Douglas Cunningham, Petra Hofstedt, Klaus Meer, Ingo Schmitt (Hrsg.): INFORMATIK 2015
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2015

# Building A State-Of-The-Art Model Checker[1]

Sebastian Wolff, Technische Universität Kaiserslautern

**Abstract:** Verification is the activity of proving a software artefact correct with respect to its specification. In this paper we focus on the algorithmic approach to verification. Therefore, we present how one can build a state-of-the-art model checker for recursive integer programs. Since those programs are present in different environments, many techniques evolved. Inspired by the driver verification with Microsoft's SLAM toolkit, we show how to integrate some of the available techniques into a single tool. One of those techniques is predicate abstraction. It allows us to handle one of today's core problems of model checking, namely a infinite data domain which is introduced by integer variables. Moreover, we integrate a reachability analysis to check the abstraction for correctness. This analysis uses procedure summaries to cope with recursive programs and potentially infinite call stacks. To complete our tool we also integrate a refinement for the abstraction based on Craig interpolation. Altogether, this stack of state-of-the-art techniques allows us to perform a CEGAR loop.

**Keywords:** Verification, Model Checking, Recursive Programs, Infinite Data Domain.

## 1 Introduction

Verification is a discipline of computer science which is dedicated to prove software artefacts correct. The question for correct software systems is important as those systems are widespread and deployed in a variety of electronic devices. Among those devices are lots of safety-critical ones, like controller units in cars and aircraft. Quite naturally, we want those systems to operate properly, that is, they should precisely meet their specification. As testing is expensive and not capable of proving correctness, we want to establish techniques that are able to automatically prove systems correct.

Model Checking is an algorithmic verification technique which meets our requirements from above. As it has been around for over 30 years, a lot of different approaches and procedures have been developed for a broad spectrum of systems and software. In this paper, we focus on the class of recursive programs with integer variables.

Recursive integer programs are, however, challenging as they are complex in two dimensions. On the one hand, they feature unbounded-depth recursion. On the other hand, they allow the usage of integer variables which have a potentially infinite domain. Even when restricting integers to 64 bit, as in modern processors, the search space is too big to be explored entirely [He04]. One widespread approach to tackle those challenges is the so called *Counter-Example Guided Abstraction Refinement (CEGAR)* loop [Cl00] depicted in Figure 1. The main idea is to discard unimportant aspects yielding a smaller abstract system. This abstract system is then steadily analysed and refined until it may be proven

[1] Author version, last updated on: October 22, 2015. Accepted for SKILL 2015.

correct or incorrect. Unfortunately, termination of the CEGAR approach is not guaranteed due to recursive programs with integer variables being Turing complete [Mi67].

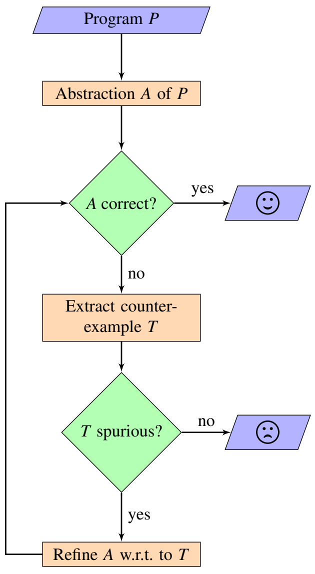

Fig. 1: The CEGAR loop.

A promising approach for generating a smaller system is *Predicate Abstraction*. Its working principle is to use a set of predicates to describe a subset of a system's behaviours. It provides multiple benefits for a model checker. Firstly, if conducted properly, it allows reasoning about the original program via the abstraction [JM09]. Secondly, the abstraction has a boolean, hence finite, data domain tackling one dimension of complexity.

The remaining dimension of complexity is introduced by recursion. To address this challenge, we employ a technique called *procedure summaries*. This approach computes the impact of calling a procedure on a system's state. By combining this technique with a reachability analysis, we are able to check finite systems, like our abstractions, for correctness.

Inspired by the Microsoft SLAM toolkit for static driver analysis, we implemented the CEGAR loop in our tool *RocketScience*[2]. Our contribution is to present how to integrate available state-of-the-art techniques into a functioning model checker. Therefore, we go along an execution of our tool. The input to our tool is a recursive integer program $P$ as described in Section 2. Then, a boolean abstraction $B(P)$ of the input program is generated via the predicate abstraction from Section 3. This new boolean program $B(P)$ is checked for correctness. Therefore, the program is translated into a control flow graph and a reachability analysis with procedure summaries is conducted as presented in Section 4. As a result of the reachability analysis, the program may be proven correct. If this is not the case, a counter-example $T$ is generated and checked for validity according to the procedure from Section 5. If $T$ reveals to be valid, it resembles a run of $P$, $P$ is shown to be incorrect. Otherwise, refinement, which is described in Section 6, is issued to remove the spurious counter-example from the abstraction. This completes the CEGAR loop and the procedure is repeated.

## 2 Recursive Programs

In the following, we give the definition of a simple programming language, which is inspired by curly braces languages, mainly C. For a formal definition consider Listing 1. The main features are recursive function calls, variables local to functions, global variables and integer variables with an unlimited domain. For the sake of simplicity, functions

[2] The source code is available at: `https://github.com/Wolff09/RocketScience`

```
Program     ::= VarDef* FunDef*
VarDef      ::= [int | bool] VarName ;
FunDef      ::= void FunName () { VarDef* Statement* }
Statement   ::= if ( BoolExpr ) { Statement* }
              | if ( BoolExpr ) { Statement* } else { Statement* }
              | while ( BoolExpr ) { Statement* }
              | FunName () ;
              | VarName [, VarName]* = Expr [, Expr]* ;
              | assert( BoolExpr ) ;
              | ; // skip
Expr        ::= BoolExpr | IntExpr | VarName | Literal
Literal     ::= true | false | <integer>
BoolExpr    ::= IntExpr [> | < | >= | <= | == | !=] IntExpr
              | BoolExpr [&& | ||] BoolExpr
              | BoolExpr ? BoolExpr : BoolExpr
              | ! BoolExpr
IntExpr     ::= IntExpr [+ | - | * | /] IntExpr
              | - IntExpr
```

List. 1: Language definition.

do not feature formal parameters and return values. This is no limitation to the expressibility of our language because communication between caller and callee can be handled via global variables. To that end, the caller writes the actual parameters to global variables and the callee copies them into local variables. Return values can be handled in the same way.

For the rest of the paper we assume that programs satisfy the following constrains: (a) expressions and statements are properly typed, (b) no global variable is shadowed by a local variable, (c) no variable appears more than once on the left hand side of an assignment, (d) the first statement of a function initializes all local variables, and (e) there is a `main` function which additionally initializes the global variables. These assumptions can be checked easily as they are of a static kind[3].

A *boolean* program is a regular program which is restricted to variables and literals of type `bool`. Additionally, boolean programs support `assume` statements which coincide with `assert` statements beside the fact that they never fail, i.e. simply block the control flow when their condition evaluates to *false*. Furthermore, we allow boolean programs to be non-deterministic by adding the literal *unknown*. These two additions to boolean programs compared to ordinary integer programs come in handy during the abstraction described in the next section.

## 3 Abstracting Integer Programs

A crucial part of model checking infinite state systems is an abstraction of the original integer program into a boolean program. A well-known approach is predicate abstraction

[3] Since our language is statically typed even (a) can be checked statically.

[GS97, JM09]. It divides the infinite state space of the integer program into finitely many equivalence classes. These classes are characterized by a set of predicates which are assertions about the states of the integer program.

In the following, an abstraction procedure is described which is due to Ball et al. [Ba01]. The input to the abstraction is an integer program $P$ and a set of predicates *Preds*. Each predicate $p \in \mathit{Preds}$ is a first order logic formula and comes with a scope. This scope is either *global* or a function. Naturally, global predicates may only range over global variables of $P$, whereas local predicates may additionally range over local variables of the corresponding function.

The output of the abstraction procedure is a boolean program, denoted by $B(P)$, which has a boolean variable $x_p$ for every predicate $p \in \mathit{Preds}$. The abstraction $B(P)$ imitates the behaviour of $P$ by updating its boolean variables in such a way that they capture how executing a statement from $P$ changes the truth of the predicates. Therefore, the original control flow is preserved and assignments to $x_p$, $p \in \mathit{Preds}$, replace the statements from $P$. An example abstraction can be found in Listing 2.

```
// program ExP
int x;
int y;

void main() {
  x, y = 5, 13;
  swap();
  assert(x > y);
}

void swap() {
  x = x + y;
  y = x - y;
  x = x - y;
}

// predicates (global)
p = x <= 5;
q = y >= 13;
```

```
// abstraction B(ExP) w.r.t. {p, q}
int p;
int q;

void main() {
  p, q = true, true;
  swap();
  if (unknown) { assume(!p || !q); }
  else { assume(true); assert(false); }
}

void swap() {
  p = !p && q ? false
              : unknown; // act1
  q =  p && q ? false
              : unknown; // act2
  p =  p && q ? true
              : unknown; // act3
}
```

List. 2: Example program and its abstraction.

**Weakest Preconditions** Central to the abstraction are weakest preconditions. The weakest precondition $wp(s, \varphi)$, for some statement $s$ and some first order formula $\varphi$, is the weakest predicate the truth of which before $s$ entails the truth of $\varphi$ after $s$. The weakest precondition for an assignment $x = e$ is thereby defined as $wp(x = e, \varphi) = \varphi[x \mapsto e]$ where $\varphi[x \mapsto e]$ equals $\varphi$ with every occurrence of $x$ replaced with $e$.

We also define a strengthening $F(\varphi)$ of $\varphi$. It is the weakest formula implying $\varphi$ and ranging over the set $\{x_p \,:\, p \in \mathit{Preds}\}$. Furthermore, a weakening $G(\varphi)$ will be useful and is defined by $G(\varphi) = \neg F(\neg\varphi)$. We employ both those notions as they take formulas ranging over predicates and produce formulas ranging over the corresponding variables from the abstract program. The intuition behind this is an abstraction from formulas from the integer program to formulas in the abstract program. These basic building blocks of our abstraction procedure are effectively computable and an optimized implementation is given in [Ba01].

**Conditionals** Given an `if(c){...}else{...}` construct from program $P$, we know at the beginning of the then-branch that $c$ holds. Thus, in the abstract program, we want the then-branch to be executed only if the abstraction does not imply $\neg c$. Remember that we already introduced this notion with $G(c)$ and hence come up with the following rule [Ba01].

```
if (c) {
  ...
} else {
  ...
}
```

$\xmapsto{\text{abstraction}}$

```
if (unknown) {
  assume( G(c) );
  ...
} else {
  assume( G(!c) );
  ...
}
```

Note here, that we introduce a non-deterministic choice which is guarded with assume statements. We utilize the non-determinism as both $G(c)$ and $G(\neg c)$ might hold in the abstract program [Ba01]. Additionally, this construct allows to explore both branches in the following reachability analysis. The same approach applies to while loops as follows.

```
while (c) {
  ...
}
```

$\xmapsto{\text{abstraction}}$

```
while (unknown) {
  assume( G(c) );
  ...
}
assume( G(!c) );
```

**Assignments** Consider an assignment $x = e$ in $P$. This statement may influence the truth of a whole range of predicates, namely those containing $x$. So the abstraction of the assignment is an assignment again which captures the impact of $x = e$ on all predicates. Therefore, consider a predicate $p_i$ which is modelled by the boolean variable $b_i$. Then, by definition, $p_i$ is *true* after $x = e$ if $wp(x = e, p_i)$ can be shown to evaluate to *true* under every possible assignment to the free variables. That is, $b_i = \mathit{true}$ is a valid assignment if $F(wp(x = e, p_i))$ holds. Analogously, $b_i = \mathit{false}$ is valid if $F(wp(x = e, \neg p_i))$ holds. However, the predicates might be to weak to prove any assignment valid. Naturally, we assign $b_i = \mathit{unknown}$ in such a case.

According to [Ba01], there is always at most one valid assignment to $b$. By exploiting this fact, one can come up with the following rule [Ba01].

```
x = e;
```

$\xmapsto{\text{abstraction}}$

```
b1, ..., bn =
  F(wp(x=e,p1)) ? true
                : F(wp(x=e,!p1)) ? false : unknown,
  ...,
  F(wp(x=e,pn)) ? true
                : F(wp(x=e,!pn)) ? false : unknown;
```

**Asserts** An `assert(`$c$`)` is supposed to have no effect if the boolean condition $c$ evaluates to *true* and should fail otherwise. Hence, we handle this statement in the same way as an `if`. We come up with the following rule.

```
                                                   if (unknown) {
                            if (c) {                 assume( G(c) );
                              // skip               } else {
assert(c);  --translation-> } else {   --abstraction->  assume( G(!c) );
                              assert(false);          assert(false);
                            }                       }
```

Note here, that we introduced an `assert(false)` to identify an assertion error in the abstract program. The abstraction, however, is not recursively continued for the newly introduced assertion.

**Calls** Function calls are simply copied to the abstraction since they have neither formal parameters nor return values. As functions communicate via global variables, the communication can be observed via global predicates and tracked throughout the entire program. Hence, our approach is sufficient for handling simple function calls.

## 4 Checking Boolean Programs

The next step in the CEGAR loop, after abstracting the input program, is to check the abstraction for correctness. That is, in our context, to check whether there is an execution of the abstracted program which raises an assertion error. According to our abstraction procedure, assertions in the original program are translated into an `if` construct where only the failing branch contains an `assert(false)`. Thus, we only need to check whether any `assert` statement is reachable. If so, the abstraction is considered incorrect.

The above reachability problem can be solved algorithmically on a graph rather than directly on the program code level. To that end, we first introduce a translation from code to control flow graph and then conduct a reachability analysis on the resulting graph. Note that, although the data domain is finite due to the previous abstraction, the reachability analysis still needs to tackle the challenge of arbitrary large call stacks and possibly non-terminating recursion.

**Control Flow Graphs** A control flow graph $G$ is a finite directed graph $G = (S, V, T, C)$ with a set of nodes $S$, a set of boolean variables $V$, a set of edges $T$ and another dedicated set of edges $C$ which are used for function calls only. The set of variables is split into local and global variables, i.e $V = \mathit{Locals} \uplus \mathit{Globals}$. $T$-edges additionally come with a *guard* and a set of *actions*. The guard is a first order logic formula and an action has the form $x = e$ with $x \in V$ and a boolean term $e$ denoting the new value of $x$. For a $T$-edge from $s$ to $s'$ with guard $g$ and actions $x_1 = e_1, \ldots, x_n = e_n$ we write

$$s \xrightarrow{g/x_1=e_1,\ldots,x_n=e_n}_T s'.$$

A configuration is a tuple $cf = (s, val)$ with state $s \in S$ and $val : V \mapsto \{true, false, *\}$ being a valuation to the variables in $V$. The valuation $val$ may map some variables to $*$ denoting an arbitrary truth value. If such a mapping exists we call the valuation *partial* and *complete* otherwise. The evaluation of a formula $\varphi$ based on a valuation $val$ is denoted by $||\varphi||_{val}$.

Consider some transition $s \xrightarrow{g/x_1=e_1,\ldots,x_n=e_n}_T s'$ and some configuration $cf = (s, val)$. The transition can be taken by $cf$ if $g$ is enabled, i.e. if $||g||_{val} = true$. The result of taking the transition is a new configuration $cf' = (s', val')$ with

$$val'(x) := \begin{cases} ||e_i||_{val}, & \text{if } x \in \{x_1, \ldots, x_n\} \\ val(x), & \text{otherwise.} \end{cases}$$

**From Boolean Programs to Control Flow Graphs** Next, we present the translation procedure. Therefore, consider a boolean program $B(P)$ and an empty control flow graph $G$ which is extended as we process $B(P)$ and used as output of our procedure.

Firstly, we generate a skeleton for every function from $B(P)$. That is, for every function we add two nodes to $G$: an *entry* and an *exit* node both of which are unique. Those skeletons are required to translate recursion properly since a function might invoke itself or another function which has not been (completely) translated yet. Given some function $f$, we may reference those nodes as *f.entry* and *f.exit*, respectively. Secondly, we add a dedicated *error* node which we use to identify assertion errors. Lastly, we add sufficiently many variables to $G$ such that every variable $x$ from $B(P)$ can be mapped to a variable $x_G \in V$. Hence, an expression $e$ from $B(P)$ can be translated by simply replacing every variable $x$ occurring in $e$ with its corresponding graph variable $x_G$.

With this basic structure set up, the actual translation of the function bodies from $B(P)$ is straight forward. Figure 2 presents the required translation rules. The only statement that needs some special treatment is the function call. A call to function $f$ in $B(P)$ introduces two new nodes in $G$ – the *call* node and the *return* node. Additionally, we add a $C$ edge between the call node and *f.entry* as depicted in Figure 2c. Using the dedicated *call* transition relation $C$ allows us to differentiate between sequences of statements and function calls when performing a reachability analysis. This will ultimately allow us to restore local variables in unbounded-depth recursion during the later reachability analysis.

**Reachability Analysis with Procedure Summaries** Given the control flow graph $G$ resulting from a translation of a boolean program $B(P)$, it remains to conduct a reachability analysis to check whether an assertion error might occur. By construction, we simply need to check whether the dedicated *error* state is reachable in $G$.

For finite transition systems, the set of reachable configurations can be effectively computed as a fixed point to the equation $x = x \cup post_T(x)$ [Sc04]. This approach is, however, insufficient for our purpose as the post image does not handle local variables properly. Naturally, a function call should not alter the local variables. But if we would simply compute post images for the *call* transitions, we could not restore local variables for returning

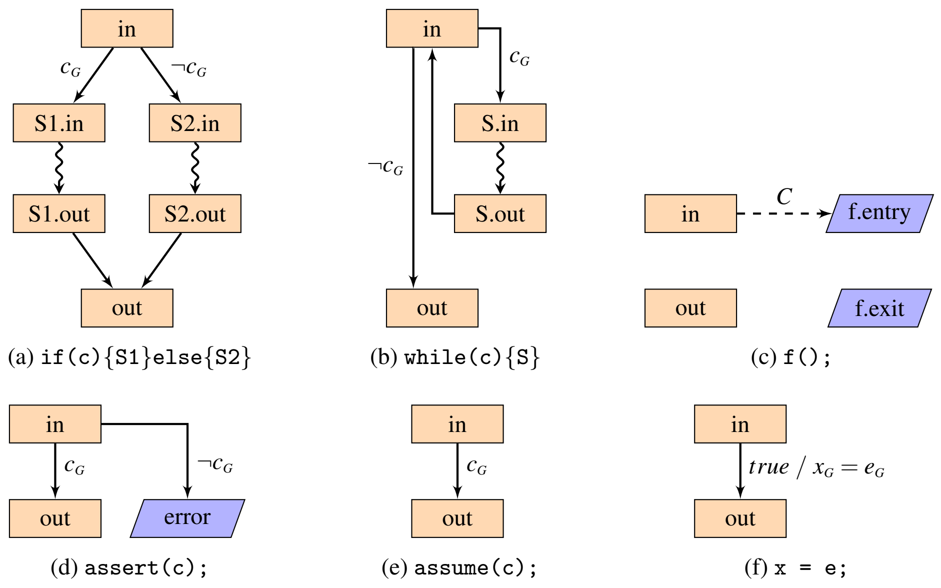

Fig. 2: Translation rules to turn a boolean program into a control flow graph.

recursive calls[4]. Hence, we apply a technique called *procedure summaries* [BR00]. It basically executes a separate *sub*-analysis and as a result augments the control flow graph with an additional *T*-edge which summarises the effect of the function call on the global variables. For an example procedure summary consider Figure 3.

Example path of configurations (states omitted) through the abstract version of `swap()` from Listing 2:

$$\begin{Bmatrix} p \mapsto 1 \\ q \mapsto 1 \end{Bmatrix} \xrightarrow{act1} \begin{Bmatrix} p \mapsto * \\ q \mapsto 1 \end{Bmatrix} \xrightarrow{act2} \begin{Bmatrix} p \mapsto 1 \\ q \mapsto 0 \end{Bmatrix} \xrightarrow{act3} \begin{Bmatrix} p \mapsto * \\ q \mapsto 0 \end{Bmatrix}$$

Resulting summary edge:

$$call \xrightarrow{\substack{p\leftrightarrow 1 \wedge q \leftrightarrow 1\,/ \\ p=*,\, q=0}}_T return$$

Fig. 3: Procedure summary example.

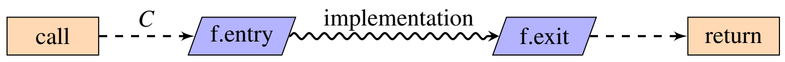

Fig. 4: Call scenario.

A formal description is in order. Therefore, consider a configuration $cf_0 = (call, val_0)$ with respect to Figure 4. Furthermore, assume that *val* is complete. Then, compute the post image of $cf_0$ relative to the *call* transition relation $C$ and quantify out all local variables. This gives a new (partial) configuration $cf_1 = (f.entry, val_1)$, with $val_1(g) = val_0(g)$ for global variables $g$ and $val_1(l) = *$ for local variables $l$. Next, conduct a reachability analysis relative to $T$, as described above, yielding a set of reachable configurations $CF$. If there is

[4] Memorizing the local variables of the call site is not possible as the stack of recursive functions might grow beyond all bounds and our control flow graph is required to be finite.

some $cf \in CF$ with $cf = (f.exit, val)$, we can add a *summary edge* to $G$ describing the effect of $f$ to the global variables. Hence, we augment $T$ by adding the edge $call \xrightarrow{g/a}_T return$ with

$$g = \bigwedge_{x \in Globals} x \leftrightarrow val_0(x) \qquad \text{and} \qquad a = \{x = \|x\|_{val} \,:\, x \in Globals\}.$$

Additionally, we might need to recursively invoke this procedure when a new call site is found, i.e. if there is some $cf \in CF$ where the state of $cf$ describes some function's *call* block and no summary has been computed yet. However, when recursively descenting, one must prevent repetitions. That is, when computing the summary for $cf$, no sub analysis must be issued for $cf$ (again). This is because a reoccurring configuration in the recursive call stack resembles a non-terminating function call in the program. Hence, no summary is computed for such a call. Furthermore, this guarantees termination of our procedure.

Our overall approach for model checking boolean programs simply interleaves both above techniques exhaustively. That is, we compute the fixed point for $x = x \cup post_T(x)$, conduct procedure summaries for all call sites, and repeat this until the set of configurations saturates. Lastly, it remains to check membership of the *error* state.

## 5 Counter-Example Traces

Applying the procedure described in the previous section might reveal that the abstract boolean program $B(P)$ can run into an assertion error by showing that the dedicated *error* state of the control flow graph associated with $B(P)$ is reachable. In that case, we have to check if the malicious behaviour is also present in the original integer program $P$. Therefore, we have to compute a so called *counter-example trace* and check whether this trace is valid. A *counter-example trace* is thereby a sequence of statements from $P$ which ultimately runs into an assertion error. Intuitively, it is a linearisation of $P$ which does not contain control structures like *if* and *while* statements. Lastly, checking the validity of the trace means to check whether or not it is a valid execution of $P$. If so, $P$ is proven incorrect. Otherwise the abstraction $B(P)$ is too imprecise as it allows malicious – so called *spurious* – behaviours which are not present in $P$.

**Generating Traces** In the following we give a description of a method for generating a counter-example trace based on the reachability analysis from Section 4. Therefore, consider the control flow graph $G$ corresponding to $B(P)$ and the set of reachable configurations $CF$ which results from the above mentioned analysis. A counter-example trace is basically a lifting of a path through $G$ to a sequence of statements from $P$. Hence, we first need to compute a path through $G$. Therefore, consider some configurations $cf_0$ and $cf_n$. A path from $cf_0$ to $cf_n$ through $G$ consisting only of configurations from $CF$ can be found by a wavefront-like approach. We iteratively compute the sets $step_k$ of configurations which can reach $cf_n$ in exactly $k$ steps. Those sets can be defined recursively with $step_0 = \{cf_n\}$ and $step_{k+1} = pre_{T \cup C}(step_k) \cap CF$. This sequence of sets is extended until eventually some set $step_n$ contains $cf_0$. Then, a path from $cf_0$ to $cf_n$ through $G$ is given by $\pi = \pi_0 \dots \pi_n$ with

$$\pi_0 = cf_0 \qquad \textit{and} \qquad \pi_{i+1} = pickone(\, post_{T \cup C}(\pi_i) \cap step_{n-i-1}\,)$$

where *pickone* chooses some arbitrary complete configuration from a given set[5].

Given such a path $\pi$ we can lift it to a sequence of statements from $B(P)$ by simply backtracking the translation process from Section 4. From the resulting sequence, we can generate the desired counter-example trace by, again, backtracking the abstraction process from Section 3.

Since the computed path may contain *summary edges* the resulting trace can contain function calls. As a last step, we *flatten* the trace from above and replace every function call `f();` with `f();` $\tau_{sub}$ `return;` where $\tau_{sub}$ is a recursively computed flat trace for `f`. Finally note, when recursively descending one must not use a summary edge twice as this indicates non-termination (cf. Section 4). Listing 3 continues the example from above and gives an example trace.

```
// trace for B(ExP)
x, y = 5, 13;
swap();
// failing assert
assume(!(x > y));
assert(false);
```

```
// sub-trace for swap()
swap();
  x = x + y;
  y = x - y;
  x = x - y;
return;
```

List. 3: Spurious counter-example trace.

**Validating Traces** Given a flattened counter-example trace $\tau$ we want to check whether the original program $P$ is actually able to execute statements in that particular order. This is the case if $\{true\}\tau\{false\}$ is no valid Hoare triple [Le05]. The validity of this Hoare triple can be checked by computing either the strongest postcondition of $\tau$ relative to *true* or the weakest precondition of $\tau$ relative to *false*. We choose to use the weakest precondition as it does not introduce quantifiers and tends to produce a smaller formula [He04, Le05]. That is, it remains to compute the weakest precondition $wp(\tau, false)$ according to the rules from Figure 5 and check whether it is equal to *true*.

$$wp(\tau_1;\tau_2,\varphi) = wp(\tau_1, wp(\tau_2,\varphi)) \qquad wp(assert(c),\varphi) = \varphi \wedge c$$
$$wp(x = e,\varphi) = \varphi[x \mapsto e] \qquad wp(assume(c),\varphi) = \varphi \vee \neg c$$
$$wp(x_1 \dots x_n = e_1 \dots e_n, \varphi) = \varphi[x_1 \mapsto e_1, \dots, x_n \mapsto e_n]$$
$$wp(f(),\varphi) = \varphi \text{ where every local variable is prefixed with some symbol}$$
$$wp(return,\varphi) = \varphi \text{ where one prefix is removed from every local variable}$$

Fig. 5: Rules of the weakest precondition calculus, adapted from [Le05].

[5] When using a symbolic encoding, e.g. BDDs [Sc04], multiple configurations might be "merged". Thus, $\pi$ could represent multiple paths with identical length if one would skip *pickone* in the definition of $\pi_{i+1}$.

## 6 Abstraction Refinement

The abstraction needs refinement when a spurious counter-example trace $\tau$ from Section 5 was found. To proceed checking the original program, the abstraction needs refinement such that $\tau$ will not be produced as counter-example trace again.

An initial idea for refinement would add all formulas that where computed as weakest preconditions during the spuriosity check from Section 5. This, however, cannot be handled by our abstraction procedure as the weakest preconditions introduce copies of local variables. Hence, we need to generate new predicates that are well-scoped and well-typed. To that end, we apply the technique proposed by Henzinger et al. [He04] which is based on Craig interpolation. A Craig interpolant [Cr57] for a pair $(\varphi^-, \varphi^+)$ is a formula $\psi$ with (a) $\varphi^- \Rightarrow \psi$, (b) $\varphi^+ \wedge \psi$ is unsatisfiable, and (c) $\psi$ does only contain variables common to $\varphi^-$ and $\varphi^+$ .

The refinement is conducted in three steps. First, a constraint trace[6] $\vartheta$ is generated. The constraint trace $\vartheta$ is generated from $\tau$ by giving every intermediate run-time value a name. That is, a variable $x$ is replaced with a symbolic constant $\langle x, k\rangle$ which denotes the $k$-th value of $x$. The formal rules for generating a constraint trace are given in Figure 6. The procedure requires a function *last* which maps variables to integers indicating the last write to a variable. This function is updated during the procedure to keep track of assignments and the most recent values of variables. Additionally, we utilize a function $upd_f$ which replaces all variables $x$ with a symbolic constant $\langle x, f(x)\rangle$.

Secondly, an interpolant for every intermediate position in $\vartheta$ is computed. That is, we compute the interpolants $\psi_1, \ldots, \psi_n$ where $n$ is the number of statements in $\vartheta$ and $\psi_i$ is the interpolant for the pair $(\varphi_i^-, \varphi_i^+)$ with

$$\varphi_i^- = \bigwedge_{j=0}^{i-1} \vartheta[j] \qquad\qquad \varphi_i^+ = \bigwedge_{j=i}^{n} \vartheta[j]$$

where $\vartheta[k]$ is the $k$-th statement in $\vartheta$. Lastly, we post-process those interpolants by replacing every symbolic constant $\langle x, k\rangle$ with its corresponding variable $x$ and extend the set of predicates with every atomic predicate contained in the post-processed interpolants. The newly added predicates are obviously well-typed and, as they originate from interpolants, are also well-scoped [He04]. The latter is due to the fact that an interpolant contains only symbols common to $\varphi^-$ and $\varphi^+$, i.e. symbols that are ”in scope” in $\varphi^-$ and $\varphi^+$.

## 7 Conclusion and Future Work

In this paper we presented an integration of state-of-the-art techniques for model checking sequential recursive integer programs. To handle the infinite state space caused by integer variables and recursion, we implemented the CEGAR loop. We instantiated this loop

[6] Such constraint traces where already introduced in [BR02], but the proposed refinement suffers from a similar problem as the naive approach based on weakest preconditions. So called *symbolic constants* are contained in the newly computed predicates which cannot be handled by our abstraction procedure.

| Statement $s$ | Constraint Trace relative to $(s,f)$ |
|---|---|
| $S_1;S_2$ | $(s';s'',f'')$<br>with<br>$(s',f') =$ constraint trace relative to $(S_1,f)$<br>$(s'',f'') =$ constraint trace relative to $(S_2,f')$ |
| $x = e;$ | $(\langle x,k\rangle = upd_f(e),\ f[x \mapsto k])$<br>with $k = f(x)$ |
| $x_1 \dots x_n = e_1 \dots e_n;$ | $(\langle x_1,k_1\rangle \dots \langle x_n,k_n\rangle = upd_f(e_1)\dots upd_f(e_n),\ f')$<br>with<br>$k_i = f(x_i)$<br>$f' = f[x_1 \mapsto k_1,\dots,x_n \mapsto k_n]$ |
| $assert(c);$ | $(assert(upd_f(c)),f)$ |
| $f();$<br>$\tau;$<br>$return;$ | $(\vartheta,\ f'')$<br>with<br>$(\vartheta,f') =$ constraint trace relative to $(\tau,f)$<br>$f'' = \begin{cases} f(x), & \text{if } x \text{ is a local variable} \\ f'(x), & \text{otherwise.} \end{cases}$ |

Fig. 6: Rules for generating a constraint trace, adapted from [He04].

with predicate abstraction to construct boolean programs which feature a finite data domain. To check those boolean programs, which still allowed recursion, we showed how to conduct a reachability analysis with procedure summaries. Here, we skipped some details of our actual implementation which uses binary decision diagrams (BDDs) [Sc04], an efficient data structure for handling boolean functions, to encode the generated control flow graphs. This representation allowed us to implement a variation of the procedure summaries. Instead of handling a single configuration at a time, our tool is able to compute a precise relation for multiple configurations at once. With the reachability analysis at hand, we were able to show the correctness of a program or to extract a counter-example. This counter-example was then checked for validity with weakest preconditions. A valid counter-example immediately proved the program incorrect, while a spurious one issued refinement. The refinement was conducted on the basis of Craig interpolation and enriched the abstraction with new predicates.

As our model checker was developed during a masters project, we did simplify and skip some parts. First of all, we did no exhaustive benchmarking and performance analysis. This is considered to be future work. During such an activity bottlenecks of our implementation might be identified which one could try to cure. Currently, we believe that our checker is mostly busy computing the abstraction whereas the remaining parts seem to take only a minor part of the computation time. One promising approach addressing this potential bottleneck is the so called *parsimonious* abstraction from [He04].

A further aspect, which is considered future work, too, is an improved language featuring more powerful functions. To that end, one could introduce formal parameters and (multiple) return values. There are already results for abstraction and refinement procedures for those kind of functions available in [Ba01, He04]. We believe that more powerful functions could reduce the size of programs and thus the size of the generated control flow graphs. This reduction could then lead to performance improvements.

Another area of further improvement is the counter-example generation. Currently, counter-example paths are computed by a simple breadth-first search in the k-step reachability sets. Here, state-of-the-art approaches, like proposed in [Cl95], might improve our checker.

## References

[Ba01] Ball, Thomas; Majumdar, Rupak; Millstein, Todd D.; Rajamani, Sriram K.: Automatic Predicate Abstraction of C Programs. In (Burke, Michael; Soffa, Mary Lou, eds): Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah, USA, June 20-22, 2001. ACM, pp. 203–213, 2001.

[BR00] Ball, Thomas; Rajamani, Sriram K.: Bebop: A Symbolic Model Checker for Boolean Programs. In (Havelund, Klaus; Penix, John; Visser, Willem, eds): SPIN Model Checking and Software Verification, 7th International SPIN Workshop, Stanford, CA, USA, August 30 - September 1, 2000, Proceedings. volume 1885 of Lecture Notes in Computer Science. Springer, pp. 113–130, 2000.

[BR02] Ball, Thomas; Rajamani, Sriram K.: Generating Abstract Explanations of Spurious Counterexamples in C Programs. Technical Report MSR-TR-2002-09, Microsoft Research, January 2002.

[Cl95] Clarke, Edmund M.; Grumberg, Orna; McMillan, Kenneth L.; Zhao, Xudong: Efficient Generation of Counterexamples and Witnesses in Symbolic Model Checking. In: DAC. pp. 427–432, 1995.

[Cl00] Clarke, Edmund M.; Grumberg, Orna; Jha, Somesh; Lu, Yuan; Veith, Helmut: Counterexample-Guided Abstraction Refinement. In (Emerson, E. Allen; Sistla, A. Prasad, eds): Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings. volume 1855 of Lecture Notes in Computer Science. Springer, pp. 154–169, 2000.

[Cr57] Craig, William: Linear Reasoning. A New Form of the Herbrand-Gentzen Theorem. J. Symb. Log., 22(3):250–268, 1957.

[GS97] Graf, Susanne; Saïdi, Hassen: Construction of Abstract State Graphs with PVS. In (Grumberg, Orna, ed.): Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, June 22-25, 1997, Proceedings. volume 1254 of Lecture Notes in Computer Science. Springer, pp. 72–83, 1997.

[He04] Henzinger, Thomas A.; Jhala, Ranjit; Majumdar, Rupak; McMillan, Kenneth L.: Abstractions from proofs. In (Jones, Neil D.; Leroy, Xavier, eds): Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004. ACM, pp. 232–244, 2004.

[JM09] Jhala, Ranjit; Majumdar, Rupak: Software model checking. ACM Comput. Surv., 41(4), 2009.

[Le05] Leino, K. Rustan M.: Efficient weakest preconditions. Inf. Process. Lett., 93(6):281–288, 2005.

[Mi67] Minsky, Marvin L.: Computation: Finite and Infinite Machines. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1967.

[Sc04] Schneider, Klaus: Verification of Reactive Systems: Formal Methods and Algorithms. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.