Software Verification

Fault Tolerant Systems Research Group
Department of Measurement and Information Systems
Budapest University of Technology and Economics

2021

1 Introduction

Formal software verification techniques aim to provide mathematically precise proofs for the correct operation of computer programs. One of the most widely used method is *model checking* [1, 12], which exhaustively traverses every possible execution of the program for every possible input and checks whether certain *properties* are satisfied. There is a wide range of properties to be checked on a software, including failing assertions, indexing out of bounds, overflows, etc.

The advantage of the exhaustive behavior of model checking (as opposed to testing) is that it can prove both the presence and the *absence* of errors. However, a major drawback of model checking is its computationally expensive nature. Even if a program only takes three 32 bit integers as input, there are $2^{32} \cdot 2^{32} \cdot 2^{32}$ possibilities for their values, which cannot be exhaustively enumerated in practice. This is often referred to as the *"state space explosion"* problem. A wide variety of advanced model checking techniques have been proposed in the past decade to tackle this issue, including *symbolic* methods [8], *bounded* model checking [6] and *abstraction* [10, 11].

After introducing *control flow automata* [5], a formal representation for programs (Section 2), we focus on two variants of abstraction for model checking: *explicit value abstraction* [5] (Section 3.1) and *predicate abstraction* [15] (Section 3.2). Finally, we give a brief summary and suggest some materials for further reading (Section 4).

2 Control Flow Automata

Computer programs can be represented in different ways. The source code for example is a suitable format for people to read and write, whereas the compiled binary is ideal for execution. In order to perform model checking on software, the program code must be given in a mathematically precise, formal representation. A widely used formalism is the *control flow automata* (CFA) [5], which is a graph-based representation of a program.

Formally, a CFA is a tuple (V, L, l_0, E) with the following elements.

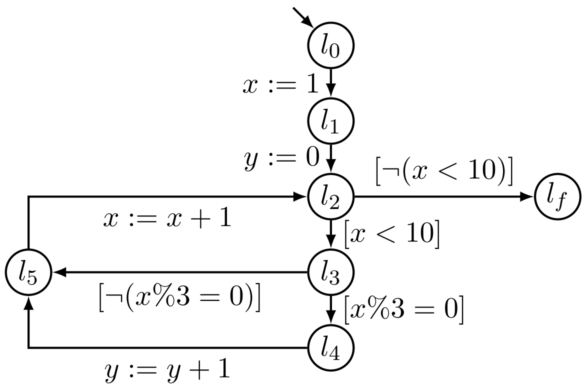
- $V = \{v_1, v_2, \ldots\}$ is the set of *variables* appearing in the program. Each variable $v_i \in V$ is associated with a domain D_{v_i}. In this document we focus on pure mathematical domains (e.g., Booleans and unbounded integers), but domains can also be architecture specific (e.g., 32 bit signed integers).
- $L = \{l_0, l_1, \ldots\}$ is the set of control *locations* modeling the actual position of the program counter.
- $l_0 \in L$ is the *initial location* representing the entry point of the program.
- $E \subseteq L \times Ops \times L$ is a set of directed *edges* between the location, annotated with *operations* over the variables that get executed when control flows from one location to another in the program. In this document we focus on simple programs with a single function, where operations are either *assignments* (e.g. $x := x + 1$) or *assumptions* (e.g., $[x > 0]$). Assignments and assumptions play a similar role as actions and guards in statechart models.

Locations and edges are denoted by circles and arrows respectively in the graphical representation of a CFA. The initial location is marked with an incoming arrow.

Example 2.1. *An example C program can be seen in Figure 1(a) with the corresponding CFA in Figure 1(b). The entry of the program, i.e., the initial location is l_0. The sequential statements in lines*

```
1 int x = 1;
2 int y = 0;
3 while (x < 10) {
4   if (x % 3 == 0) {
5     y++;
6   }
7   x++;
8 }
```

(a) Simple C program.



(b) CFA representation of the program.

Figure 1: Simple C program and its corresponding CFA.

1–2 are simply encoded by the path $l_0 \to l_1 \to l_2$. The head of the loop is l_2 where two options are possible depending on the loop condition. If the condition does not hold, the program jumps after the loop, which is the end of the program, usually denoted by a distinguished final location l_f. Otherwise, the program enters the body of the loop (l_3). The if statement in line 4 is encoded by the two outgoing edges from l_3. If the condition holds, the program moves to l_4, increments y (line 5) and proceeds to l_5 after the if statement. Otherwise, the program simply moves to l_5 since the if statement does not have an else branch. Finally, after l_5 the program increments x (line 7) and returns to the loop head (l_2).

As it can be seen in the previous example, the basic elements of structured programming (sequence, selection, repetition) can be represented in the CFA in the following way.

- *Sequential* statements are represented by a path (an alternating sequence of locations and edges).
- *Selections* (*if/then/else* statements) are represented by separate paths (with assumptions).
- *Repetitions* (loops) are represented by cycles. There is a location corresponding to the loop header with two outgoing edges: one into the body of the loop and one pointing after the loop. The loop body may contain further sequences, selections or repetitions, but their paths eventually return to the loop header location.

2.1 Assertions

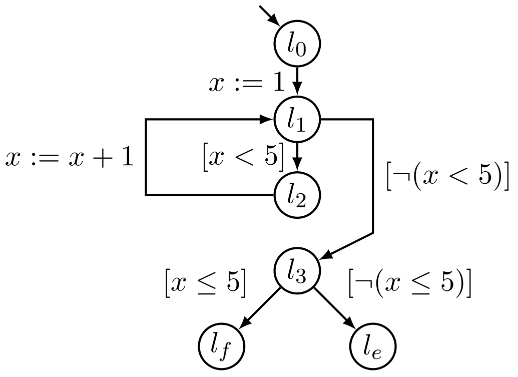
There is a wide variety of properties that can be verified on programs, including assertion failures, indexing out of bounds, arithmetic overflow, deadlock, null pointer dereference, violating pre- or post-conditions and so on. In this document we focus on verifying *assertions*, which check if a Boolean condition holds at a certain point of the program.

Assertions are usually represented in CFA as a special selection: if the condition holds, the program continues to the next location, otherwise it goes to a distinguished *error location* $l_e \in L$. If there are multiple assertions, they can use the same error location l_e to indicate the assertion failure. This way the task of formal verification is to check whether the given error location $l_e \in L$ is *reachable* in a CFA. Therefore, we call the pair of a CFA and an error location (CFA, l_e) a *verification task* [3].

Example 2.2. *Consider the C program in Figure 2(a), with an assertion in line 5. The corresponding CFA can be seen in Figure 2(b), where the condition of the assertion is evaluated at the outgoing edges of l_3. If the condition holds, the program continues to the next location, which is actually final location l_f, since the program ends after the assertion. Otherwise, the program proceeds to the distinguished error location l_e.*

```c
int x = 0;
while (x < 5) {
  x++;
}
assert x <= 5;
```

(a) Simple C program with an assertion.



(b) CFA representation of the program.

Figure 2: Simple C program with an assertion and its corresponding CFA. The assertion failure is represented by the distinguished error location l_e.

2.2 State Space

In order to formally verify the reachability of the error location $l_e \in L$ of a CFA (V, L, l_0, E), the model checking algorithm exhaustively explores all possible *states* and *transitions* (also called the *state space*).

The actual state of the program is described by two components: (1) the actual location $l \in L$, and (2) the actual values assigned to the variables $v \in V$. Hence, the actual state of the program is a tuple $(l, d_1, \ldots, d_n)$ where $l \in L$ is the actual location and $d_i \in D_{v_i}$ is a value assigned to variable v_i from its domain D_{v_i}. It can be seen that the number of possible states of a program is equal to the product of the number of locations and the sizes of the domains, i.e., $|L| \cdot |D_{v_1}| \cdot \ldots \cdot |D_{v_n}|$.

In the CFA formalism variables are not initialized at the beginning (i.e., they can have any value), hence all states are considered as *initial states* that have l_0 as their location. Note, that initial states (of the state space) should not be confused with the initial location (of the CFA). There is a single initial location (l_0) in a CFA, but due to the arbitrary values of variables, there can be many corresponding initial states $(l_0, \ldots)$ in the state space.

Example 2.3. *The program in Figure 1 has* 7 *locations and* 2 *variables. Supposing that the variables* x, y *are 32 bit integers, the number of possible states is* $7 \cdot 2^{32} \cdot 2^{32} \approx 10^{20}$*. For example,* $(l_2, 1, 0)$ *is a possible state where the program is at location* l_2 *and* $x = 1, y = 0$*. Furthermore all states* (l, x, y) *with* $l = l_0$ *and any* x, y *value are initial.*

Transitions in the state space of the program can be computed in the following way. Given an actual state $(l, d_1, \ldots, d_n)$, there is a potential transition for each outgoing edge $(l, op, l') \in E$ from l depending on the type of the operation *op*.

- If *op* is an assumption of the form [*cond*], where *cond* is a Boolean expression, then there is a transition to the successors state $(l', d_1, \ldots, d_n)$ if *cond* evaluates to true. In other words, if the condition of the assumption on the edge holds for the actual state, then there is a transition to the target location of the edge and the values of the variables remain the same.
- If *op* is an assignment of the form v_k := *expr*, where *expr* is an expression, then there is a transition to the target state $(l', d'_1, \ldots, d'_n)$ where $d'_i = d_i$, except for d'_k, which is equal to the result of evaluating *expr* with $d_1, \ldots, d_n$. In other words, the values of the actual state are substituted into the expression on the right side of the assignment, and the result is assigned to the variable on the left side. Other variables are left unchanged.

Example 2.4. *Consider the program in Figure 1. The state of this CFA can be described with a triple* (l, d_x, d_y) *where* l *is the actual location and* d_x, d_y *are the actual values of the variables* x, y*. Some example transitions are listed below.*

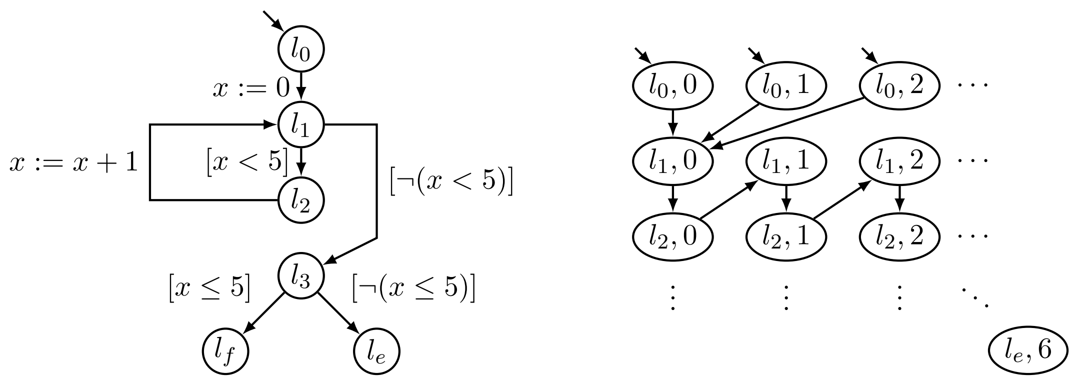
- $(l_4, 3, 0) \to (l_2, 3, 1)$ *is a transition since there is an edge from* l_4 *to* l_2 *with the operation* $y := y + 1$*.*

- *$(l_2, 1, 0) \rightarrow (l_3, 1, 0)$ is a transition since there is an edge from l_2 to l_3 with the assumption $[x < 10]$, which holds for the state $(l_2, 1, 0)$.*
- *Note however, that there is no transition from $(l_2, 1, 0)$ with the other edge $[\neg(x < 10)]$, since this condition does not hold.*

2.3 Model Checking

The purpose of software model checking is to determine whether a state with the error location l_e can be *reached*[1] starting from the initial state(s) of the CFA. A path leading to the error location is called a *counterexample* as it proves the incorrectness of the program. However, the previous examples demonstrated that the state space can be prohibitively (or even infinitely) large for even small programs, which makes explicit traversal of the state space infeasible in practice.

Example 2.5. *The CFA in Figure 3(a) has a single variable x, therefore states are described by pairs (l, d_x) of a location and the value of x. A part of the state space of the CFA can be seen in Figure 3(b). Since x is initially arbitrary, every state with l_0 is an initial state. However, since the only edge from*



(a) CFA. (b) State space.

Figure 3: Example CFA and its state space.

l_0 goes to l_1 and sets x to 0, the successor of each initial state will be $(l_1, 0)$. From l_1 there are two edges, but for $x = 0$ only the condition $[x < 5]$ holds, therefore the only transition goes from $(l_1, 0)$ to $(l_2, 0)$. From l_2 there is a single edge to l_1 updating $x := x + 1$, so the successor of $(l_2, 0)$ is $(l_1, 1)$. The example continues similarly, as it can be seen in Figure 3(b). Note, that there are states with l_e somewhere in the state space, for example $(l_e, 6)$ in the figure. If we continued the example, we would soon reach the final location l_f with no outgoing edges, and therefore we could conclude that l_e is not reachable. In real-life examples however, this question is often impossible to decide due to the large number of reachable states.

To overcome the problem of state space explosion, many advanced approaches have been developed in the past decades, including symbolic methods [8], bounded model checking [6] and abstraction [10, 11]. All of these methods try to reduce the size of the state space by bounding the search or using a compact representation.

3 Abstraction-Based Model Checking

The main reason for state space explosion is that variables in a program can have a great, or even an infinite amount of different values. Two of the most prominent abstraction based methods try to tackle this issue in the following ways.

- *Explicit value abstraction* [5] only tracks the values for a subset of the variables and also allows variables to have an *unknown* value (instead of enumerating all possibilities). Unknown values

[1]Model checking originally refers to checking whether temporal logic expressions (e.g., CTL or LTL) hold [9]. Checking reachability is a special case that can be expressed using temporal logic, but it is often also referred to as model checking.

mean that the variable can take any value from its domain. For example, if a CFA has variables x and y, we might only track x and treat y as unknown.

- *Predicate abstraction* [15] only tracks certain facts or relationships about the variables (called predicates). For example, instead of tracking the concrete values of x and y, we might only track whether $x > y$ holds or not.

Of course, by applying abstraction, we *lose information*, which might make the outcome of model checking incorrect. However, these methods are usually *over-approximations* [11], which means that there can be *false positives*, but no *false negatives*. In other words, if there are no counterexamples in the abstract state space (error location is not reachable), then there are no counterexamples in the original program (no false negative). However, it is possible that there is a counterexample in the abstract state space (due to the loss of information), but no counterexample in the original program (false positive). If the abstract counterexample turns out to be a false positive, then one must find a different abstraction, i.e., track other variables, or other predicates. This is called *counterexample-guided abstraction refinement* [5, 10, 16], but it is out of the scope of this document. The following sections introduce how explicit value and predicate abstractions work in a more detailed way.

3.1 Explicit Value Abstraction

Explicit value abstraction [5] tries to reduce the size of the state space by *tracking* only a subset $V_0 \subseteq V$ of the variables V. The subset V_0 is called the *set of explicitly tracked variables.*[2] Other variables are assigned an unknown value, denoted by $\top$ (“top”). Unknown values mean that the variable can take any value from its domain. Note, that it is also possible for a tracked variable in V_0 to have an unknown value if it is not initialized yet, or if an assignment cannot be evaluated. For example, if x is tracked, but y is not, the value of x will be unknown after an operation such as $x := y + 1$.

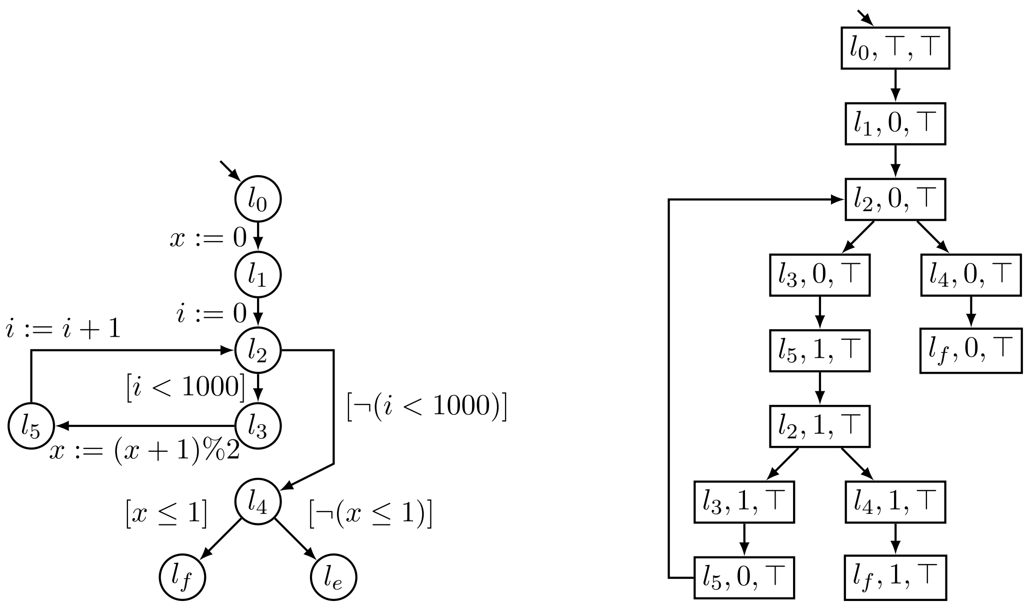
An abstract state in explicit value abstraction is a tuple $(l, d_1, \ldots, d_n)$, where some d_i values are possibly $\top$. Since $\top$ values represent all possibilities from the domain, a single abstract state can also represent multiple (or even infinite) number of states from the original CFA. For example, the abstract state $(l_0, 1, \top)$ represents states $(l_0, 1, 0), (l_0, 1, 1), (l_0, 1, 2), \ldots, (l_0, 1, n)$.

Given a CFA (V, L, l_0, E) and the subset $V_0 \subseteq V$ of explicitly tracked variables, the *abstract state space* is constructed in the following way. The only initial state is $(l_0, \top, \ldots, \top)$, since the CFA starts at l_0 and no variable is initialized. Then, given an actual state $(l, d_1, \ldots, d_n)$ (where some d_i values are possibly $\top$), there is a possible transition for each outgoing edge $(l, op, l') \in E$ from l depending on the type of the operation op.

- If op is an assumption of the form $[cond]$, where $cond$ is a Boolean expression, then there is a transition to the successor state $(l', d_1, \ldots, d_n)$ if $cond$ evaluates to true or if it cannot be evaluated (due to $\top$ values). For example, $[x > 0]$ cannot be evaluated if $x = \top$. Note, that treating unknown values this way yields an over-approximation. For example, if the condition of an *if/else* statement cannot be evaluated, we will explore both paths, avoiding false negatives (but possibly introducing false positives).
- If op is an assignment of the form $v_k := expr$, where $expr$ is an expression, then there is a transition to the successor state $(l', d'_1, \ldots, d'_n)$ where $d'_i = d_i$, except for d'_k, which is $\top$ if $expr$ cannot be evaluated (due to $\top$ values) or if $v_k \notin V_0$. Otherwise d'_k is equal to the result of evaluating $expr$ with $d_1, \ldots, d_n$. In other words, the variable on the left side of the assignment becomes unknown if the expression on the right side cannot be evaluated or if the variable is not tracked. Otherwise, the variable takes the result of the expression. Other variables are left unchanged.

Example 3.1. *Consider the CFA in Figure 4(a), which alternates x between 0 and 1 in a loop, and checks in the end whether $x \leq 1$. Although it can be seen that the error location cannot be reached (since x is only 0 or 1), a basic model checking algorithm would need to explore many states due to the*

[2]A similar approach was previously proposed for transition systems, where tracked and untracked variables are called “visible” and “invisible” respectively [13].



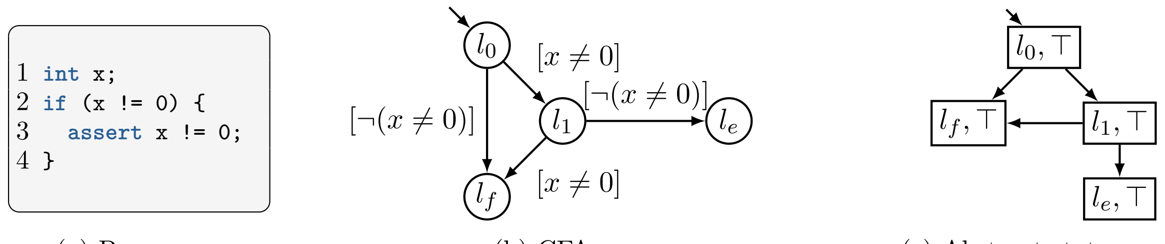
(a) CFA. (b) Abstract state space.

Figure 4: Example CFA and its state space using explicit value abstraction.

bound ($i < 1000$) of the loop. However, when explicit value analysis is used, and only x is tracked (i.e., $V_0 = \{x\}$), the abstract state space can be seen in Figure 4(b). Abstract states (denoted by rectangles) are triples of the form (l, d_x, d_i), but since i is not tracked, the third value is always $\top$.

The initial state is $(l_0, \top, \top)$ since the CFA starts at l_0 and no values are initialized. The first transition sets x to 0 and the second sets i to 0, but since i is not tracked, we arrive to state $(l_2, 0, \top)$. The loop condition $[i < 1000]$ cannot be evaluated (because $i = \top$), thus we explore both possibilities. If we do not enter the loop we arrive to $(l_4, 0, \top)$ where the assumption $[x \leq 1]$ holds so we proceed to the final location l_f with no outgoing edges. If we enter the loop we set x to 1 and arrive at the loop header $(l_2, 1, \top)$ again, while passing l_5. Note, however, that the operation of the edge $l_5 \to l_2$ has no effect on the state space as i is not tracked. At $(l_2, 1, \top)$ the condition cannot be evaluated again, hence we both take the path to $(l_4, 1, \top)$, leading to the final location and the path into the loop that sets x to 0 again. The latter path goes back to the state $(l_2, 0, \top)$, which was already explored. There are no more states to explore and the error location was not reached so we can conclude that it is not reachable in the original state space as well (due to over-approximation).

Example 3.2. *Consider now the program in Figure 5(a) with the corresponding CFA in Figure 5(b). It can be seen that the error location cannot be reached, since $[x \neq 0]$ and its negation cannot hold at the same time. Basic model checking cannot solve this task since x can take any value (for example*


```
int x;
if (x != 0) {
  assert x != 0;
}
```

(a) Program. (b) CFA. (c) Abstract state space.

Figure 5: Example CFA that cannot be checked with explicit value abstraction.

because it is an input). However, explicit value abstraction also fails even if x is tracked. The abstract state space can be seen in Figure 5(c). States are triples of the form (l, d_x), but since x is never assigned, it is always $\top$. The only initial state is $(l_0, \top)$, where the condition cannot be evaluated. One of the possibilities lead to the final location, but the other leads to $(l_1, \top)$. In the latter case, even *though we know that $x \neq 0$, we cannot represent this fact in explicit value abstraction: the value of x can be anything other than 0 so we must set x to the unknown value $\top$. However, this way the error location can be reached, since the condition of the assertion is also unknown (false positive).*

The previous examples demonstrated that while explicit value abstraction can reduce the state space effectively, it cannot solve certain kinds of tasks.

3.2 Predicate Abstraction

Predicate abstraction [15] tries to reduce the size of the state space by tracking a set of *predicates* (denoted by $P = \{p_1, p_2, \ldots, p_k\}$) instead of concrete values of variables. Predicates are Boolean formulas over V that capture facts or relationships about variables, for example $(x > 0)$, $(y + 2 = x)$, $(x\%2 = 0)$, and so on. An abstract state in predicate abstraction consists of a location (as usual) and of predicates or their negations, e.g., $(l_0, p_1, \neg p_3)$. It is also possible that neither a predicate p_i nor its negation $\neg p_i$ appears in a state if we cannot evaluate the predicate. For example, if the variable x is not initialized, the predicate $x < 0$ cannot be evaluated.

An abstract state represents all original states of the CFA for which the associated predicates hold. For example, if there is a single variable x and a predicate $(x\%2 = 0)$, then the abstract state $(l_0, x\%2 = 0)$ represents states $(l_0, 0), (l_0, 2), (l_0, 4), \ldots, (l_0, 2k)$. Similarly, the abstract state $(l_0, \neg(x\%2 = 0))$ with the negated predicate represents states $(l_0, 1), (l_0, 3), (l_0, 5), \ldots, (l_0, 2k + 1)$.

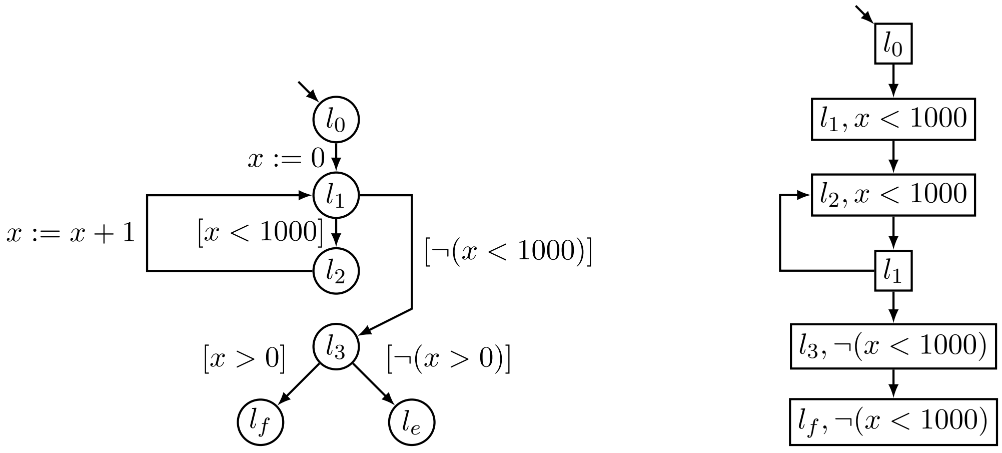
Given a CFA (V, L, l_0, E) and a set of predicates $P = \{p_1, p_2, \ldots, p_k\}$ to be tracked, the abstract state space is constructed in the following way. The only initial state is (l_0), since the CFA starts at l_0 and no variable is initialized, therefore we cannot evaluate any of the predicates. Let $\hat{p}_i$ denote the predicate p_i or its negation $\neg p_i$ or *true* (when the predicate is not present in a state). Then, given an actual state $(l, \hat{p}_1, \ldots, \hat{p}_k)$, there is a possible transition for each outgoing edge $(l, op, l') \in E$ from l depending on the type of the operation *op*.

- If *op* is an assumption of the form [*cond*], then we check if it *contradicts* the predicates of the actual state $\hat{p}_1, \ldots, \hat{p}_k$. If not, there is a transition to a successor state with location l'. The successor state will have those predicates (or their negations) from P that are *implied* by the predicates of the source state and the assumption. For example, a predicate $\neg(x < 0)$ on the source state and an assumption $[x \neq 0]$ together imply that the predicate $(x > 0)$ holds for the successor state.
- If *op* is an assignment of the form $v_k := expr$, then there is a transition to a successor state with location l'. The successor state will have those predicates (or their negations) from P that are *implied* by the predicates of the source state and the assignment. For example a predicate $(x > 0)$ on the source state and an assignment $x := x + 1$ together imply that the predicate $(x > 0)$ holds for the successor state.

In practice, contradictions and implications between formulas are calculated by *satisfiability modulo theory* (SMT) solvers [2, 7, 14].

Example 3.3. *Consider the CFA in Figure 6(a), which increments the variable x in a loop until 1000 and then checks if $x > 0$ holds. It can be seen that the error location is not reachable, but the basic model checking algorithm would need to explore many states due to the bound $(x < 1000)$ of the loop. The same holds for explicit value analysis, since x must be tracked, otherwise the condition of the edge between l_3 and l_e cannot be evaluated, leading to a false positive. However, using predicate abstraction with a single predicate $P = \{x < 1000\}$ yields the state space in Figure 6(b).*

The initial state is (l_0) since the CFA starts at l_0 and no values are initialized (so predicates cannot be evaluated). The operation of the edge $l_0 \to l_1$ sets x to 0, which implies that the predicate $x < 1000$ will hold for the successor state (at location l_1). At the state $(l_1, x < 1000)$ only the edge $l_1 \to l_2$ can be taken, since the operation of the other edge $(l_1 \to l_3)$ contradicts the predicate $x < 1000$. Therefore, the only successor is $(l_2, x < 1000)$, where the only edge $l_2 \to l_1$ increments x. Taking this edge to (l_1), we do not know if the predicate holds or not, since $x < 1000$ held previously, but incrementing x can



(a) CFA. (b) Abstract state space.

Figure 6: Example CFA and its state space using predicate abstraction.

make the predicate either hold or not. From (l_1) *we can now take both paths, from which* $l_1 \to l_2$ *will lead to the already explored state* $(l_2, x < 1000)$. *The other path leads to* l_3 *where we know that the negation of the predicate holds, due to the assumption on edge* $l_1 \to l_3$. *Now at* l_3 *we cannot go to the error location, since the assumption* $[\neg(x > 0)]$ *and the predicate* $\neg(x < 1000)$ *contradict each other. The other path is feasible, leading to the final location* l_f *and concluding that* l_e *cannot be reached.*

4 Summary

This document presented abstraction-based methods for formal software verification. A basic model checking approach often suffers from the large number of potential states and transitions. Explicit value abstraction can often reduce the state space efficiently, but might not be able to solve certain kind of tasks. Predicate abstraction can be an alternate solution, but it often requires more computational power due to checking implications and contradictions between predicates.

4.1 Further reading

A crucial point of abstraction-based methods is to find the proper abstraction, i.e., the appropriate variables or predicates to be tracked. It is out of scope for this document, but there are automatic algorithms for this purpose, usually referred to as “counterexample-guided abstraction refinement” [10, 16]. Such algorithms usually start with a coarse initial abstraction and apply refinements based on false positives (counterexamples). Refinement often relies on SMT solvers [2, 7, 14] and interpolation [5, 17, 19] that can automatically infer the set of variables or predicates to be tracked.

There are various tools implementing abstraction-based verification for software, including CPAchecker[3] [4] and Theta[4] [18], the latter being developed at our research group. There is also an annual competition on software verification [3], where tools compete on a large set of benchmarks.

References

[1] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking.* MIT Press, 2008.

[2] Clark W Barrett, Roberto Sebastiani, Sanjit A Seshia, and Cesare Tinelli. Satisfiability modulo theories. In *Handbook of satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, chapter 26, pages 825–885. IOS press, 2009.

[3] `http://cpachecker.sosy-lab.org`
[4] `http://github.com/FTSRG/theta`

[3] Dirk Beyer. Software verification with validation of results. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 10206 of *LNCS*, pages 331–349. Springer, 2017.

[4] Dirk Beyer and M. Erkan Keremoglu. CPAchecker: A tool for configurable software verification. In *Computer Aided Verification*, volume 6806 of *LNCS*, pages 184–190. Springer, 2011.

[5] Dirk Beyer and Stefan Löwe. Explicit-state software model checking based on CEGAR and interpolation. In *Fundamental Approaches to Software Engineering*, volume 7793 of *Lecture Notes in Computer Science*, pages 146–162. Springer, 2013.

[6] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer, 1999.

[7] Aaron R Bradley and Zohar Manna. *The calculus of computation: Decision procedures with applications to verification.* Springer, 2007.

[8] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and Lain-Jinn Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the 5th Annual IEEE Symposium on Logic in Computer Science*, pages 428–439. IEEE, 1990.

[9] Edmund M. Clarke. The birth of model checking. In *25 Years of Model Checking*, volume 5000 of *Lecture Notes in Computer Science*, pages 1–26. Springer, 2008.

[10] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, 50(5):752–794, 2003.

[11] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994.

[12] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model checking.* MIT Press, 1999.

[13] Edmund M. Clarke, Anubhav Gupta, and Ofer Strichman. SAT-based counterexample-guided abstraction refinement. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23(7):1113–1123, 2004.

[14] Leonardo De Moura and Nikolaj Bjørner. Satisfiability modulo theories: Introduction and applications. *Communications of the ACM*, 54(9):69–77, 2011.

[15] Susanne Graf and Hassen Saidi. Construction of abstract state graphs with PVS. In *Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer, 1997.

[16] Ákos Hajdu, Tamás Tóth, András Vörös, and István Majzik. A configurable CEGAR framework with interpolation-based refinements. In *Formal Techniques for Distributed Objects, Components, and Systems*, volume 9688 of *LNCS*, pages 158–174. Springer, 2016.

[17] K.L. McMillan. Applications of Craig interpolants in model checking. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 3440 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 2005.

[18] Tamás Tóth, Ákos Hajdu, András Vörös, Zoltán Micskei, and István Majzik. Theta: a framework for abstraction refinement-based model checking. In *Proceedings of the 17th Conference on Formal Methods in Computer-Aided Design*, pages 176–179. FMCAD inc., 2017.

[19] Yakir Vizel and Orna Grumberg. Interpolation-sequence based model checking. In *Formal Methods in Computer-Aided Design*, pages 1–8. IEEE, 2009.