf{t}) \vee (\top \wedge \mathsf{t}) \vee (\bot \wedge \mathsf{t}) = \mathsf{t}$.

Additional abstract states solve our problem; however, they can significantly increase the size of the abstract model and complicate the abstraction process. In YASM, we take a different approach, similar in spirit to "hyper-transitions" (e.g., [25, 28, 9]). Instead of increasing the abstract domain, we use the fact that for any concrete (**2**-valued) left-total transition relation R, $wp[R](Q) \subseteq pre[R](Q)$ – if all successors of a state s are in Q, then at least one successor is in Q. Thus, the pre-image computation of an abstract transition relation R_a can be augmented from $pre[R_a](Q)$ to $pre[R_a](Q) \vee (wp[R_a](Q) \succeq \mathsf{t})$, i.e., a state is assigned t if either it has a definite successor in Q, or all of its non-f successors are definitely in Q. In our example, this changes the third iteration of model-checking of the Kripke structure in Figure 5(a) as follows: in addition to computing *pre* along the edges $(2, 3)$ and $(2, 4)$, *wp* along each edge is computed to be $\neg\bot \vee \mathsf{t} = \mathsf{t}$, and the node $pc = 2$ is marked with t as desired. Thus, we can obtain a conclusive abstraction without the need to add the predicate $x > 0$.

Our approach also enables us to give definite results for certain programs with non-linear predicates. Consider the program in Figure 5(d). If the goal of a successful model-checking run is to *examplify* the path to END (as is the case in standard software model-checking approaches), the presence of complex mathematical operations will make the theorem-proving quite difficult, if not impossible. Our approach enables us to avoid these problems: we simply conclude that the path to END exists, whether it goes through

Name	LOC	YASM			Result	BLAST	
		Iterations	# of Pred	Time (sec)		Time (sec)	# of Pred
`tlan`	6885	4	3	15.4	t	52	9
`qpmouse`	4065	3	2	2.5	t	1	2
`qpmouse_err`	4065	12	20	5.7	f	–	–
`s3_srvr`	2261	3	30	2.9	t	16.6	15
`s3_srvr.3`	2240	3	38	25.1	t	152	20

Table 1. Experimental results.

the nested `if` statement or not. For such cases, we effectively perform context-sensitive slicing, removing parts of the abstraction which are not necessary to achieve a conclusive answer.

6 Experiments

The techniques described in this paper have been implemented in a software model-checker YASM. YASM is written in JAVA and uses theorem prover CVC Lite [3] to approximate program statements, and CUDD [29] library as a decision diagram engine.

Table 1 summarizes the performance of YASM on several programs based on the examples distributed with BLAST [20]. The experiments were performed on a Pentium 4 2.4 GHz machine running Linux 2.4.20. For each experiment, we list the number of iterations required by the abstraction-refinement phase, the final number of predicates, the overall model-checking time, and the final analysis result. For example, running YASM on a 4065-line `qpmouse` program took three iterations and yielded two predicates, in 2.5 seconds, whereas BLAST solved this problem in 1 second, also using two predicates. For every example, we checked whether an error condition is unreachable, which holds everywhere except `qpmouse_err`.

For these experiments, YASM was configured to prefer adding new predicates instead of computing a more precise abstraction. Our results clearly show that the running time of YASM is comparable to that of BLAST. We ran the latter as a baseline, to determine a reasonable performance for a software model-checker: a more direct comparison is not possible because the techniques used in the two model-checkers are significantly different. We do not report the results of running BLAST on `qpmouse_err` because the answer it gives when error is reachable is unsound: the paths reported by the tool are often infeasible.

7 Conclusion and Related Work

In this paper, we have presented YASM – a BDD-based software model-checker that combines automatic predicate abstraction-refinement with reasoning over Belnap logic. Our experience indicates that the CEGAR framework can be successfully extended to do proofs of reachability and proofs of unreachability, using the same abstraction. This approach allows us to shorten the abstraction-refinement cycle, is applicable to programs with non-deterministic control-flow, and provides support for model-checking of arbitrary CTL formulas.

There has been a lot of progress in applying automatic predicate abstraction of Graf and Saïdi [13] to software model-checking. The approach closest to ours is the one taken by SLAM [1] – YASM simply reinterprets SLAM's boolean programs using 4-valued semantics. Like our work, [26] makes a distinction between "non-deterministic" and "unknown" transitions and shows that performance of an explicit-state software model-checker is improved by guiding it towards the former. In our terminology, this would mean guiding the search to prefer non-$\perp$ transition, which happens automatically in our (symbolic) approach.

4-valued Kripke structures and their application to abstraction are equivalent to Mixed Transition Systems [8, 18]. They can also be seen as an extension of Modal Transition Systems [11] that are defined using Kleene logic.

We are not the first to use multi-valued logic to model abstraction in model-checking. Specifically, Kleene logic has been previously applied to reason about abstractions [27], and suggested as a basis for abstract model-checking [11]. Belnap logic has also been used to model abstraction in the context of (G)STEs [19] in a manner similar to ours.

Models based on Kleene logic have been used in [10] to separate handling of "unknown" and "non-determinism". Unlike our work or that of [8], it does not account for the relationship between abstract states, i.e., the case where $\gamma(a) \subseteq \gamma(b)$ for some abstract states a and b. It uses a (rather expensive [10, 17]) *generalized model-checking* approach and does not address the issue of generating counterexamples which are essential for an application of the CEGAR framework.

The current implementation of YASM is still limited. For instance, non-recursive functions are currently handled by in-lining; handling recursion requires a **4**-valued analog of Push Down Systems (PDS). We believe that Weighted PDSs [24] can provide such an analog, and are currently exploring their application to YASM.

Acknowledgments

We are grateful to Xin Ma, Kelvin Ku and Shiva Nejati for their help implementing, evaluating and improving YASM. We would like to acknowledge the financial support provided by NSERC. The first author has also been partially supported by an IBM Ph.D. Fellowship.

References

1. T. Ball, A. Podelski, and S. Rajamani. "Boolean and Cartesian Abstraction for Model Checking C Programs". *STTT*, 5(1):49–58, November 2003.
2. T. Ball and S. Rajamani. "The SLAM Toolkit". In *Proceedings of CAV'01*, volume 2102 of *LNCS*, pages 260–264, 2001.
3. C. Barrett and S. Berezin. "CVC Lite: A New Implementation of the Cooperating Validity Checker". In *Proceedings of CAV'04*, volume 3114 of *LNCS*, pages 515–518, 2004.
4. N.D. Belnap. "A Useful Four-Valued Logic". In Dunn and Epstein, editors, *Modern Uses of Multiple-Valued Logic*, pages 30–56. Reidel, 1977.
5. M. Chechik, B. Devereux, and A. Gurfinkel. "χChek: A Multi-Valued Model-Checker". In *Proceedings of CAV'02*, volume 2404 of *LNCS*, pages 505–509, July 2002.
6. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. "Counterexample-Guided Abstraction Refinement for Symbolic Model Checking". *Journal of the ACM*, 50(5):752–794, 2003.

7. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.
8. D. Dams, R. Gerth, and O. Grumberg. "Abstract Interpretation of Reactive Systems". *ACM TOPLAS*, 2(19):253–291, 1997.
9. L. de Alfaro, P. Godefroid, and R. Jagadeesan. "Three-Valued Abstractions of Games: Uncertainty, but with Precision". In *Proceedings of LICS'04*, pages 170–179, 2004.
10. P. Godefroid. "Reasoning about Abstract Open Systems with Generalized Module Checking". In *Proceedings of EMSOFT'2003*, volume 2855 of *LNCS*, pages 223–240, 2003.
11. P. Godefroid, M. Huth, and R. Jagadeesan. "Abstraction-based Model Checking using Modal Transition Systems". In *Proceedings of CONCUR'01*, volume 2154 of *LNCS*, pages 426–440, 2001.
12. P. Godefroid and R. Jagadeesan. "Automatic Abstraction Using Generalized Model-Checking". In *Proceedings of CAV'02*, volume 2404 of *LNCS*, pages 137–150, July 2002.
13. S. Graf and H. Saïdi. "Construction of Abstract State Graphs with PVS". In *Proceedings of CAV'97*, volume 1254 of *LNCS*, pages 72–83, 1997.
14. O. Grumberg, M. Lange, M. Leucker, and S. Shoham. "Don't Know in the μ-Calculus". In *Proceedings of CAV'05*, volume 3385 of *LNCS*, pages 233–249, 2005.
15. A. Gurfinkel and M. Chechik. "Generating Counterexamples for Multi-Valued Model-Checking". In *Proceedings of FME'03*, volume 2805 of *LNCS*, September 2003.
16. A. Gurfinkel and M. Chechik. "Multi-Valued Model-Checking via Classical Model-Checking". In *Proceedings of CONCUR'03*, volume 2761 of *LNCS*, pages 263–277, 2003.
17. A. Gurfinkel and M. Chechik. "How Thorough is Thorough Enough". In *Proceedings of CHARME'05*, volume 3725 of *LNCS*, October 2005.
18. A. Gurfinkel, O. Wei, and M. Chechik. "Logical Abstract Interpretation". Technical Report 532, University of Toronto, September 2005.
19. S. Hazelhurst and C. H. Seger. "Model Checking Lattices: Using and Reasoning about Information Orders for Abstraction". *Logic Journal of the IGPL*, 7(3):375–411, May 1999.
20. T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. "Lazy Abstraction". In *Proceedings of POPL'02*, pages 58–70, 2002.
21. T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. "Abstractions from Proofs". In *Proceedings of POPL'04*, pages 232–244, January 2004.
22. S. C. Kleene. *Introduction to Metamathematics*. New York: Van Nostrand, 1952.
23. D Kozen. "Results on the Propositional μ-calculus". *Theoretical Computer Science*, 27:334–354, 1983.
24. A. Lal, T. Reps, and G. Balakrishnan. "Extended Weighted Pushdown Systems". In *Proceedings of CAV'05*, volume 3576 of *LNCS*, pages 434–448, 2005.
25. K.G. Larsen and L. Xinxin. "Equation Solving Using Modal Transition Systems". In *Proceedings of LICS'90*, 1990.
26. C. Pasareanu, M. Dwyer, and W. Visser. "Finding Feasible Counter-examples when Model Checking Abstracted Java Programs". In *Proceedings of TACAS'03*, volume 2031 of *LNCS*, pages 284–298, 2003.
27. T.W. Reps, M. Sagiv, and R. Wilhelm. "Static Program Analysis via 3-Valued Logic". In *Proceedings of CAV'04*, volume 3114 of *LNCS*, pages 15–30, 2004.
28. S. Shoham and O. Grumberg. "Monotonic Abstraction-Refinement for CTL". In *Proceedings of TACAS'04*, volume 2988 of *LNCS*, 2004.
29. F. Somenzi. "CUDD: CU Decision Diagram Package Release", 2001.