Rheinisch-Westfälische Technische Hochschule Aachen
Lehrstuhl für Informatik 2
Software Modeling and Verification

Bachelor Thesis

Optimization of Model Checking by Large Block Encoding

Thomas Mertens

September 24th 2014

First Reviewer:
Prof. Dr. Thomas Noll
Second Reviewer:
Prof. Dr. Joost-Pieter Katoen

Advisor:
M.Sc. Tim Lange

Erklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Aachen, 24. September 2014

_____

Thomas Mertens

Abstract

Reducing the runtime is one of the main problems in software model checking. Especially the runtime of CEGAR algorithms can be dramatically reduced by minimizing the abstract reachability tree. Using large-block encoding loop free parts of a control flow automaton are summarized into a single edge. This thesis provides the theoretical background of large-block encoding and introduces an advanced version of large-block encoding. Both versions of large-block encoding are evaluated on several test cases.

Contents

List of Figures

1 Introduction

Code verification can improve software development in various design phases. We distinguish between two types of code verification: a semantic check of code and software model checking. A semantic check of a software can be done for example via the Hoare calculus. By using the Hoare calculus the program or a set of functions of the program can be checked. Therefore, one introduces preconditions and postconditions for each function. For a precondition P, a program Q and the postcondition R following notation is used: $P\{Q\}R$. If the precondition P holds before executing the program Q, the postcondition R must hold as well. In case that the program Q has no precondition one denotes it with $true\{Q\}R$ [Hoa69].
Software model checking is another prominent technique which will be evaluated in this thesis. Due to many years of research and development, model checking tools are able to check software of remarkable size. But still, the problems of efficiency and scalability are not solved completely. The main problem is the state space explosion. The amount of states is correlated exponentially with the number of variables. For n variables in a domain of k possible values there are k^n possible states. Even small program have a huge amount of states, for example a program with 10 locations, 4 boolean variables and 3 integers with domain in $\{0, ..., 9\}$ has $10 \times 2^4 \times 10^3 = 160.000$ states [BK08]. The state explosion problem can be handled by abstraction. The abstraction of a program is usually defined by a function, which transforms a set of concrete states of the program into an abstract state. A single abstract state might represent a set of various concrete states. This leads to various problems in the adoption of the techniques in practical use [BCG+09]. For example one needs a more complex formula to represent an abstract state than for representing a concrete state, due to the fact, that an abstract state might be a set of concrete states.
Research and development made it obvious that the use a of so-called *abstract reachability tree (ART)* and predicate abstraction can enhance the verification process. In first approaches of using ARTs in software model checking tools every program operation is described as single edge in the ART. This method is called *single-block encoding (SBE)*, which leads to very big control flow automata during realistic application. Due to this, the ART, which is unrolled from the CFA, is very big as well. For model checking a big ART is a main problem. A control flow automaton (CFA) represents the control flow of a program. This is why their size has to be reduced which can be done by *large-block encoding (LBE)*. The thesis focuses on large-block encoding and the effect to model checking algorithms. This approach does not represent a single program operation in a one edge, but rather contains entire parts of the program at once. This is how the number of nodes is reduced potentially exponentially [BCG+09].
Using large-block encoding instead of single-block encoding has two main consequences.

On the one hand, a more general representation of the abstract states is required for large-block encoding. In this thesis there are three standard rules [BCG+09] and one new rule. The advanced sequence rule was developed during this thesis. Each rule aims at the minimization of a given control flow automaton. After a rule application the CFA is smaller, regarding the number of states or the number of edges depending the applied rule, than before. For representing the abstract state one needs an arbitrary boolean combination of predicates, which defines an abstract state. The other consequence is that for the abstract-successor computations in large-block encoding a more accurate abstraction is required. An abstract edge represents several paths of the program [BCG+09].
When using large-block encoding with boolean abstraction in addition to traditional techniques one can observe two main points: First, the large-block encoding approach can reduce the amount of successor computations up to exponentially. Second, the abstraction which is used for large-block encoding is more expensive than the one which is used for single-block encoding. The large-block encoding abstraction needs more time and memory, because all satisfiable assignments are required [BCG+09].
To make use of counterexample-guided abstraction refinement one also has to take into account how the control-flow automaton is labelled. As label for the control flow automaton the guarded command language of Dijkstra, which is explained in Sec. 4.3, is used. By using the guarded command language the weakest precondition can be calculated very efficiently. The weakest precondition needs to be calculated for using the counterexample-guided abstraction refinement.

After providing the introduction and a short overview about related work, this thesis starts with giving the necessary background information for large-block encoding. In Section 3 the rules of large-block encoding and its termination and runtime are described. Section 4 gives an overview over special details of implementation and problems during implementation and testing. The results of performances tests are topic of section 5. Finally, Section 6 concludes this thesis.

1.1 Related Work

There are two typical examples for model checkers that use the single-block encoding approach: BLAST and SLAM [BHJM07, BR02]. Both of them are based on the counterexample-guided abstraction refinement (CEGAR) [CGJ+00]. Model checking, however, also uses several other approaches; for instance lazy abstraction can be found in SATabs [HJMS02]. Instead of using predicate abstraction, [McM06] proposes an approach based on Craig interpolants from infeasible error paths.

Another way of minimizing a control flow automaton is *adjustable-block encoding (ABE)* [BKW10]. ABE fills the gap of missing configurations and unifies single-block encoding and large-block encoding. Formulas for large blocks of the control flow automaton are constructed on the fly during the analysis. The number of operations summarized in a block is given by a special parameter. Therefore it is possible to construct completely

new blocks, which are not possible with single-block encoding or large-block encoding. Even bigger blocks than summarized with large-block encoding are possible by using ABE. In comparison to large-block encoding an abstract state has to store different formulas. First, there is the abstraction formula, which is result of the the abstraction computation. Second, the path formula representing the strongest postcondition since the last computation of an abstract state [BKW10].

2 Background

This chapter provides the theoretical background of software model checking via large-block encoding. Given a program in a language, we start by constructing the control flow automaton via standard construction rules. On top of that the predicate abstraction, which is necessary to represent abstract states, is explained. Finally bounded model checking, the counterexample-guided abstraction refinement and the fixpoint theorem of Tarski and Knaster are topics of this subsection.

2.1 Control Flow Automaton

Definition 1 (Operations)**.** *The operations which are used in a control flow automaton are defined by the language*

$$basic\_op = x := c \mid x \circ c \text{ with } c \text{ variable or value or expression } \circ \in \{=, \leq, \geq, \neq, <, >\}$$
$$op = basic\_op \mid op_1; op_2 \mid op_1 \mid\mid op_2$$

Definition 2 (Program)**.** *A sequential program P with the operation set Ops is defined as a finite sequence $P = op_0, op_1, ..., op_n$ with $op_i \in Ops$. A program location is represented by the index its subsequent operation.*

Definition 3 (Control Flow Automaton)**.** *A Control Flow Automaton (CFA) A is defined as the four tuple $A = (L, G, l_0, L_E)$, where L is a finite set of locations, G is a set $L \times Ops \times L$, $l_0 \in L$ is a unique initial location and $L_E \subseteq L$ is a set of error locations. The initial location l_0 has no ingoing edges.*

The set L models the program counter l and G represents the edges in the CFA. We distinguish between two different types of *basic_op*s. First, an operation in a program can be an assignment such as $x := y$. Second, an operation can be an assumption like $x > y$ which is evaluated either to true or false. This definition of edge labels is more general but after applying large-block encoding one can not distinguish between assume and assign edges. When moving from location $l_x \in L$ to location $l_y \in L$, where $l_x = l_y$ is allowed due to self-loops, one operation $op_x \in Ops$ is executed. The set of all used variables in at least one operation is called VAR.

Definition 4 (Function on a Control Flow Automaton)**.** *To use a Control Flow Au-*

tomaton easier in applications the following functions are defined:

$$\begin{aligned} &\text{Let } l_x \in L \\ succ(l_x) &:= \{l_y | (l_x, op_x, l_y) \in G \wedge op_x \in Ops \wedge l_y \in L\} \\ pred(l_x) &:= \{l_y | (l_y, op_y, l_x) \in G \wedge op_y \in Ops \wedge l_y \in L\} \\ outdegree(l_x) &:= |\{(l_x, op_x, l_y) \in G\}| \\ indegree(l_x) &:= |\{(l_y, op_x, l_x) \in G\}| \\ connecting\_edges(l_x, l_y) &:= \{(l_x, op_x, l_y) \in G\} \end{aligned}$$

In the following, initial locations shown in figures are marked with dashed borders, error location with a red filling.

An function $c : VAR \to Z$ that assigns an integer value to every variable is called a concrete data state, where Z is the domain of integer values. All concrete data states are in the set denoted by C. A region is a subset of concrete data states and is represented with a first-order formula φ. This formula φ defines a set S of all data states c that model φ. A concrete data state is a set of all concrete states with $\{(l, c) \mid c \models \varphi\}$.
The strongest postcondition operator SP_{op} defines the concrete semantics of an operation $op \in Ops$. SP_{op} indicates the set consisting of all states that are reachable from the region, given by the formula φ, after executing an operation op. For a given formula φ and an assignment operation $s := e$ we have $SP_{s:=e}(\varphi) = \exists s' : \varphi_{\{s \longmapsto s'\}} \wedge (s = e_{\{s \longmapsto s'\}})$ as well as for an assume operation $assume(p)$ we have $SP_{assume(o)}(\varphi) = \varphi \wedge p$. The notation $s \longmapsto s'$ says that s is replaced by s' [BCG+09].

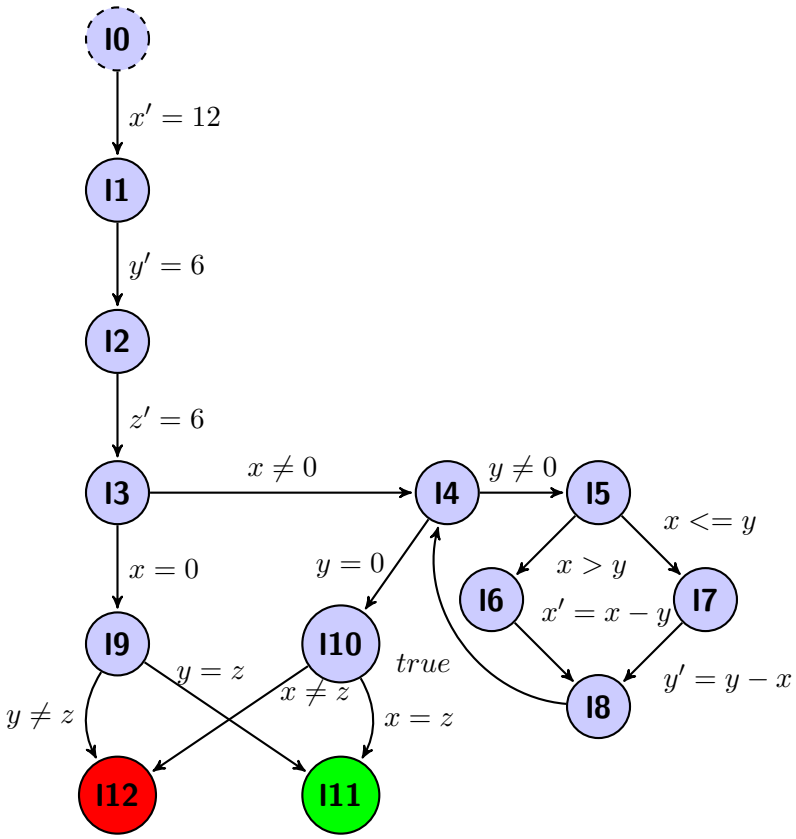
A path σ is a sequence $\langle (op_0, l_1), ..., (op_{n-1}, l_n) \rangle$ of operations $(\forall i \in \{0, n-1\}.op_i \in Ops)$ and locations $(l_i \in L \; \forall i \in \{1, n\})$. If G contains edges (l_{i-1}, op_{i-1}, l_i) s.t. there is a path from l_0 to a terminating location $l_t \in L$ σ is a program path. The successive application of the strongest postoperator for path $SP_\sigma(\varphi) = SP_{op_{n-1}}(...SP_{op_0}(\varphi)...)$ defines the concrete semantics for a program path $\sigma = \langle (op_0, l_1), ..., (op_{n-1}, l_n) \rangle$. If $SP_\sigma(true)$ is satisfied the program path σ is feasible. A concrete state (l_i, c_i) of a path σ is reachable if the path σ is feasible and ends in the location l_i such that $c_i \models SP_\sigma(true)$. An arbitrary location l can be reached if there is a reachable concrete state (l, c). In case the location l_E is not reachable the program is safe [BCG+09].

Sample Transformation As mentioned in the introducing lines of this section programs can be transformed into CFAs. This will be performed on the following example code. The code checks if z is the greatest common divisor of x and y. When reaching a "return true" the program terminates successfully, when reaching "return false" the program terminates with exception. The first step by transforming code into a CFA is creating an initial location. Second, the program lines are read one by one. For each assignment a new edge to the next location is created. For an if statement the CFA is divided into branches which are merged at the end of the if statement. Loops are also represented by branches in the CFA, where one branch ends in the start location of

the branch and the other branch proceeds to the next location in the program. In this example it can be seen at location $l4$.

```
x:=12
y:=6
z:=6
if(x=0) then
  if(y=z) then
    return true
  else
    return false
else
  while (y!=0) do begin
   if(x>y) then
     x:=x-y
   else
     y:=y-x
  end
if(x=z) then
  return true
else
  return false
```

(a) Code



(b) CFA

Figure 2.1: Sample transformation

2.2 Predicate Abstraction

Predicates can be seen as boolean expressions. A theory T is defined over a set of variables, a quantifier-free theory has no quantifiers, i.e. $\exists$ or $\forall$ in the formula [KS08]. Therefore, the predicates in T are only expressions over the variables in the program. One cannot use variables which do not occur in the program. The predicates over program variables in a quantifier-free theory T are in a set $\mathcal{P}$. A formula φ is a Boolean combination (with boolean operator $\vee, \wedge, \neg$) of predicates from $\mathcal{P}$. A precision of a formula is a finite subset $\pi \subset \mathcal{P}$ of predicates.

2.2.1 Cartesian Predicate Abstraction

The basic idea of Cartesian predicate abstraction is ignoring dependencies between components of tuples. For example, the set $\{< 0, 1 >, < 1, 0 >\}$ is abstracted by the Cartesian predicate abstraction with $< *, * >$ where $*$ is a do not care value. Therefore the

abstracted set contains four elements in total, where the original set does only contain two elements.

The Cartesian predicate abstraction φ^{π}_{C} of the formula φ is the strongest conjunction of predicates in π that is caused by $\varphi : \varphi^{\pi}_{C} := \bigwedge\{p \in \pi | \varphi \Rightarrow p\}$ [BCG+09]. This abstraction for a formula φ representing a region of concrete program states is used in program verification to represent an abstract state. The Cartesian predicate abstraction for a given formula φ and a given precision π can also be computed by $|\pi|$ SMT-Solver queries. The abstract strongest postoperator SP^{π} for a predicate abstraction with precision π transforms the abstract state φ^{π}_{C} into its successor $\varphi{*}^{\pi}_{C}$ for a program operation $op \in Ops$. This is written as follows: $\varphi{*}^{\pi}_{C} = SP^{\pi}_{op}(\varphi^{\pi}_{C})$. In case that $\varphi{*}^{\pi}_{C}$ is a Cartesian predicate abstraction of $SP_{op}(\varphi^{\pi}_{C})$ then $\varphi{*}^{\pi}_{C} = (SP^{\pi}_{op}(\varphi^{\pi}_{C}))^{\pi}_{C}$ [BPR03].

2.2.2 Boolean Predicate Abstraction

The abstract states are separated into different classes depending on the evaluation under a finite set of predicates, so called Boolean expressions. Let $\mathcal{P} = \{p_1, ..., p_n\}$ be a finite set of predicates. Each p_y for $y \in \{1...n\}$ denotes a set of states s.t. $\{s \in L \mid s \models p_y\}$. The set $\mathcal{P}$ defines two things, first an abstract transition system and second the strongest postoperator. The abstract transition system creates an equivalence relation between the states, these equivalences partition the state space. Two states are in the same equivalence class, if they satisfy the same set $\mathcal{P}' \subseteq \mathcal{P}$ [BPR03]. Following the previous example, the set $\{< 0, 1 >, < 1, 0 >\}$ is abstracted by the Boolean predicate abstraction with $\neg(x \wedge y) \wedge (x \vee y)$. The Boolean abstraction is much more precise than the Cartesian abstraction, because it only in contains two elements instead of four elements.
The Boolean predicate abstraction φ^{π}_{B} of the formula φ is the strongest combination of predicates from π that is entailed by φ. For a given precision π and a formula φ querying the SMT solver results in the Boolean predicate abstraction φ^{π}_{B}. The SMT solver does it in the following way: A new propositional variable v_i gets introduced for every predicate $p_i \in \pi$. The SMT solver is asked to enumerate all satisfying assignments of $v_1...v_{|\pi|}$ in the formula $\varphi \wedge \bigwedge_{p_i \in \pi}(p_i \Leftrightarrow v_i)$. A conjunction of all predicates from π whose corresponding propositional variable occurs not negative in the assignment is computed for each satisfying assignment. The Boolean predicate abstraction for φ is the disjunction over all such conjunctions. The abstract strongest postoperator SP^{π} for a predicate abstraction with precision π transforms the abstract state φ^{π}_{B} into its successor $\varphi{*}^{\pi}_{B}$ for a program operation $op \in Ops$. This is written as follows: $\varphi{*}^{\pi}_{B} = SP^{\pi}_{op}(\varphi^{\pi}_{B})$. In case that $\varphi{*}^{\pi}_{B}$ is a Boolean predicate abstraction of $SP_{op}(\varphi^{\pi}_{B})$ then $\varphi{*}^{\pi}_{B} = (SP^{\pi}_{op}(\varphi^{\pi}_{B}))^{\pi}_{B}$ [LNO06].

2.3 Bounded Model Checking

The Bounded Model Checker (BMC) unrolls the program up to a given bound k, i.e. the BMC does only check the first k steps in the program. The given property is only checked up to the given bound k. If there is a counterexample in this part of the program

such that the property is violated, the bounded model checker finds this counterexample. If the BMC does not find counterexample within the first k steps of the program, one can not say that the program is safe or not. Due to the given bound, the BMC will not find a counterexample with a path length $> k$. Even in case that the property is violated at position $k + 1$ the counterexample will not be detected. By using BMC the returned counterexample is the minimal one that violates the property, if there exists a violation up to bound k. The main disadvantage of BMC is caused by the given bound. Therefore, the BMC can not proof the correctness of a program in general. Another technique of checking if a program violates or satisfies a property is topic of the next section [BCC+03].

2.4 Counterexample-Guided Abstraction Refinement

The counterexample-guided abstraction refinement is one of the used ways to model check a program in this thesis. Counterexample-guided abstraction refinement checks if a program abstraction contains any counterexample to prove that at least one assertion is violated. Before checking the existence of a counterexample the abstraction of the program has to be constructed. The abstraction of a program P is called P'. The abstraction function for a program P is a surjective function $\alpha : D \to D'$ where D is the set of all possible concrete states in P and D' the abstract domain of states in P'. After constructing the abstraction the algorithm checks the existence of a path to a state $l_e \in l_E$, a so called counterexample for a given assertion. If there does not exist a counterexample, the program does not violate the assertion and can be called safe. In case that there is a counterexample, one distinguishes between real and spurious counterexamples. A real counterexample is feasible in the concrete program. Having a real counterexample, the program is called not safe [CGJ+00].
Second, a counterexample can be spurious, i.e. the counterexample is not feasible in the concrete program. In this case the abstraction of the program has to be refined. During the refinement process the locations, which are responsible for creating the spurious counterexample, have to be found. The equivalence classes which are separated from the abstracted states have to be modified as well. After the abstraction is refined, the assertion is checked again. As long as the counterexample is spurious, one can not say if the program is safe or unsafe. To decide if a program is safe or not, it has to be ensured, that the found counterexample is not a result of a coarse abstraction of the program. The progress of refining a program does not terminate in any case.
The following figure illustrates the run of the counterexample-guided abstraction refinement algorithm.

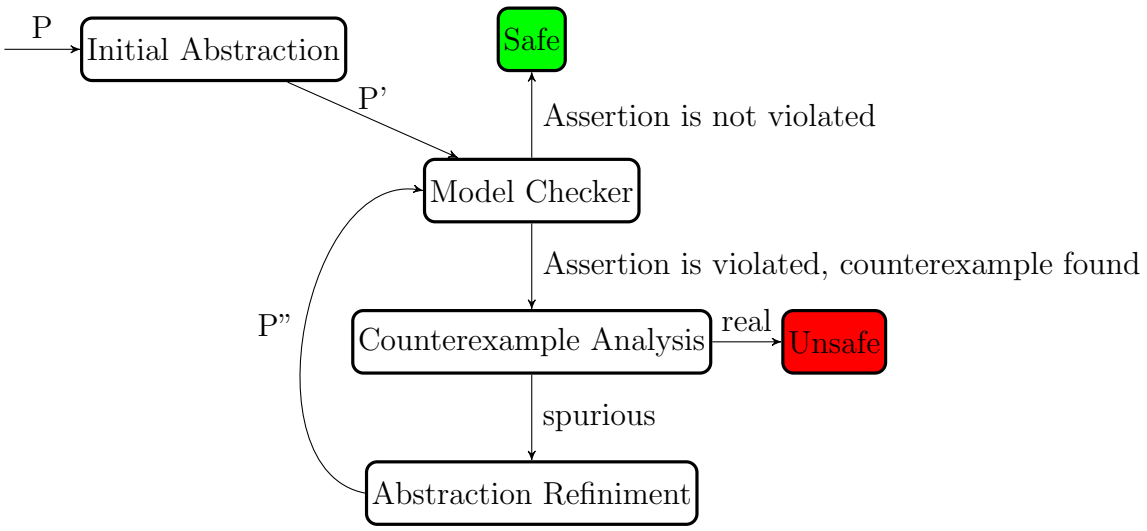


Figure 2.2: Counterexample Guided Abstraction Refinement

A sample abstraction as it is done in the counterexample-guided abstraction refinement algorithm is performed on the following program shown in Fig. 2.3.

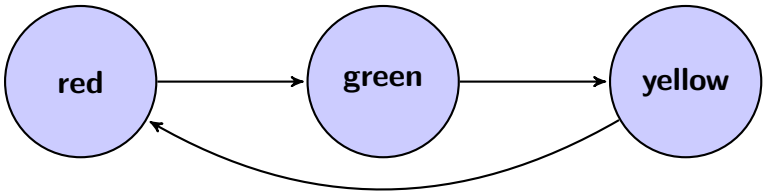


Figure 2.3: Original program representing a simplified traffic light

The shown program M represents a simplified traffic light. Every run of the program contains a sequence $< red, green, yellow >$. Let $\psi = \mathbf{AGAF}(state = red)$ be a property to check [CGJ+00]. The property says that on every path one will eventually reach a red state. Obviously $M \models \psi$. The abstraction function α constructs the abstraction of the initial program as follows:

$$\alpha(red) = red$$
$$\alpha(green) = \alpha(yellow) = go$$

This leads to the abstraction M' shown in Fig. 2.4.

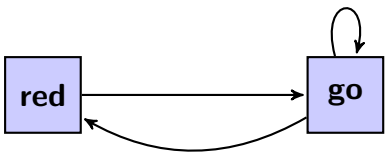


Figure 2.4: Quotient transition system of a simplified traffic light program

States are summarized in an equivalence class if they satisfy the same property. If two states of the same equivalence class are connected in the original program one gets a self-loop for this equivalence class in the abstraction. Two equivalences classes are connected if the states are connected in the original program.

By using this abstraction an infinite sequence like $< red, go, go, go, ... >$ can be taken. Such a sequence violates the given property, therefore $M' \not\models \psi$. The counterexample-guided abstraction refinement algorithm checks the counterexample if it is real or spurious. This counterexample can never be taken in the original program shown in Fig. 2.3. Therefore, the abstraction causes a spurious counterexample has to be refined. Every other abstraction, which differs from the original program, will return a spurious counterexample.

2.5 Fixpoint Theorem by Tarski and Knaster

The termination of large-block encoding will be shown using the Fixpoint Theorem by Tarski and Knaster [Tar55]. To be able to show the termination in the next section the needed preliminaries are provided in the following paragraph [Nol12]. At first the partial order is introduced and is needed to be able to compare analysis results.

Definition 5 (Partial order)**.** *A partial order (PO)* $(D, \trianglerighteq)$ *consists of a domain* D *and a relation* $\trianglerighteq \subseteq D \times D$ *such that, for every* $d_1, d_2, d_3 \in D$*,*

$$\begin{aligned} \textit{reflexivity: } & d_1 \trianglerighteq d_1 \\ \textit{transitivity: } & d_1 \trianglerighteq d_2 \textit{ and } d_2 \trianglerighteq d_3 \Rightarrow d_1 \trianglerighteq d_3 \\ \textit{antisymmetry: } & d_1 \trianglerighteq d_2 \textit{ and } d_2 \trianglerighteq d_1 \Rightarrow d_1 = d_2 \end{aligned}$$

To define a point of no further improvement the fact of a least upper bound for a partial order has to be introduced.

Definition 6 ((Least) upper bound)**.** *Let* $(D, \trianglerighteq)$ *be a partial order and* $S \subseteq D$*.*

(1) An element $d \in D$ *is an upper bound of* S *if* $s \trianglerighteq d$ *for every* $s \in S$*.*
(2) An upper bound d *of* S *is least upper bound of* S *if* $d \trianglerighteq d'$ *for every upper bound* d' *of* S*.*

Definition 7 (Complete lattice)**.** *A complete lattice is a partial order* $(D, \trianglerighteq)$ *such that all subsets of* D *have least upper bounds. In this case*

$$\bot := \sqcup \emptyset$$

denotes the least element of D*.*

Definition 8 (Ascending Chain Condition)**.** *A sequence* $(d_i)_{i \in \mathbb{N}}$ *is called an ascending chain in* D *if* $d_i \trianglerighteq d_{i+1}$ *for each* $i \in \mathbb{N}$*.*
A partial order $(D, \trianglerighteq)$ *satisfies the Ascending Chain Condition(ACC) if each ascending chain* $d_0 \trianglerighteq d_1 \trianglerighteq ...$ *eventually stabilizes, i.e. there exists* $n \in \mathbb{N}$ *such that* $d_n = d_{n+1} = ...$

After defining the partial order and least upper bound for each arbitrary subset of a partial order. It is already said, that a partial can satisfy the ascending chain condition. To ensure that the ascending chain condition is not violated, there must be a monotonic function.

Definition 9 (Monotonicity)**.** *Let* $(D, \trianglerighteq)$ *and* $(D', \trianglerighteq')$ *be partial orders, and let* $\Phi : D \to D'$*.* Φ *is called monotonic if, for every* $d_1, d_2 \in D$*,*

$$d_1 \trianglerighteq d_2 \Rightarrow \Phi(d_1) \trianglerighteq' \Phi(d_2)$$

After all preliminaries which are required for the application of the Fixpoint Theorem by Tarski and Knaster are mentioned and defined, the fixpoint theorem can be defined in the next step.

Theorem 1 (Fixpoint Theorem by Tarski and Knaster)**.** *Let* $(D, \trianglerighteq)$ *be a complete lattice satisfying Ascending Chain Condition and* $\Phi : D \to D$ *monotonic. Then*

$$fix(\Phi) := \sqcup\{\Phi^k(\bot) | k \in \mathbb{N}\}$$
is the least fixpoint of Φ *where*
$$\Phi^0(d) := d \text{ and } \Phi^{k+1}(d) := \Phi(\Phi^k(d))$$

3 Large-Block Encoding

In this section we use the definition of a program and a CFA as given in Section 2.1. At first this section gives the explanation of the three standard rules of large-block encoding and the advanced sequence rule. Later on the termination of the advanced large-block encoding version is shown. Finally, the runtime of large-block encoding is analysed.

3.1 Error Sink Rule

Given a location $l_e \in L_E$ with $outdegree(l_e) \geq 1$ and outgoing edges $G_{out} = \{(l_e, op_x, l_x) \in G\}$. The error sink rule removes all edges $g \in G_{out}$.

The error sink rule is executed once at the beginning of large-block encoding. By applying this rule all outgoing edges for all locations in the set of error location (l_E) are removed. This is valid, because every model checking algorithm will immediately stop when reaching an error location. So there is no reason to keep these edges in the CFA, see Fig. 3.1.

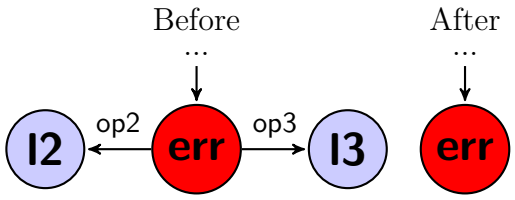


Figure 3.1: Example for error sink rule

3.2 Sequence Rule

Given a location $l_x \in L \setminus L_E$ with $indegree(l_x) = 1$, $outdegree(l_x) \geq 1$,$l_x \notin succ(l_x)$ and the edge $(l_y, op_y, l_x) \in G$, $G_{out} = \{(l_x, op_x, l_z) \in G\}$. The sequence rule adds new edges with $(l_y, op_y; op_x, l_z)$ $\forall g \in G_{out}$ to the CFA and removes all edges $g \in \{(l_y, op_y, l_x) \in G\} \cup G_{out}$.

The basic sequence rule of large-block encoding can only be applied if there is a location which has an in-degree of one. In that case G contains an edge (l_x, op_x, l_y) and further edges $\{(l_y, op_y, l_z) | l_z \in succ(l_y)\}$ the sequence rule can be applied. By applying the rule all ingoing and outgoing edges of location l_y will be combined. Therefore new edges from l_x to l_z are added to the CFA and the edges (l_x, op_x, l_y) and $\{(l_y, op_y, l_z) | l_z \in succ(l_y)\}$ are deleted. The edge label of the new edges is the sequential execution of op_x and op_y.

Figure 3.2 shows that every run of the program or part of the program that visits location l_1 and is going to location l_3 or l_4 is forced to visit location l_2. Therefore location l_2 can be removed and the operation which is executed before reaching location l_2 is added to the operations executed before reaching location l_3 or l_4.

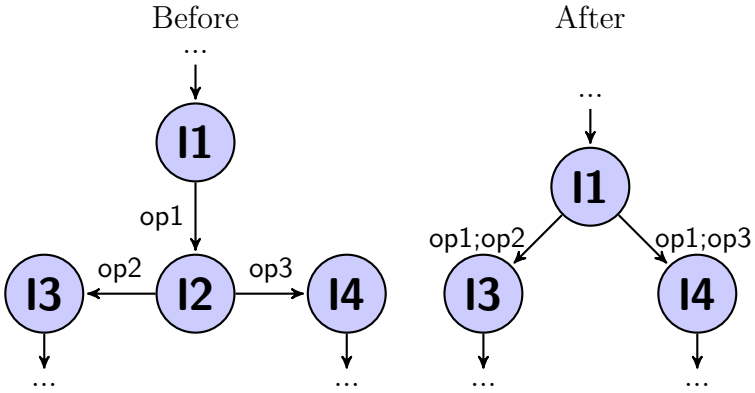


Figure 3.2: Example for sequence rule

3.3 Choice Rule

Given a location $l_x \in L \setminus L_E$ with $l_y \in succ(l_x)$, $G_{con} = connecting\_edges(l_x, l_y)$ and $\mid G_{con} \mid = n > 1$. The choice rule adds a new edge (l_x, op, l_y) with
$op = op_1 \mid\mid op_2 \mid\mid ... \mid\mid op_n$ to the CFA and removes all edges $g \in G_{con}$.

The CFA can be reduced by removing all edges $(l_x, op, l_y) \in G$ and adding a new edge (l_x, op_{new}, l_y) with $op_{new} = op_1 \mid\mid op_2 \mid\mid ... \mid\mid op_n$ s.t $(l_x, op_i, l_y) \in G \; \forall i \in \{1...n\}$. $op_x \mid\mid ... \mid\mid op_y$ formalises the non-deterministic choice, i.e. that either op_x or op_y has to be executed. In case a part of the CFA contains two locations that are connected with at least two different edges, all edge labels can be summarized in a single edge label, see Fig. 3.3.

3.4 Advanced Sequence Rule

Given a location $l_x \in L \setminus L_E$ with $indegree(l_x) > 1$, $outdegree(l_x) > 1$, $l_x \notin succ(l_x)$ and the edge sets $G_{in} = \{(l_y, op_y, l_x) \in G\}$, $G_{out} = \{(l_x, op_x, l_z) \in G\}$. The sequence rule adds new edges with $(l_y, op_y; op_x, l_z)$ for all possible connections of edges in G_{in} and G_{out} to the CFA and removes all edges $g \in G_{in} \cup G_{out}$.

While working on large-block encoding and testing the three previous rules on some sample CFAs the following question came up: can we apply a modified sequence rule on locations which have more than one incoming edge? In case the location does not have any self-loops we can apply an advanced sequence rule. All incoming edges will be

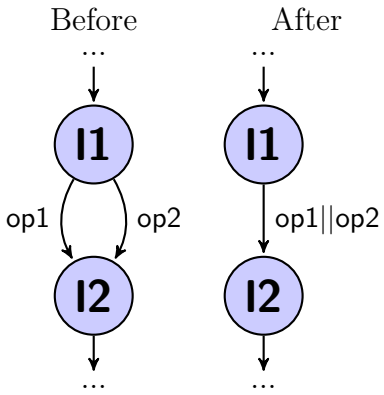


Figure 3.3: Example for choice rule

modified such that they will start in their original location and end in all successors of the intermediate location l_i. The edge labels are connected by the sequential execution operator ";", such that the new edge label is the sequential execution of op_x and op_y. By applying this rule one location and $(indegree(l_i) + outdegree(l_i))$ edges are removed and $(indegree(l_i) \times outdegree(l_i))$ edges are added to the CFA, see Fig. 3.4.

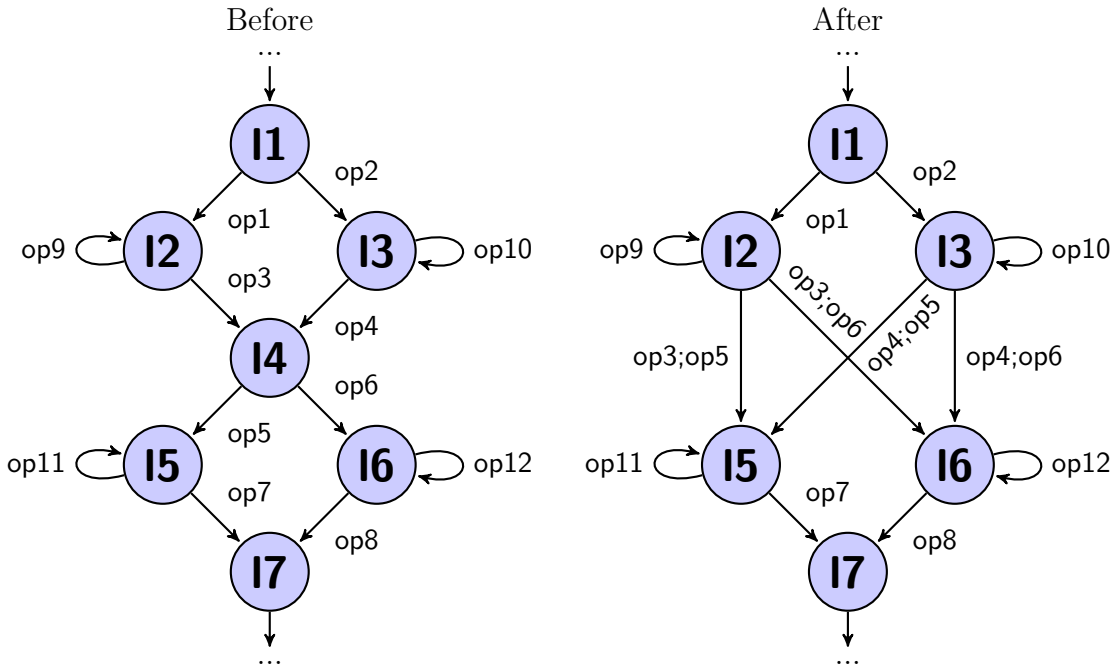


Figure 3.4: Example for advanced sequence rule

One application of large-block encoding does not change the reachability for at least ons error state. Due to the error sink rule it might be the case that an error state which is a successor of another error state is not reachable in the large-block encoding CFA. But reaching one error state is enough for getting a valid result of the model checker. If one

or more error states are reachable in the single-block encoding CFA, then at least one error state is reachable in the large-block encoding CFA. If an error state is reachable in the large-block encoding CFA then it is definitly reachable in the single-block encoding CFA.

3.5 Termination for Large-Block Encoding

The termination proof of large-block encoding mentioned in the official paper can not be applied any more. Due to the advanced sequence rule there is no monotonic function on edges representing the LBE. To show the termination of this version of large-block encoding the fixpoint theorem of Tarski and Knaster is used. Let $(D, \trianglerighteq)$ be a partial order, where every element d of the domain D and the relation are defined as:

$$\begin{aligned} d &= (|L|, |G|) \\ d_1 \trianglerighteq d_2 &= (a, b) \trianglerighteq (c, d) \equiv c \leq a \wedge (c < a \vee d \leq b) \end{aligned}$$

Definition 10 (Rule application)**.** *The application of one of the four rules on a given* $d = (|L|, |G|)$ *is defined as*

$$d' = rule_x(d),\ x \in \{sink, sequence, choice, advanced\}$$

If no rule can be applied the CFA is returned without any changes.

As mentioned in Sec. 2.5 there are properties to proof before applying the Fixpoint Theorem. At first it is shown, that $(D, \trianglerighteq)$ is a partial order.

- reflexivity $d_1 \trianglerighteq d_1$:

$$\begin{aligned} \text{Let } d_1 &= (a, b) \\ d_1 \trianglerighteq d_1 &= (a, b) \trianglerighteq (a, b) \\ &\equiv a \leq a \wedge (a < a \vee b \leq b) \\ &\text{obviously this holds} \end{aligned}$$

- transitivity: $d_1 \trianglerighteq d_2$ and $d_2 \trianglerighteq d_3 \Rightarrow d_1 \trianglerighteq d_3$

  Let $d_1 = (a, b), d_2 = (c, d)$ and $d_3 = (e, f)$
  (1) $d_1 \trianglerighteq d_2 = (a, b) \trianglerighteq (c, d) \equiv c \leq a \wedge (c < a \vee d \leq b)$
  (2) $d_2 \trianglerighteq d_3 = (c, d) \trianglerighteq (e, f) \equiv e \leq c \wedge (e < c \vee f \leq d)$
  From (1) and (2) it follows:
  (3) $e \leq c \wedge c \leq a \Rightarrow e \leq a$
  (4a) $(c < a \vee d \leq b) \wedge (e < c \vee f \leq d)$
  (4b) $\equiv ((c < a \wedge e < c) \vee (c < a \wedge f \leq d) \vee (d \leq b \wedge f \leq d) \vee (d \leq b \wedge e < c))$
  (4c) $\rightarrow ((e < a) \vee (c < a \wedge f \leq d) \vee (f \leq b) \vee (d \leq b \wedge e < c))$
  (5) $e \leq a \equiv e < a \vee e = a$
  (6) Because of (4c) $e = a \rightarrow (f \leq b)$
  Because of (3), (6) and (5):
  $\Rightarrow e \leq a \wedge (e < a \vee f \leq b) \equiv (a, b) \trianglerighteq (e, f) = d_1 \trianglerighteq d_3$

- antisymmetry: $d_1 \trianglerighteq d_2$ and $d_2 \trianglerighteq d_1 \Rightarrow d_1 = d_2$

  Let $d_1 = (a, b)$ and $d_2 = (c, d)$
  (1) $d_1 \trianglerighteq d_2 = (a, b) \trianglerighteq (c, d) \equiv c \leq a \wedge (c < a \vee d \leq b)$
  (2) $d_2 \trianglerighteq d_1 = (c, d) \trianglerighteq (a, b) \equiv a \leq c \wedge (a < c \vee b \leq d)$
  From (1) and (2) it follows:
  (3) $c \leq a \wedge a \leq c \Rightarrow a = c$
  (4) $(c < a \vee d \leq b) \wedge (a < c \vee b \leq d)$
  $\equiv ((c < a \wedge a < c) \vee (c < a \wedge b \leq d) \vee (d \leq b \wedge a < c) \vee (d \leq b \wedge b \leq d))$
  $\equiv ((c \neq a) \vee (c < a \wedge b \leq d) \vee (d \leq b \wedge a < c) \vee (d = b))$
  From (3) it follows:
  $((c \neq a) \vee (c < a \wedge b \leq d) \vee (d \leq b \wedge a < c) \vee (d = b)) \overset{sat}{\equiv} (d = b)$
  $\Rightarrow (a, b) = (c, d) = d_1 = d_2$

The next paragraph shows that there exists an upper bound of $(D, \trianglerighteq)$.
Let $u = (|L|, |G|)$ with $|L| = 2$ and $|G| = 1$, then u is the upper bound of the partial order D. The CFA represented by u, see Fig. 3.5, has an initial location with one outgoing edge. In a given Domain D there exists an upper bound s.t. $\neg\exists c \in D.c \neq u \wedge u \trianglerighteq c$.

After showing that $(D, \trianglerighteq)$ has a least upper bound, the existence of a least upper bound for every subset is shown.

The least upper bound of $S \subseteq D$ is the element with the minimal amount of locations and edges. Formally it is defined as $s_{max} \in S$, such that $\forall s \in S.\ s \trianglerighteq s_{max}$.

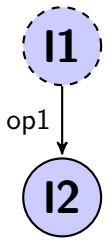


Figure 3.5: Minimal CFA represented by u

The next property which has to been shown is the ascending chain condition. This is done in the following. As shown before every rule decreases the tuple d. Therefore the following is defined as $d_1 = rule_x(d_0)$, $x \in \{sink, sequence, choice, advanced\}$ and $d_n = rule_x(d_{n-1})$, $n \in \mathbb{N}^+$ where d_0 is the initial control flow automaton. The partial order $(D, \trianglerighteq)$ satisfies the ascending chain condition if each ascending chain $d_0 \trianglerighteq d_1 \trianglerighteq ...$ eventually stabilizes, i.e. for some $n \in \mathbb{N}$, $d_n = d_{n+1}$ holds. Finally, the monotonicity of the functions is shown. The monotonicity for every of the four rules is shown in the following paragraph.

- $d \trianglerighteq rule_{sink}(d)$

  Let $d = (|L|, |G|)$ and $d' = (|L'|, |G'|) = rule_{sink}(d)$
  The application of the error sink rule does not change the amount of locations
  $\Rightarrow L' = L$
  $\Rightarrow |L'| = |L|$
  Outgoing edges from all $l_x \in L_E$ are removed
  $G' = G \setminus \{(l_x, op_x, l_y) | l_x \in L_E\}$
  $\Rightarrow G' \subset G$
  $\Rightarrow |G'| \leq |G|$
  $\Rightarrow d \trianglerighteq rule_{sink}(d)$

- $d \trianglerighteq rule_{sequence}(d)$

  Let $d = (|L|, |G|)$ and $d' = (|L'|, |G'|) = rule_{sequence}(d)$
  The application of the sequence rule removes one location l_x
  $\Rightarrow L' = L \setminus \{l_x\}$
  $\Rightarrow |L'| < |L|$
  $\Rightarrow d \trianglerighteq rule_{sequence}(d)$

- $d \trianglerighteq rule_{choice}(d)$

  Let $d = (|L|, |G|)$ and $d' = (|L'|, |G'|) = rule_{choice}(d)$
  The application of the choice rule does not change the amount of locations
  $\Rightarrow L' = L$

$\Rightarrow |L'| = |L|$
All edges between l_x and l_y are replaced by one single edge
$G_{int} = \{\{(l_x, op_x, l_y) | op_x \in Ops\}$
Due to the description of the sequence rule G_{int} has at least two elements.
$G' = G \setminus G_{int}$
$G'' = G' \cup \{(l_x, op_{new}, l_y) | op_{new} = \bigvee \forall op_x \text{ s.t. } (l_x, op_x, l_y) \in G_{int}\}$
$\Rightarrow |G'| < |G|$
$\Rightarrow d \trianglerighteq rule_{choice}(d)$

- $d \trianglerighteq rule_{advanced}(d)$

  Let $d = (|L|, |G|)$ and $d' = (|L'|, |G'|) = rule_{advanced}(d)$
  The application of the advanced sequence rule removes one location l_x
  $\Rightarrow L' = L \setminus \{l_x\}$
  $\Rightarrow |L'| < |L|$
  $\Rightarrow d \trianglerighteq rule_{advanced}(d)$

This section has shown that $(D, \trianglerighteq)$ with $D = (|L|, |G|)$, and $d_1 \trianglerighteq d_2 = (a, b) \trianglerighteq (c, d) \equiv c \leq a \wedge (c < a \vee d \leq b)$ is a complete lattice that satisfies the ascending chain condition and $rule_x$ is a monotonic function.
Thus by the fixpoint theorem of Tarski and Knaster and the given preconditions especially the ascending chain condition large-block encoding will eventually reach a fixpoint and this fixpoint is in fact the least fixpoint.

3.6 Runtime

After explaining the four rules to minimize a CFA, the following paragraph focuses on the runtime for each rule and the large-block encoding overall. The error sink rule is only applied on states which are in L_E. For every location in this set, all outgoing edges are deleted. Therefore one can say, that the runtime is defined as follows:

$$| L_E | \cdot \max(outdegree(l_e)), \forall l_e \in L_E$$

The error sink rule is only applied once at the beginning of large-block encoding. Afterwards the three remaining rules are applied. The sequence rule focuses on the intermediate location l_x. For each outgoing edge of the intermediate location a new edge is added to the CFA. Therefore the runtime of the sequence rule is defined as follows:

$$\max(outdegree(l_x)), \forall l_x \in L$$

The choice rule merges all edges between two given locations. Between two locations are at least k edges where k is the outdegree of the first location l_x. In general the runtime for one application of the choice rule is defined as:

$$\max(outdegree(l_x)), \forall l_x \in L$$

The advanced sequence can also be applied if the intermediate locations has more than one ingoing edge. Therefore, the runtime of one application of the advanced sequence rule gets extended by the indegree of the intermediate location l_x. In general the runtime can be defined as:

$$\max(outdegree(l_x)) \cdot \max(indegree(l_x)), \forall l_x \in L$$

Except for the error sink rule, all other rules can be applied to every location $l_x \in (L \backslash L_E)$. By focusing on a single location all ingoing and outgoing edges have to be taken into account. Therefore the runtime of checking on which location which rule can be applied is done in $\mid G \mid \cdot \mid L \mid$. The maximum runtime of a single rule application is given by the advanced sequence rule with a runtime of $k \cdot m$ where $k = \max(outdegree(l_x), \forall l_x \in L)$ and $m = \max(indegree(l_x), \forall l_x \in L)$. Thus the whole process of large-block encoding is performed in $\mathcal{O}(\mid G \mid \cdot \mid L \mid \cdot k \cdot m)$. One can see that large-block encoding is performed in polynomial time.

4 Implementation

This section provides details about the implementation the large-block encoding in a current verification project. At first the control flow automaton and large-block encoding implementation are explained. Afterwards this section deals with some difficulties which occured during implementation and testing. Following the convention of the project the implementation is done in the functional programming language OCaml. There exists an own module for representing the CFA and performing the large-block encoding.

4.1 Control Flow Automaton Module

The implementation of the control flow automaton module builds up on the graph library of Ocaml. Unused functions of the graph library are not visible in the control flow automaton module. Some functions are edited to make them more efficient for application during the large-block encoding. Functions like indegree or outdegree of a given location are not changed in this implementation, therefore the result of the graph library is immediately forwarded as result of the control flow automaton module. Some other functions are edited and explained in the following list, while the use of these functions will be mentioned in the next section.

- **predecessor_edges**
  This function returns a list of three tuples which contain all ingoing edges of a given location.
- **successor_edges**
  This function returns a list of three tuples which contain all outgoing edges of a given location.
- **find_all_connections**
  Returns an edge label list of all edges between two given locations.

All functions listed above, usually return a list of edges, for the implementation it was decided to change the returned lists of some functions as follows. The functions predecessor_edges and successor_edges return a list of three tuples as follows. At first there is the source location, than the edge label and the last component is the destination location of the edge. The advantages of having a list of three tuples will be discussed in the next section. All functions in the control flow automaton module work on a generic location and edge label type. It is possible to use the same module to work on different edge label types. In this implementation the control flow automaton will work with instructions from a guarded command language as edge labels.

The control flow automaton module provides in detail four different location types. At first there is an initial state in every control flow automaton. Second, a location can be an error location as defined in Sec. 2.1. For optimization of the model checkers there exist stop locations to represent the successful termination of a program. An error location is introduced through the presence of a property to be verified. Such an error state indicates the unsuccessful termination of the program. The fourth location type is a default state which does not represent a termination of the program, these states represent the program counter. The large-block encoding is performed in a separate module, which is explained in the following section.

4.2 Large-Block Encoding Module

The large-block encoding module provides the function to perform large-block encoding on a given control flow automaton. A recursive function performs the large-block encoding until the fixpoint is reached. This function is divided into two parts. The first performs the standard large-block encoding until the fixpoint is reached in this part. After reaching a fixpoint, the advanced sequence rule is applied, if possible. In case the advanced sequence rule was able to do any minimization on the control flow automaton, the two standard rules (sequence and choice rule) are checked for application again. The recursive function stops if the standard rules and the advanced sequence can not be applied any more.

The application of the sequence rule uses the function predecessor_edges to get the ingoing edge of the intermediate location. Of the returning three tuple the function extracts the edge's source location and the edge label. In the following step all outgoing edges of the intermediate location are returned by the successor_edges function of the control flow automaton module. The ingoing edge is connected with each edge in the successor list. Due to the list of three tuples as returned list the destination location and edge label can be extracted quite simple.

The choice rule checks at first if all conditions are satisfied to apply the rule (see Sec. 3.3). Afterwards the function merges all edges between two given location in a single edge. The function find_all_connnections of the control flow automaton module returns a list of all edge labels between two locations. The implementation is able to create a new edge from a list of edge labels, therefore we only need one application of the choice rule to merge edges between two locations.

The advanced sequence rule uses the predecessor_edges and successor_edges function to merge each edge in the list with each edge in the other list. Even in this function the advantage of the three tuples in the list becomes clear. One does not need to transform the return lists any more, one is already able to extract the needed components from the lists and generate a new edge.

4.3 Guarded Command Language

In an early version of the implementation, terms were used as edge labels in the CFA. By developing test cases it became obviously that there are some problems with the propagation of unused variables in an OR statement. By choosing a specific branch of an OR statement it might happen that an assignment of a variable to a new value is skipped. A way to handle this problem is topic of Sec. 4.5. By deciding to extend the project by the counterexample-guided abstraction refinement the weakest preconditions needed to be computed. Therefore it was decided to replace terms as edge labels by the guarded command language. By using the guarded command language one is able to calculate the weakest precondition without using $\forall$ or $\exists$ in the solver queries [JB10]. The guarded command language has the following syntax.

$x := e$	Assign variable x to the value of e
assume b	Only continues if evaluation b is true
$S_1;S_2$	Execute first S_1 then S_2
$S_1 \square S_2$	Execute either S_1 or S_2

Figure 4.1: Guarded Command Language [JB10]

A list of guarded command statements as it can be found in edge labels in our control flow automaton, can be translated into terms such that the bounded model checker is able to check the whole program.

The weakest precondition is build bottom up, syntax driven and automatic calculated. It can be calculated from a given program in guarded command language. For a given postcondition and a statement, the precondition is calculated, such that the postcondition is satisfied after executing the given statement. The calculation of the weakest precondition for a given statement S and a given variable setting Q is shown in the following figure.

S	wp(S,Q)
$x := e$	$Q[e/x]$
assume b	$b \Rightarrow Q$
$S_1;S_2$	$wp(S_1, wp(S_2, Q))$
$S_1 \square S_2$	$wp(S_1, Q) \wedge wp(S_2, Q)$

Figure 4.2: Weakest Preconditions [JB10]

4.4 Variable Substitution

By using large-block encoding for bounded model checking there is one problem that has to be solved.
In case that an edge label is an assignment, it is interpreted as a function
$f : VAR \rightarrow VAR'$ where $var \in VAR$ is the value before executing the assignment

and $var' \in VAR'$ the value afterwards. As long as the edge label contains only one assignment there is no problem. As soon as there are more than one assignments, which is caused by the (advanced) sequence rule of large-block encoding and they use the same variables, a substitution has to be performed. When connecting the assignments with the sequential execution operator ";" without doing a variable substitution the result of the first assignment is not taken into account for the second assignment. Without performing the substitution an abstracted view of the sequential execution looks as follow $f_1(x); f_2(x)$. Both assignment functions use the same value of x without respect to the other function. By performing the substitution one gets an abstracted view of $f_2(f_1(x))$. In the case f_2 uses the result of f_1 as input value for it's assignment. For variable substitution six sets are calculated, for each side there is a set that contains all variables in the formula and for each side a set that contains all primed variables. The fifth set contains variables that are only primed on the left side. The last set contains variables, that are primed on both sides. The pseudo code in Fig. 4.3 shows how the substitution function works.

```
1  sequence term1 term2
2     vars_left := {v | v ∈ term1}
3     vars_right := {v | v ∈ term2}
4     primed_left := {pv | pv' ∈ term1}
5     primed_right := {pv | pv' ∈ term2}
6     primed_onlyleft := primed_left \ primed_right
7     primed_inboth := primed_left ∪ primed_right
8
9     ∀pv ∈ primed_inboth substitute with pv_i in term1
10
11    ∀v ∈ vars_right ∩ primed_inboth substitute with v_i in term2
12
13    ∀v ∈ vars_right ∩ primed_onlyleft substitute with v' in term2
```

Figure 4.3: Pseudo code for variable substitution

The variable substitution while applying the sequence rule is demonstrated on the following example, see Fig. 4.4.

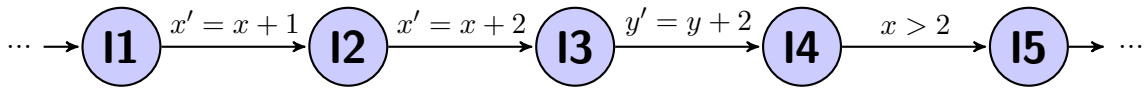


Figure 4.4: Example for variable substitution before Step 1

Step 1: Merge $x' = x + 1$ and $x' = x + 2$
vars_left=$\{x\}$
vars_right=$\{x\}$

primed_left=$\{x\}$
primed_ right=$\{x\}$
primed_onlyleft=$\emptyset$
primed_inboth=$\{x\}$
In this scenario all primed variables on the left side of the formula ($x' = x + 1$) are replaced by a new, unique and primed variable. To ensure that we introduce a new unused variable there is a global index that is incremented after every application of the sequence rule. As new variable name we choose x'_a where a is current value of the global index and x is the primed variable, see Fig. 4.3 line 9. On the right side of the formula all primed variables have to be replaced by the newly introduced variable on the left side, see Fig. 4.3 line 11. After this the assignment on the left side sets the new variable which is used for the assignment on the right side.

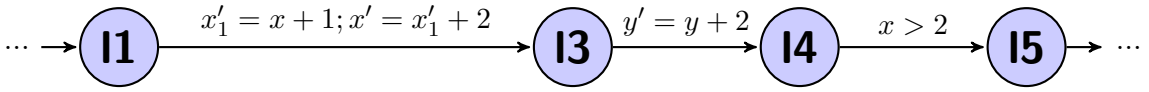


Figure 4.5: Example for variable substitution after Step 1

Step 2: Merge $x'_1 = x + 1; x' = x'_1 + 2$ and $y' = y + 2$
vars_left=$\{x\}$
vars_right=$\{y\}$
primed_left=$\{x\}$
primed_right=$\{y\}$
primed_onlyleft=$\{x\}$
primed_inboth=$\{\}$
In this step there is no substitution necessary. None of the primed variables does occur on the other side.

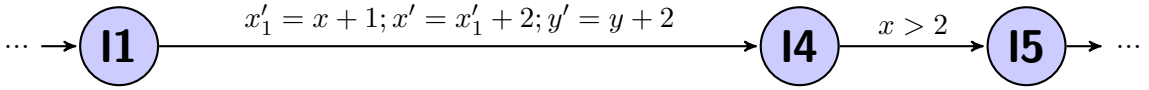


Figure 4.6: Example for variable substitution after Step 2

Step 3: Merge $x'_1 = x + 1; x' = x'_1 + 2; y' = y + 2$ and $x > 2$
vars_left=$\{x\}$
vars_right=$\{x\}$
primed_left=$\{x, y\}$
primed_right=$\{\}$
primed_onlyleft=$\{x, y\}$
primed_inboth=$\{\}$
According to the pseudo code in Fig. 4.3 only line 13 needs to be executed.

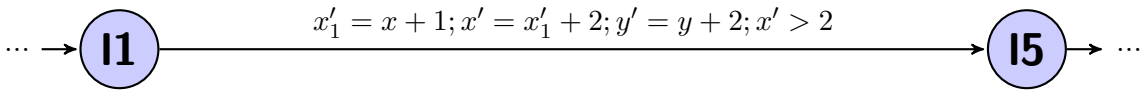


Figure 4.7: Example for variable substitution after Step 3

The cases that are not shown in this example do not require any substitution. In case that only assumptions get connected by the sequential operator, they all work on the same value of the variables. In case that the left side is an assumption and the right side is an assignment there is also no substitution necessary, because the assumption is executed before the assignment.

4.5 Variable Propagation

The previous section pointed out how the edge labels have to be modified when applying the (advanced) sequence rule. Even when applying the choice rule the edge labels have to be modified. To ensure that the model checking does not skip an unused variable by taking a specific branch of OR statement the variables from all other statements have to be propagated. Figure 4.8 shows an example for corrupted dataflow after applying the variable substitution caused by the sequence rule.

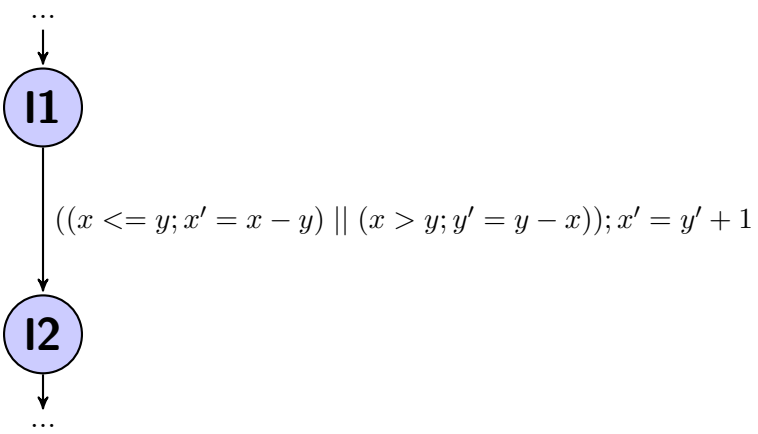


Figure 4.8: Example for corrupted dataflow

By choosing the $x <= y; x' = x - y$ branch the assignment of y to a new values will be skipped by the model checker. The data-flow of the program gets corrupted by choosing a specific path in the control flow automaton, which leads to invalid result of the model checker. In this case the solver can choose an arbitrary value for y' because it is not defined before.

Primed variables of all parallel edge labels are collected in a list. Afterwards each variable which does not occur in the edge label gets propagated into the edge label. Let φ an arbitrary edge label between locations l_x and l_y. Φ is the set of all parallel edges between l_x and l_y. Let $primed\_vars(\varphi) = \{x\}$ and $primed\_vars(\Phi) = \{x, y, z\}$. The

variable propagation is defined by the function: $\Pi : \Phi \to \Phi_{propagated}$ with
$\forall \varphi \in \Phi : \varphi_{propageted} = (\varphi; \{a' = a \mid \forall a \in (primed\_vars(\Phi) \setminus primed\_vars(\varphi)\})$
This is shown in the following example.

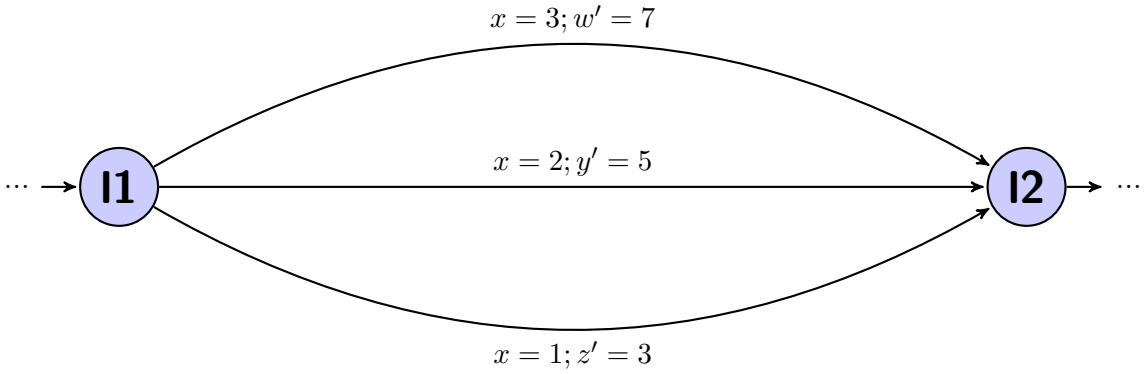


Figure 4.9: Before performing variable propagation

According to the definitions above one gets to following sets:

$$\begin{aligned} \Phi &= \{\varphi, \psi, \vartheta\} \text{ with} \\ \varphi &:= x = 3; w' = 7,\ primed\_vars(\varphi) = \{w\} \\ \psi &:= x = 2; y' = 5,\ primed\_vars(\psi) = \{y\} \\ \vartheta &:= x = 1; z' = 3,\ primed\_vars(\vartheta) = \{z\} \\ \text{thus } primed\_vars(\Phi) &= \{w, y, z\}. \end{aligned}$$

The application of the variable propagation on φ is shown in detail and the results for ψ and ϑ will be given without further details.

$$\begin{aligned} \Pi(\varphi) = \varphi_{propagated} &= (\varphi; \{a' = a \mid \forall a \in (primed\_vars(\Phi) \setminus primed\_vars(\varphi)\}) \\ &= (\varphi; \{a' = a \mid \forall a \in (primed\_vars(w, y, z) \setminus primed\_vars(w)\}) \\ &= (x = 3; w' = 7; \{a' = a \mid \forall a \in \{y, z\}\}) \\ &= (x = 3; w' = 7; y' = y; z' = z) \\ \psi_{propagated} &= (x = 2; y' = 5; w' = w; z' = z) \\ \vartheta_{propagated} &= (x = 3; w' = 7; y' = y; z' = z) \end{aligned}$$

After performing the propagation the example looks as follows:

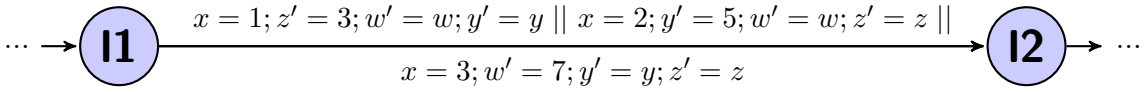


Figure 4.10: After performing variable propagation

After performing the variable substitution and the variable propagation, there are no more conflicts regarding the values of variables.

5 Evaluation

This section provides the evaluation of large-block encoding. In the first subsection single-block encoding and large-block encoding are compared. Afterwards this section focuses on the effect of the summarization. At the end the effect on the model checker's runtime is evaluated.

5.1 Comparison between Single-Block Encoding and Large-Block Encoding

In single-block encoding, there is a one-to-one correspondence between the syntax of the program and the edges/nodes in the ART. By transfering single-block encoding into large-block encoding this one-to-one correspondence is destroyed. In large-block encoding a set of paths between source and target location is represented by a single CFA edge. Thus one ART path corresponds to a set of program paths. The two following observations are induced by these differences.
First, large-block encoding may have substantially smaller ARTs than single-block encoding. The factor of minimization can be up to exponentially. Even the amounts of necessary successor computations and of abstraction refinement steps for infeasible error paths can be reduced significant, with respect to the number of locations. The formula for representing a state are more complex in large-block encoding than in single-block encoding, therefore every single operation is more expensive than before. Depending on the structure of the CFA even huge CFAs can be summarized into small CFAs. The size of large-block encoding CFAs is not influenced by the size of the single-block encoding CFA but more by the structure of the CFA. Therefore one gets more complex formulas which have to be solved by the solver. The efficiency is evaluated in Sec. 5.3.
The second observation is about the summarization of states. In single-block encoding an abstract state can be described as a set or conjunction of predicates. This is precise enough, because an abstract state represents a data region which is reachable by a single program path. Such a program path can be encoded as a conjunction of atomic formulas. In large-block encoding an abstract state represents a data region which is reachable by multiple program paths. Therefore the Cartesian abstraction is too coarse and one uses the more precise Boolean abstraction for representing abstract states. [BCG+09].

5.2 Summarization Performance

Program	|CFA|	| $\mathbf{CFA_{LBE+}}$ |	| $\mathbf{CFA_{LBE}}$ |
Case01	8	4	4
Case02	8	4	4
Case03	5	3	3
Case04	10	4	4
Case05	13	5	6
Case06	19	7	9
Case07	13	4	4
Case08	6	4	4
Case09	5	5	5
Case10	8	4	4
incorrect_gcd_minus.ivl	23	4	5
incorrect_array_record_mix.ivl	19	3	3
incorrect_records_mix.ivl	15	3	3
correct_array_record_mix.ivl	27	3	3
correct_funcalls.ivl	13	3	3
incorrect_sort2_16.ivl	15	3	3
correct_casting.ivl	15	3	3
correct_records_mix.ivl	15	3	3
correct_loop.ivl	11	4	4
incorrect_loop.ivl	11	4	4
correct_sort2_32.ivl	15	3	3
correct_endianness.ivl	8	3	3
incorrect_sort2_32.ivl	15	3	3
test_locks_05.bc	2489	3	3
test_locks_06.bc	3352	3	3
test_locks_07.bc	4343	3	3
test_locks_08.bc	5462	3	3
test_locks_09.bc	6709	3	3
test_locks_10.bc	8804	3	3
test_locks_11.bc	9587	3	3
test_locks_12.bc	11218	3	3
test_locks_13.bc	12977	3	3
test_locks_14.bc	14864	3	3
test_locks_15.bc	16879	3	3

Table 5.1: CFA reduction by large-block encoding

Table 5.1 shows the reduction of the CFA by using large-block encoding. The first column represents the used CFA example. The second column shows the size of the

CFA before applying large-block encoding. The third column contains the size of the CFA after applying large-block encoding including the advanced sequence rule and the last column after applying the standard version of large-block encoding. The maximal minimization factor in this setting was about 6. In case that the advanced sequence rule was applied the minimization factor increases slightly. Due to the fact that the advanced sequence only has noticeable effect in rare occourions it is not perform in every iteration. The application of the advanced sequence rule is checked after the standard rules have been performed. The rest section evaluates whether the time overhead for applying the advanced sequence rule gets outperformed by the time to check the program.

5.3 Performance Evaluation on Model Checkers

Table 5.2 shows the effect of large-block encoding on CEGAR. The times given were in seconds. The tests are divided into several classes. The first class of tests (Case*) are developed during the implementation of the project to check the absence of specific bugs. The second class uses the a method to generate a CFA from IVL Code. These small tests are designed to check the correctness of the translation. The last class of test is designed to cause an expontential blowup od the state space [BCG+09]. The given examples are evaluated on CEGAR exclusively, as the effect on BMC is not constant and largely depends on the memory model. Due to the result of large-block encoding shown in Tab. 5.1 only the standard version of large-block encoding is shown in the table, if not mentioned differently. As expected, the runtimes of the advanced large-block encoding version are slightly higher than for the normal version of large-block encoding. As mentioned before the advanced large-block encoding does not have effect on most of the examples. For the first three examples of the test_locks series large-block encoding does not have positive effect. One reason for this might be the complexity of the formulas. In such cases the solver can use less time to solve a certain amount of simple formulas instead of solving only a few complex formulas. For the following seven examples a positive effect can be observed. The saved time increases with the amount of locations in the single-block encoding CFA. The best result can be found in the last example. On average over the whole test cases for test_locks* large-block encoding achieves a time saving of 20%. By using large-block encoding one reduces the search for counterexample paths, but creates more complex queries for the solver. Therefore the total runtime increases with a smaller factor.

Program	LBE	CEGAR LBE	CEGAR SBE	Σ LBE	Saving
Case01	0.0027	0.1875	0.3879	0.1902	49%
Case02	0.0003	0.3874	1.8821	0.3877	21%
Case03	0.0003	0.0424	TO	0.0427	-
Case04	0.0004	TO	0.2395	TO	-
Case05	0.0004	TO	TO	TO	-
Case05 LBE+	0.0009	TO	TO	TO	-
Case06	0.0007	8.7159	TO	8.7166	-
Case06 LBE+	0.0016	11.3575	TO	11.3591	-
Case07	0.0005	TO	4.4192	TO	-
Case08	0.0002	TO	TO	TO	-
Case09	0.0001	1.6386	3.0652	1.6387	53%
Case10	0.0007	2.9593	3.9136	2.9600	76%
incorrect_gcd_minus.ivl	0.0018	11.2494	TO	11.2513	-
incorrect_array_record_mix.ivl	0.0002	0.0086	0.0734	0.0089	12%
incorrect_records_mix.ivl	0.0025	0.0501	TO	0.0526	-
correct_array_record_mix.ivl	0.0025	TO	TO	TO	-
correct_funcalls.ivl	0.0005	TO	TO	TO	-
incorrect_sort2_16.ivl	0.0002	0.0050	0.0081	0.0052	64%
correct_casting.ivl	0.0066	TO	TO	TO	-
correct_records_mix.ivl	0.0002	TO	TO	TO	-
correct_loop.ivl	0.0002	TO	TO	TO	-
incorrect_loop.ivl	0.0004	TO	TO	TO	-
correct_sort2_32.ivl	0.0007	TO	79.7497	TO	-
correct_endianness.ivl	0.0003	0.6167	0.8597	0.6170	72%
incorrect_sort2_32.ivl	0.0000	0.0224	0.0144	0.0224	-155%
test_locks_5.c	3.10	0.27	3.27	3.37	-3%
test_locks_6.c	5.17	0.52	5.22	5.69	-9%
test_locks_7.c	9.18	0.71	9.61	9.89	-3%
test_locks_8.c	15.76	1.05	17.49	16.81	4%
test_locks_9.c	22.76	1.51	27.81	24.27	13%
test_locks_10.c	36.05	2.21	43.04	38.26	11%
test_locks_11.c	48.87	2.91	63.6	51.78	19%
test_locks_12.c	69.46	3.87	92.54	73.33	21%
test_locks_13.c	98.35	5.48	130.25	103.83	20%
test_locks_14.c	129.91	6.93	174.19	136.84	21%
test_locks_15.c	165.05	9.26	235.09	174.26	26%

Table 5.2: Effect of large-block encoding on model checkers

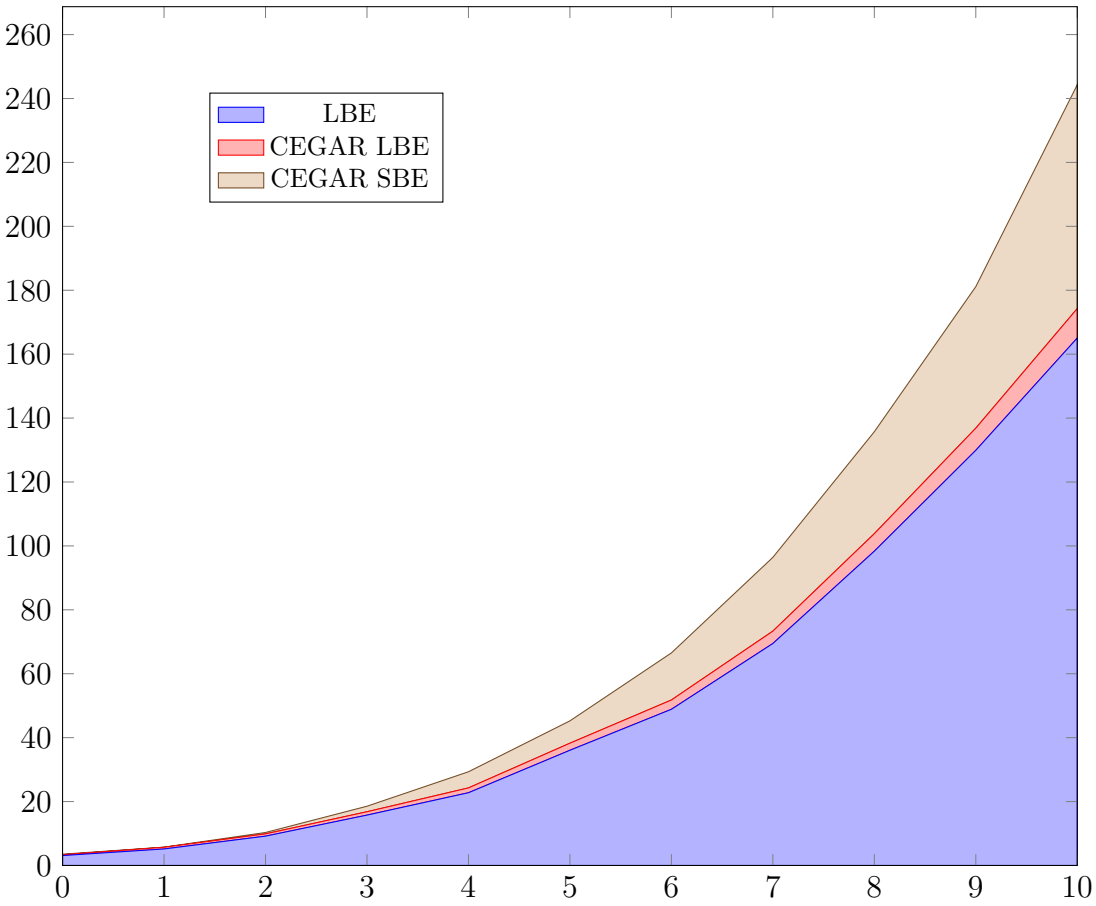


Figure 5.1: CEGAR runtime on test_locks*

Figure 5.1 shows the runtime of CEGAR on the third class of the test cases (test_locks*.c). The beige zone illustrates the CEGAR runtime with single-block encoding. The blue area is the runtime of large-block encoding and red represents total runtime of CEGAR with large-block encoding. The beige area gets bigger with more complex programs (see Tab. 5.1). The relatively long runtime of large-block encoding and the runtime of CEGAR are in total significant lower than the runtime of CEGAR with single-block encoding.

6 Conclusion

The aim of this thesis was to reduce the runtime of model checking in particular CEGAR. Therefore large-block encoding was introduced as alternative to single-block encoding. The standard version of large-block encoding contains three rules, as extension to this version a new rule, the advanced sequence rule, was introduced in this thesis. The advanced sequence rule does not have such strict restrictions than the standard sequence rule. Du to this the original fixpoint proof was not suitable for the new version of large-block encoding, therefore it was proven that advanced large-block encoding will also eventually reach a fixpoint. In single-block encoding there exists an one-to-one correspondence between edge and syntax of the program code. After applying the rules of large-block encoding there is no one-to-one correspondence overall any more. In large-block encoding edges are summarized in a single one and one edge corresponds to a set of edges in the single-block encoding CFA. Therefore large-block encoding needs a more expensive and precise abstraction than single-block encoding.
One can not say, that large-block encoding has a positive effect on programs in general but rather depends on the complexity of the formulas in the program. Section 5.3 provides some examples where large-block encoding is not faster than single-block encoding, but the time saving of up to 25% in the examples shows that large-block encoding is usable for many programs. The negative results on comparable small programs can be neglected, because real world programs are usually quite large. Therefore the possibility of negative effects in real world applications is very low, which makes large-block encoding attractive.

In future work one could analyse how a pre-selection of locations for each iteration of large-block encoding would perform. A pre-selection of locations could be executed in a way such, that not every location has to be checked for a rule application, because some locations might not satisfy any pre-conditions for the standard rules. Another optimization could be the adaptation of a a the work list algorithm for summarizing single-block encoding CFAs into large-block encoding CFAs.

Bibliography

[BCC+03] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Advances in Computers*, 58:117–148, 2003.

[BCG+09] Dirk Beyer, Alessandro Cimatti, Alberto Griggio, M. Erkan Keremoglu, and Roberto Sebastiani. Software model checking via large-block encoding. In *Proceedings of 9th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2009, 15-18 November 2009, Austin, Texas, USA*, pages 25–32, 2009.

[BHJM07] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker blast. *STTT*, 9(5-6):505–525, 2007.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008.

[BKW10] Dirk Beyer, M. Erkan Keremoglu, and Philipp Wendler. Predicate abstraction with adjustable-block encoding. In *Proceedings of 10th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2010, Lugano, Switzerland, October 20-23*, pages 189–197, 2010.

[BPR03] Thomas Ball, Andreas Podelski, and Sriram K. Rajamani. Boolean and cartesian abstraction for model checking C programs. *STTT*, 5(1):49–58, 2003.

[BR02] Thomas Ball and Sriram K. Rajamani. The SLAM project: debugging system software via static analysis. In *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002*, pages 1–3, 2002.

[CGJ+00] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, pages 154–169, 2000.

[HJMS02] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002*, pages 58–70, 2002.

[Hoa69] Charles A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[JB10] Ivan Jager and David Brumley. Efficient directionless weakest preconditions. Technical Report CMU-CyLab-10-002, Carnegie Mellon University CyLab, February 2010.

[KS08] Daniel Kroening and Ofer Strichman. *Decision Procedures: An Algorithmic Point of View*. Springer Publishing Company, Incorporated, 1 edition, 2008.

[LNO06] Shuvendu K. Lahiri, Robert Nieuwenhuis, and Albert Oliveras. SMT techniques for fast predicate abstraction. In *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, pages 424–437, 2006.

[McM06] Kenneth L. McMillan. Lazy abstraction with interpolants. In *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, pages 123–136, 2006.

[Nol12] Thomas Noll. Static program analysis. RWTH Aachen, 2012.

[Tar55] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955.