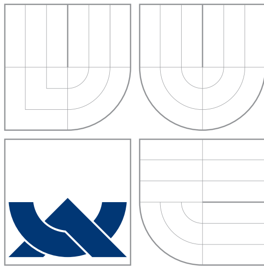
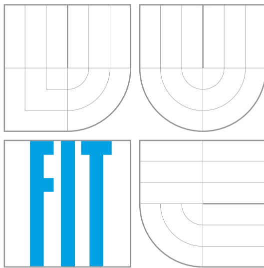


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

VERIFIKACE UKAZATELOVÝCH PROGRAMŮ POMOCÍ LESNÍCH AUTOMATŮ

VERIFICATION OF POINTER PROGRAMS BASED ON FOREST AUTOMATA

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. MARTIN HRUŠKA
AUTHOR

VEDOUCÍ PRÁCE — Mgr. LUKÁŠ HOLÍK, Ph.D.
SUPERVISOR

BRNO 2015

Abstrakt

V této práci je rozvíjena existující metoda pro shape analýzu programů založená na lesních automatech. Dále je také vylepšována implementace této metody, nástroj Forester. Lesní automaty jsou založeny na stromových automatech, jejichž jednoduchou implementaci Forester obsahuje. Prvním přínosem této práce je nahrazení této implementace knihovnou VATA, která obsahuje efektivní algoritmy pro reprezentaci a manipulaci stromových automatů. Verze nástroje Forester používající knihovnu VATA se zúčastnila mezinárodní soutěže SV-COMP 2015. Dále je verifikace založená na lesních automatech v této práci rozšířena o predikátovou abstrakci a analýzu nalezených protipříkladů. Výsledek této analýzy je možné využít následujícími způsoby. Prvním je určení toho, zda je nalezené chyba reálná nebo naopak nepravá. Druhým je pak zjemnění predikátové abstrakce pomocí predikátů odvozených při zpětném běhu. Obě techniky byly také implementovány v nástroji Forester. Na závěr je zhodnoceno zlepšení, které tyto techniky přinesly oproti původní verzi nástroje Forester.

Abstract

In this work, we focus on improving the forest automata based shape analysis implemented in the Forester tool. This approach represents shapes of the heap using forest automata. Forest automata are based on tree automata and Forester currently has only a simple implementation of tree automata. Our first contribution is replacing this implementation by the general purpose tree automata library VATA, which contains the highly optimized implementations of automata operations. The version of Forester using the VATA library participated in the competition SV-COMP 2015. We further extended the forest automata based verification method with two new techniques — a counterexample analysis and predicate abstraction. The first one allows us to determine whether a found error is a real or spurious one. The results of the counterexample analysis is also used for creating new predicates which are used for the refinement of predicate abstraction. We show that both of these techniques contribute to an improvement over the early approach.

Klíčová slova

lesní automaty, formální verifikace, statická analýza, složité datové struktury, stromové automaty, zpětný běh, predikátová abstrakce.

Keywords

forest automata, formal verification, static analysis, complex data structures, tree automata, backward run, predicate abstraction.

Citace

Martin Hruška: Verification of Pointer Programs Based on Forest Automata, diplomová práce, Brno, FIT VUT v Brně, 2015

Verification of Pointer Programs Based on Forest Automata

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Mgr. Lukáše Holíka, Ph.D.

....
Martin Hruška
27. května 2015

Poděkování

Rád bych poděkoval vedoucímu této práce, Mgr. Lukáši Holíkovi, Ph.D., za odborné rady a vedení při tvorbě této práce. Dále bych rád poděkoval Ing. Ondřeji Lengálovi za četné konzultace, během kterých mě trpělivě seznamoval s nástrojem Forester, Prof. Ing. Tomáši Vojnarovi, Ph.D., a Mgr. Adamu Rogalewiczovi, Ph.D., za rady poskytnuté během konzultací. Také děkuji Ing. Tomáši Fiedorovi za cenné připomínky k finální podobě technické zprávy.

© Martin Hruška, 2015.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Contents

1 Introduction **3**

2 Preliminaries **5**
2.1 Graphs, Trees and Forests 5
2.1.1 Graphs and Trees 5
2.1.2 Forests 6
2.1.3 Graphs and Forests with Ports 7
2.1.4 Minimal and Canonical Forest 8
2.1.5 Tree Automata 9
2.1.6 Forest Automata 9
2.2 Forest Automata of Higher Level 10
2.2.1 Structured Labels 10
2.2.2 Tree Automata over Structured Labels 11
2.2.3 Forest Automata of Higher Level 11

3 Forest Automata based Verification **13**
3.1 Heap Representation 13
3.2 Symbolic Execution 14
3.2.1 Normalization 15
3.2.2 Boxes 16
3.2.3 Abstraction over Forest Automata 16

4 The VATA Library and Forester **18**
4.1 The VATA Library 18
4.1.1 Design 18
4.1.2 Implemented NTA Encodings 19
4.2 Forester 21
4.2.1 Design 21
4.2.2 Implementation of Forest Automata Concepts 23
4.2.3 The Microcode Instructions 24

5 Forester with VATA **31**
5.1 Refactoring of Forester 31
5.2 Adapter Design Pattern 32
5.3 Implementation 32

6 Backward Run and Predicate Abstraction in Forester 34
6.1 Backward Run and Predicate Abstraction . . . 34
6.2 Intersection of Forest Automata . . . 36
6.3 Designing Backward Run over Symbolic States . . . 38
6.4 Designing Predicate Abstraction over Forest Automata . . . 41
6.5 Implementation . . . 41
6.5.1 Backward Run . . . 41
6.5.2 Intersection of Forest Automata . . . 42
6.5.3 Predicate Abstraction . . . 42

7 Experimental Evaluation 43
7.1 The Evaluation of Forester with VATA . . . 43
7.2 Backward Run Evaluation . . . 45
7.3 Predicate Abstraction . . . 45

8 Conclusion 48

A Storage Medium 51

Chapter 1

Introduction

The growing number of applications of computer programs in the past few decades brings the need for their greater safety and security. But it is not an easy task to guarantee that software has the specified properties. The programs often go through many states during the computations and it could be very time and space consuming or even impossible to check whether no undesirable behavior may appear in any of the program runs. One of the approaches to ensure the software quality is *testing* (and dynamic analysis) which is based on running the program in different contexts with different inputs and matching a program behavior and the outputs with the expected ones. This method can satisfy the most of the software quality requirements and often covers the most of the program behavior. On the other side, it is only possible to find the errors using testing, not to prove their absence [7]. Finding some of the errors during testing also does not imply the elimination of all of them.

The mentioned weakness of testing can be resolved by *formal verification*. Formal verification aims at using rigorous mathematical methods to check whether a given system meets a given specification [30]. There are three main branches of formal verification. The first one is *model checking* which systematically explores the state space of a system (e.g., a program) or its model to prove that it satisfies the verified property. The second one is *static analysis* which is performed over a source code (or some modification of it) of a system. It is done without a need of its explicit execution and it rather explores a syntactic structure of the analysed program. One of the important and very widely used approaches in static analysis is called *abstract interpretation* where the analysis is performed by applying abstract transformers corresponding to the original program semantics in an abstract domain. Abstract interpretation evolved from static analysis but since it deals with semantics of programs, and since later approaches to model checking use abstraction, boundaries of the two are rather unclear. The last approach is *theorem proving*. It proves the program safety in the standard mathematical way — starting from axioms and using inference rules to verify properties of the given system. Theorem proving can be partially automated.

This work deals with a specific branch of static analysis called *shape analysis* which focuses on verification of programs manipulating complex dynamic data structures (such as different kinds of lists and trees), typically allocated on the heap. The checked properties are for example absence of invalid pointer dereference, that no invalid pointer is freed (no invalid free) or that all allocated memory is freed during the program execution (no memory leaks). There are different approaches to shape analysis with different advantages. The approaches based on *separation logic* [27, 2] provide great scalability. On

the other hand, automata based approaches, particularly *abstract regular tree model checking* (ARTMC) [3], are superior in their flexibility and generality (with the exception of the most recent separation logic based methods [18]). This work focuses on a verification procedure based on the concept of forest automata (FA) which combines the benefits of the both mentioned approaches [10].

Forest automata have been introduced in [10] and they are an extension of finite tree automata (TA). They are used as an abstract domain in a verification procedure which performs symbolic execution of the analyzed program. A prototype of this verification procedure has been implemented as the *Forester* tool [23]. Forester verifies programs written in C and it can detect safety violations such as invalid dereferences, invalid frees, memory leaks, and also reachability of an error line. It is capable of verifying non-trivial data structures such as skip-lists of the second and the third level. However, the current implementation is far from perfect. For instance, Forester currently does not fully support the complete syntax of C and it also does not check whether a found error is real or spurious. A general goal of this work is to improve Forester in the areas described further.

One of the problems of Forester is low modularity and maintainability of the code. The first goal of this thesis is to improve it by replacing the underlying implementation of finite tree automata in Forester by the VATA library — a library for manipulation with TA [20]. The VATA library implements very efficiently the most of state-of-the-art algorithms for TA. It is better to implement the algorithms efficiently with implementation optimizations only at one place (in VATA) than putting the effort to optimizing the implementation of the same algorithms in Forester again. The library replacement so improves the *maintainability* of Forester. Having the special encapsulated module for tree automata with a clear interface brings more modularity by decreasing dependencies to the internals of the module. Creating an interface between Forester and VATA will require refactoring of Forester.

Another weakness of Forester is the missing of the analysis of detected errors. Since Forester uses abstraction to deal with unboudness of dynamic data structures a found error can be spurious which is caused by overapproximating the set of reachable heap configurations by abstraction. When this happens, the analysis of the detected error could determine whether the error is a real or not. The results of the counterexample analysis could be also used for refinement of the abstraction in the case the counterexample is spurious. After the refinement, the analysis could be restarted with the refined abstraction to avoid the same run leading to the spurious counterexample. This gradual refinement of abstraction is called *counterexample-guided abstraction refinement* (CEGAR) [4]. The second goal of this thesis is to design and implement the analysis of the counterexamples and a refinement of the abstraction (when a spurious error is found) for forest automata based verification. The method for analysing counterexamples is in this case *backward run* and its results will be further used for refinement of *predicate abstraction.* Forester does not currently use predicate abstraction but *height abstraction* because predicate abstraction needs backward run for creating the new predicates for refinement. Height abstraction is less precise and less flexible compared to predicate abstraction. Using predicate abstraction would allow to analyse even more complex data structures such as red-black lists.

The outline of this master thesis is following. In Chapter 2, the preliminaries are given. The verification procedure based on FA is covered in Chapter 3. Chapter 4 provides description of the VATA library and Forester and Chapter 5 describes the implementation of the version of Forester tool using the VATA library. The design and the implementation of backward run and predicate abstraction is described in Chapter 6. Finally, Chapter 7 contains an overview of experimental evaluation and Chapter 8 summarizes this thesis.

Chapter 2

Preliminaries

This chapter provides the theoretical foundations for this thesis. First, graphs, trees and forests are defined along the automata accepting them in Section 2.1. Then the previously defined concepts are further extended to hierarchical forests and automata in Section 2.2. This section follows the definitions and the structure used in [13].

2.1 Graphs, Trees and Forests

Assume a alphabet Σ and a word $w = a_1 \cdots a_n$, we denote the i-th symbol of w from Σ as a_i. We denote $dom(f)$ the domain of a total mapping $f : A \to B$ and its range is denoted by $rng(f)$.

2.1.1 Graphs and Trees

A *ranked alphabet* is a finite set of symbols Σ and a related mapping $\# : \Sigma \to \mathbb{N}$ assigning to a symbol its rank. A (directed, ordered, labelled) *graph* is a total map $g : V \to \Sigma \times V^*$ where V is a finite set of nodes. The items of the set Σ are in context of the graphs called *labels*. The map g maps each node $v \in V$ to:

1. a label $\alpha \in \Sigma$ that we denote by $l_g(v)$,
2. a sequence of *successors* $(v_1 \cdots v_n) \in V^n$ for $n \in \mathbb{N}$. We denote successors by $S_g(v)$ and v_i is denoted by $S^i_g(v)$.

Symbol $l_g(v)$ is such that $\#(l_g(v)) = |S_g(v)|$. We will omit the subscript g when no ambiguity may arise possible.

A *leaf* of g is a node $v \in V$ such that $S_g(v) = \epsilon$. An *edge* of g is a pair $v \mapsto (a, v_1 \cdots v_n)$ where $v, v_1, \ldots, v_n \in V$, $a \in \Sigma$ such that $g(v) = (a, v_1 \cdots v_n)$. The *in-degree* of a node $v' \in V$ in the graph g is the sum of its occurrences in the tuples $t \in V^*$ get by $g(v)$ for all $v \in V$. We denote the in-degree of a node $v \in V$ by $idg_g(v)$ and again we omit the subscript g whenever it is possible. More formally, in-degree is defined as $idg(v') = |\{(v \mapsto (a, v_1 \cdots v_n), i)) \,|\, v \mapsto (a, v_1 \cdots v_n)$ is an edge such that $i \in \{1, \ldots, n\} : v' = v_i\}|$. The *joins* of g are nodes $v \in V$ such that $idg(v') > 1$.

A *path* from $v \in V$ to $v' \in V$ is a sequence $p = v_0, i_1, v_1, \ldots, i_n, v_n$ where $v = v_0, v' = v_n$ and $\forall j \in \{1, \ldots, n\} : v_j = S^{i_j}(v_{j-1})$ (informally, v_j is the i_j-th successor of v_{j-1}). The empty path has $n = 0$. The path p has *length* n denoted by $length(p) = n$. The path $p = u_0, i_1, u_1, \ldots, i_n, u_n$ is acyclic if $\forall u_i, u_j \in p : i \neq j \Rightarrow u_i \neq u_j$. The *cost* of the acyclic

path p is the sequence $i_1, \dots, i_n$. The path p is *cheaper* than path p' iff the cost of p is lexicographically smaller than that of p'. A node $u \in V$ is *reachable* from a node $v \in V$ iff there exists a path from v to u or $u = v$. A node $u \in V$ is a *root* of the graph g iff all nodes $v \in V$ are reachable from u. We use the term *root* also for a mapping $root : g \to V$ which maps a graph to its root. A graph with the root is called *rooted*.

A *tree* t is a graph which is either empty, or it has exactly one root and $\forall v \in V : idg(v) \leq 1$ (informally, each node is a successor of at most one of the other nodes).

Example 2.1.1. *We illustrate some terms related to the graphs and trees. Consider a graph t in Figure 2.1 with $V = \{v_1, v_2, v_3, v_4, v_5\}$ and the alphabet $\Sigma = \{a, b\}$ with a ranking function $\#$ such that $\#(a) = 2$ and $\#(b) = 0$. Then the graph t is the mapping $\{t(v_1) \mapsto (a, (v_2, v_3)),\ t(v_2) \mapsto (b, ()),\ t(v_3) \mapsto (a, (v_4, v_5)),\ t(v_4) \mapsto (b, ()),\ t(v_5) \mapsto (b, ())\}$. The node v_2, v_4, v_5 are leaves. An example of an edge is $v_1 \mapsto (a, (v_2, v_3))$. A path is for instance the sequence $v_1, 2, v_3, 2, v_5$ what is the path from v_1 to v_5. Thus the node v_5 is reachable from the node v_1. Since all nodes are reachable from v_1 then v_1 is a root of t. No node has more then one incoming edge, hence t is a tree.*

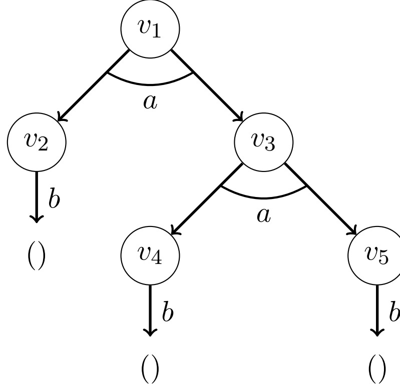


Figure 2.1: A graph t that has attributes of a tree.

2.1.2 Forests

Without the loss of generality suppose that $\Sigma \cap \mathbb{N} = \emptyset$. A Σ-labelled *forest* is a sequence of trees $t_1 \cdots t_n$ over $(\Sigma \cup \{1, \dots, n\})$ where $\forall i \in \{1, \dots, n\} : \#i = 0$. We suppose that the sets of nodes of the trees $t_1, \dots, t_n$ are disjoint. *Root references* are leaves labelled by $i \in \mathbb{N}$. The forest $t_1 \cdots t_n$ represents the graph $\otimes t_1 \cdots t_n$ that arises by interconnecting roots by the related root reference. For instance, a root reference 2 in t_1 would be replaced by the root node of t_2. Let us formalize the idea of the construction of $\otimes t_1 \cdots t_n$. The graph $\otimes t_1 \cdots t_n$ contains an edge $v \mapsto (a, v_1 \cdots v_m)$ iff $\exists i \in \{1, \dots, n\} : \exists (v \mapsto (a, v'_1 \cdots v'_m)) \in edges(t_i) : \forall j \in \{1, \dots, m\} : v_j = h(v'_j)$ where $edges(t_i)$ is the set of all edges of the tree t_i and

$$h(v'_j) = \begin{cases} root(t_k) & \text{if } v'_j \text{ is a root reference with } l(v'_j) = k \\ v'_j & \text{otherwise.} \end{cases}$$

Example 2.1.2. *We illustrate the notion of forest. Consider a forest f in Figure 2.2. It consists of three trees: t_1 with the root u_1, t_2 with the root v_1, and t_3 with the root w_1. The alphabet Σ of the trees is the same as in Example 2.1.1 but f is defined over $\Sigma \cup \{\overline{2}, \overline{3}\}$ where $\overline{2}, \overline{3}$ denotes the root references to the roots u_5 and v_3 of the second and the third tree.*

We can obtain a graph $\otimes t_1, t_2, t_3$ shown in Figure 2.3 from t_1, t_2, t_3. The edges labelled by root references are replaced by the edges leading to the roots of the referenced trees.

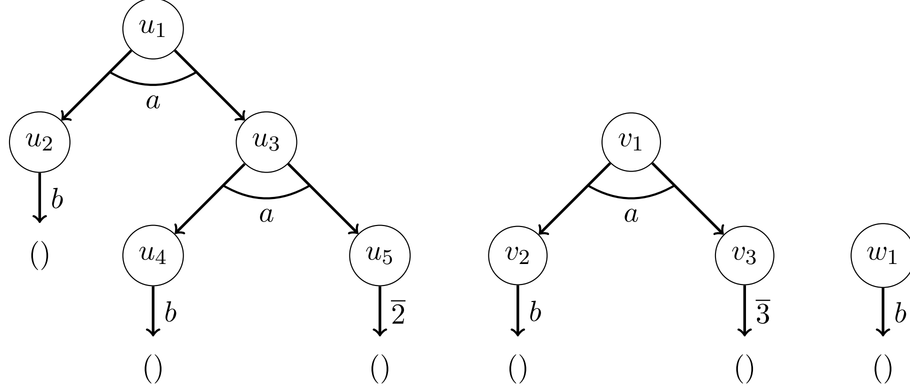


Figure 2.2: A forest f consisting 3 trees t_1, t_2, t_3 with roots u_1, v_1, w_1.

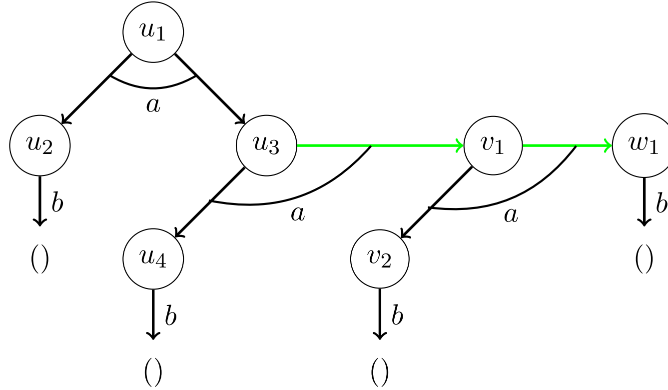


Figure 2.3: The graph $\otimes t_1, t_2, t_3$ obtained from the forest f in Figure 2.2. The green edges are the ones that were added to create the graph from the forest f.

2.1.3 Graphs and Forests with Ports

We extend the notion of graphs and forests with *input* and *output* nodes which are marked by so-called *ports*. An *input-output-graph* (io-graph) is a pair (g, ϕ) (for brevity denoted as g_ϕ) where g is a graph and $\phi = (\phi_1 \cdots \phi_n) \in dom(g)^+$ is a sequence of ports, ϕ_1 is an input port and $\phi_2 \cdots \phi_n$ is a sequence of output ports. The ports are unique in ϕ. The graph g_ϕ is called *accessible* if its root is the input port ϕ_1.

The set of *cut-points* $cps(g_\phi)$ of a graph g_ϕ consists of its ports and joins. Formally, $cps(g_\phi) = \{v \in V \mid v \ \in \phi \vee idg(v) > 1\}$.

An *io-forest* is a pair $f = (t_1 \cdots t_n, \pi)$ such that $n \geq 1$ and $\pi \in \{1, \dots, n\}^n$ is a sequence of port indices, where π_1 is an index of input port and $\pi_2 \dots \pi_{|\pi|}$ is a sequence of the indices of the output ports. As in the case of ports, the indices are unique. The io-graph $\otimes f$ is constructed from a forest f such that $\otimes f = (\otimes t_1 \cdots t_n, root(t_{\pi_1}), \dots, root(t_{\pi_n}))$. The ports of $\otimes f$ are the roots of the trees indexed by the indices in π. This means that a node of the graph pointed by π_1 is an input port of $\otimes f$. The nodes (that were the roots of the trees) pointed by the output port indices in π are the output ports of $\otimes f$.

Example 2.1.3. *Recall the graph t from Figure 2.1. It can be extended to an io-graph t_ϕ by adding ports $\phi = (v_1, v_4, v_5)$. The resulting io-graph t_ϕ has the input port v_1 and the output ports v_2, v_3. Because v_1 is the root, the graph t_ϕ is accessible. The cut-points of t_ϕ are v_1, v_4, v_5.*

In Figure 2.2, we shown the forest f. This forest could be extended to an io-forest $f_{io} = ((t_1, t_2, t_3), \pi)$ by defining a sequence of the port indices π which could be e.g., $(1, 3)$. Then the graph $\otimes f_{io}$ is a pair $(\otimes(t_1, t_2, t_3), (u_1, w_1))$. The graph $\otimes(t_1, t_2, t_3)$ is the same as in Figure 2.3. The input port u_1 is indeed $root(t_1)$ and the output port w_1 because it is the root of the third tree in f and the index of the output port is 3.

2.1.4 Minimal and Canonical Forest

There are two properties of the forests, minimality and canonicity, that we will further use. The properties make it possible to represent an io-forest in a unique way. An io-forest $f = (t_1 \cdots t_n, \phi)$ representing a graph $\otimes f$ is *minimal* iff the roots of the trees $t_1, \dots, t_n$ correspond to the cut-points of $\otimes f$, so there exists a bijection between $\{root(t_k) \mid t_k \in \{t_1, \dots, t_n\})\}$ and $cps(\otimes f)$. The minimal io-forest is hence a unique representation of $\otimes f$ up-to to permutations of $t_1, \dots, t_n$.

To be able to define truly canonicity representation of the forest f we define an ordering $\preceq_p$, so called *canonical ordering*, of the cut-points of $\otimes f$. The canonical ordering $\preceq_p \subseteq cps(\otimes f) \times cps(\otimes f)$ is defined as follows: $c_1 \preceq_p c_2 \Leftrightarrow$ the cost of the cheapest path from ϕ_1 (input port) to c_1 is *smaller* than the cost of the smallest path from ϕ_1 to c_2. The io-forest f_c is *canonical* iff it is minimal, the trees $t_1, \dots, t_n$ are ordered by $\preceq_p$, and $\otimes f$ is accessible. The canonical io-forest is then a unique representation of an accessible io-graph. The canonical io-forest can be obtained by a depth-first traversal (DFT)[17] of $\otimes f$. To make DFT traversal deterministic and hence the ordering of the trees unique we need to assume that there is an ordering $\leq_\Sigma$ over labels Σ of $\otimes f$. The DFT then uses a stack of nodes initialized with the input and output ports ordered by $\preceq_p$ with the smallest node on the top of the stack. The DFT is run over $\otimes f$. It visits the successors tuples of a node in the order given by $\leq_\Sigma$ and the nodes in successors tuples are in their order in tuple. We obtain canonical forest f_c where trees are in the following order. The first ones in canonical ordering are trees corresponding to the ports of f ordered by $\preceq_p$ and the rest of the trees is in the canonical ordering of the trees which is defined by the order of the DFT traversal visit.

2.1.5 Tree Automata

A (finite, non-deterministic, top-down) *tree automaton* (TA) is a quadruple $A = (Q, \Sigma, \Delta, R)$ where

- Q is a finite set of *states*,
- Σ is a ranked alphabet,
- Δ is a set of *transition rules* where transitions have a form $(q, a, q_1 \cdots q_n)$ where $q, q_1, \ldots, q_n \in Q$, $a \in \Sigma$, $n \geq 0$ and $\#a = n$. Alternatively, we write $q \xrightarrow{a} (q_1 \cdots q_n)$ to denote that $(q, a, q_1 \cdots q_n) \in \Delta$. The rule is called *leaf rule* when $n = 0$,
- $R \subseteq Q$ is a set of *root states*.

We can symmetrically define bottom-up tree automata as a quadruple $B = (Q, \Sigma, \Delta, F)$ where

- Q is a finite set of states,
- Σ is a ranked alphabet,
- Δ is a set of transition rules where transition has a form $(q_1 \cdots q_n, a, q)$ where q,$q_1, \ldots, q_n \in Q$, $a \in \Sigma$, $n \geq 0$ and $\#a = n$. We can again write $(q_1 \cdots q_n) \xrightarrow{a} q$ to denote a $(q_1 \cdots q_n, a, q) \in \Delta$. For $n = 0$ we call the rule *leaf rule*,
- $F \subseteq Q$ is a set of *final states*.

We further consider top-down tree automata unless stated otherwise.

Their semantics of TA is defined following as follows. A *run* of A over a tree t is mapping $\rho : dom(t) \to Q$ such that $\forall v \ \in dom(t)\ \exists q \xrightarrow{a} (q_1 \cdots q_n) \in \Delta : q = \rho(v) \wedge \forall i \in \{1, \ldots, |S(v)|\} : q_i = \rho(S(v)_i)$. We use $t \Rightarrow_\rho q$ to denote that ρ is a run of A over a tree t s.t. $\rho(root(t)) = q$ and we use $t \Rightarrow q$ to denote that exists $t \Rightarrow_\rho q$. The *language* of a state $q \in Q$ is defined as $L(q) = \{t \,|\, t \Rightarrow q\}$ and the *language* of A is defined as $L(A) = \bigcup_{q \in R} L(q)$.

Example 2.1.4. *Consider a TA* $A = (Q, \Sigma, \Delta, R)$ *where* $Q = \{q_1, q_2, q_3, q_4, q_5\}$, $\Sigma = \{a, b\}$, *such that* $\#(a) = 2, \#(b) = 0$, $R = \{q_1\}$, *and* $\Delta = \{q_1 \xrightarrow{a} (q_2, q_3), q_2 \xrightarrow{b} (), q_3 \xrightarrow{b} (q_4, q_5), q_4 \xrightarrow{b} (), q_4 \xrightarrow{b} ()\}$. *Then the map* ρ *such that* $\forall i \in \{1, \ldots, 5\} : \rho(v_i) = q_i$ *is a run A over the tree t from Figure 2.1. Since* $\rho(root(t)) = \rho(v_1) = q_1$ *then* $t \in L(q_1)$ *and because* $q \in R$ *it also holds* $t \in L(A)$.

2.1.6 Forest Automata

A *forest automaton* (FA) over alphabet Σ is a pair $F = (A_1 \cdots A_n, \pi)$ where $A_1 \cdots A_n$ is a sequence of tree automata defined over the alphabet $\Sigma \cup \{1, \ldots, n\}$ and $\pi = I_1 \cdots I_n$, where $I_1, \ldots, I_n \in \{1, \ldots, n\}$ is a sequence of port indices. There are two kinds of languages related to FA. The first one is a forest language obtained by Cartesian product of the languages of particular TA (and port indices) of FA and hence the forest language is a set of the io-forests. The other is the graph language obtained by connecting the io-forests from the forest language to io-graphs. Formally, the *forest language* of the FA F is the set

of io-forests $L_f(F) = L(A_1) \times \ldots \times L(A_n) \times \{\pi\}$. Note that it is necessary to add to the Cartesian product also the sequence of indices to preserve the structure of io-forests and hence to be able to construct graph language of F.The *graph language* of F is the set of io-graphs $L(F) = \{\otimes f \mid f \in L_f(F)\}$. We say that F respects *canonicity* if $\forall f \in L_f(F)$: f *is canonical.*

One of the most important operations over forest automata used in the program analysis is checking graph language inclusion of two forest automata. This operation is performed to check whether symbolic execution reaches a fixpoint. Checking inclusion of languages of forest automata that respect canonicity can be done *component-wise*, i.e. checking language inclusion of their tree automata one by one.

Lemma 2.1.1. *Let $F^1 = (A^1_1 \cdots A^1_{n_1}, \pi^1)$ and $F^2 = (A^2_1 \cdots A^2_{n_2}, \pi^2)$ be two canonicity respecting FA. Then $L(F^1) \subseteq L(F^2)$ iff*

- $n_1 = n_2$
- $\pi^1 = \pi^2$
- $\forall i \in \{1, \ldots, n_1\} : L(A^1_i) \subseteq L(A^2_i)$

Proof can be found in [11].

Example 2.1.5. *Consider the io-forest $f_{io} = ((t_1, t_2, t_3), (1, 3))$ from Example 2.1.3. The tree t_1 (which is the same as the tree t) belongs to the language of TA A defined in Example 2.1.4. We further have a TA $B = (Q_B, \Sigma, \Delta_B, R_B)$ where $Q_B = \{p_1, p_2, p_3\}$, Σ is same as in Example 2.1.4, $\Delta = \{p_1 \xrightarrow{a} (p_2, p_3), p_2 \xrightarrow{b} (), p_3 \xrightarrow{b} ()\}$ and $R = \{p_1\}$. The TA B contains t_2 in its language. Finally, consider a TA $C = (Q_C, \Sigma, \Delta_C, R_C)$ where $Q_C = \{r_1\}$, Σ is again the same as before, $\Delta = \{r_1 \xrightarrow{b} ()\}$ and $R = \{r_1\}$. A set $L(C)$ contains t_3. Putting all automata together we can construct the forest automaton $F = ((A, B, C), (1, 3))$. The io-forest f_{io} is in the forest language $L_f(F)$ of F because it belongs to $L(A) \times L(B) \times L(C) \times \{(1, 3)\}$. Hence the graph $\otimes f_{io} = (\otimes(t_1, t_2, t_3), (u_1, w_1))$ (shown in Example 2.1.3) is in the graph language $L(F)$.*

2.2 Forest Automata of Higher Level

The FA defined above can represent data structures, like singly-linked lists or trees, by already defined forest automata. The hierarchical FA introduced in this section extend expressive power of basic FA. They are able to represent the new classes of data structures (further described in 3.2.2) like doubly-linked lists or trees with root pointers. An alphabet of such forest automaton contains so-called structured labels (described in Section 2.2.1) and another FA (described in Section 2.2.3).

2.2.1 Structured Labels

To easy definition of FA of higher level we introduce *structured labels.* Let Γ be a ranked alphabet of sub-labels with defined total ordering $\sqsubset$ which is called *sub-labels ordering.* Let g be a graph defined over 2^Γ where A denotes a label of g and $\forall A \subseteq \Gamma : \#A = \sum_{a \in A} \#a$.

The graph g has edges in the form $v \mapsto (A, v_1 \cdots v_n)$ where $A \subseteq \Gamma$ and $\#A = n$. We denote such edge by e. Each edge e consists of *sub-edges* that creates sequence $e\langle 1\rangle = v \mapsto (a_1, v_1 \cdots v_{\#a_1}) \cdots e\langle n\rangle = v \mapsto (a_m, v_{n-\#a_m+1} \cdots v_m)$. We denote the i-th sub-edge of e in g by $e\langle i\rangle = v \mapsto (a_i, v_k \cdots v_l)$ where $i : 1 \leq i \leq m$. We use $SE(g)$ to denote all sub-edges of the graph g. A node v of a graph is *isolated* if it is not part of any sub-edge. Formally, a node v is isolated iff $\nexists e\langle i\rangle = v \mapsto (a_i, v_k \cdots v_l) \wedge \nexists e\langle i\rangle = v' \mapsto (a_i, v_k \cdots v_l) : v \in \{v_k, \ldots, v_l\}$. A graph g is *unambiguously determined* by $SE(g)$ if g has no isolated nodes.

2.2.2 Tree Automata over Structured Labels

Since we have already extended the notion of labels to the structured ones, we also extend tree automata to the ones over such labels. A TA over structured labels is quadruple $A = (Q, 2^\Gamma, \Delta, R)$ where Q, Γ and R has the same meaning as in the case of basic TA and Δ is a set of transition rules with the rules in the form $(q, \{a_1, \ldots, a_m\}, q_1 \cdots q_n)$ where $q, q_1, \ldots, q_n \in Q$, $\{a_1, \ldots, a_m\} \in \Gamma$. Each rule could be interpreted as a sequence of the *rule-terms* $d\langle 1\rangle = q \mapsto (a_1, q_1 \cdots q_{\#a_1}) \cdots d\langle n\rangle = q \mapsto (a_m, q_{n-\#a_m+1} \cdots q_n)$ and we denote the i-th rule term of sequence again by $d\langle i\rangle$ where $i \in \{1, \ldots, m\}$,

2.2.3 Forest Automata of Higher Level

Following the extension of TA, we define also FA over the structured labels. Informally, a forest automaton of a higher level can have another forest automata (of lower lever) as the symbols on its edges. This makes it possible to build a hierarchy of forest automata. We explain the hierarchy using induction. Let start from the forest automata of *level* 1. Forest automata of level 1 can have only the structured labels from 2^Γ in alphabet but not forest automata. They forms the set Γ_1. Then all forest automata of *level* i form the set Γ_i. A forest automaton F of level $i + i$ is defined over the ranked alphabet $2^{\Gamma \cup \Delta}$ where Δ is a subset of forest automata of level i which are called *boxes* of F. Finally, the set of all forest automata of all levels $\sum_{i \geq 0} \Gamma_i$ is denoted by Γ^* and it is ordered by a total ordering $\sqsubset_{\Gamma_*}$ coming the sub-labels ordering $\sqsubset$.

We define an operation *sub-edge replacement* which helps us to define semantics of forest automata of higher level. Informally, the method removes a sub-edge and matches nodes at its left-handed and right-handed side with a new graph serving like a substitution of the sub-edge by a graph. This operation will be further used for a replacement of sub-edge of FA with by a graph represented by FA of lower level.

Formally, let g be a graph with an edge $e \in edges(g)$ and sub-edge $e\langle i\rangle = v_1 \to (a, v_2 \cdots v_n)$. Let g'_ϕ be an io-graph such that $|\phi| = n$. We assume that $dom(g) \cap dom(g') = \emptyset$. The sub-edge $e\langle i\rangle$ could be replaced by g' if $\forall j \in \{1, \ldots, n\} : l_g(v_j) \cap l_{g'}(\phi_j) = \emptyset$ (this conditions checks whether there is no successor of v_j and ϕ_j reachable over the same label from the both nodes). The result of replacement (if it is possible to do it) is denoted as $g[g'_\phi/e\langle i\rangle]$. It is the graph g_n in the sequence $g_0 \cdots g_n$ of graphs which are defined as follows:

- $SE(g_0) = SE(g) \cup SE(g') \setminus \{e\langle i\rangle\}$.
- $\forall j \in \{1, \ldots, n\}$: *the graph* g_j *is obtained from* g_{j-1} *by following procedure*
    1. Deriving a graph h by replacing the origin of the sub-edges of the j-th port ϕ_j of g' by v_j.
    2. Redirecting edges leading to ϕ_j to v_j, i.e., replacing all occurrences of ϕ_j in $rng(h)$ by v_j

3. Removing ϕ_j.

We apply the concept of sub-edge replacement to the forest automata of higher level now and introduce following two procedures over FA.

- *Unfolding* of a graph g is replacement of its sub-edge with a symbol, which is an FA F', by a graph from $L(F')$. Formally, sub-edge $e\langle i\rangle$ of the graph g has a symbol a and this symbol is a FA F' (so this symbol is a box) and $g'_\phi \in L(a)$ then $h = g[g'_\phi/e\langle i\rangle]$ is an unfolding of g. We use $g \prec h$ to denote that h is unfolding of g.

- *Folding* is a replacement of g'_ϕ by $e\langle i\rangle$ in h obtaining g. So we can say that g'_ϕ is folded to $e\langle i\rangle$.

A transitive reflective closure of $\prec$ is denoted by $\prec^*$. A set of all graphs obtained by repeated application of unfolding from a graph g over ranked alphabet Γ is called Γ-*semantics*. Formally defined, it is the set of graphs g' such that $g \prec^* g'$. We denote it as $[\![g]\!]_\Gamma$ or just simply $[\![g]\!]$ when it is clear which alphabet we speak about. Finally, Γ-*semantics* is defined for a FA F of higher level as $[\![F]\!] = \bigcup_{g_\phi \in L(F)}([\![g]\!] \times \{\phi\})$. Note that the meaning of $L(F)$ and $L_f(F)$ has not been changed so the both sets (languages) contain graphs (or forests) over Γ.

When we recall the definition of canonicity respecting FA we will find that it is applicable also for FA of higher level. A FA F is canonicity respecting if $\forall f \in L_f(F) : f$ *is canonical*. Since definition of canonical FA is not affected by extending the labels to the structured ones the meaning of canonicity is the same as for basic FA. The language inclusion checking is again possible in the component-wise way like in the case of basic FA, as it is proved in [11]. However, the testing language inclusion with the same algorithm used for the non-hierarchical FA is sound but incomplete because the structured labels are compared as uninterpreted symbols and their semantics is not considered.

Chapter 3

Forest Automata based Verification

Consider programs manipulating dynamic data structures. These programs can change the shape of the heap (and so reach another heap configuration) by performing operations like allocation of a new memory node on the heap, freeing an existing memory node, updating the references between memory nodes or other pointer operations. Since dynamic data structures used in these programs can be unbounded, the number of potentially reachable heap configurations is infinite. However, we use FA to represent this infinite state space by forest automata in a finite way. Then the shape analysis can be done over the forest automata representing the reachable heap configurations to discover whether no undesirable behavior like a dereference of an uninitialized pointer can happen.

This chapter provides an overview of the shape analysis based on forest automata introduced in [12]. First, a heap representation using FA is described and then we provide an overview of the symbolic execution and also describe the method in context of the framework of abstract interpretation.

3.1 Heap Representation

It is possible to view a *heap* (more precisely a single heap configuration) as a (directed) graph where each allocated heap cell corresponds to a node in the graph [15]. The heap cells consists of *pointer selectors* and *data selectors*. A pointer selectors can point to another graph node, to the *null* value or it can be *undefined*. A data selector is related to data from some finite data domain. Such heap graph can be viewed as an io-graph where the pointer variables pointing to the nodes of the heap are the ports of the io-graph. The io-graph can be accepted by a forest automaton as a member of its graph language. Forest automata are able to represent of the reachable heap graphs by their graph languages. The data structures with heap graphs with unbounded number of cut-points (e.g., doubly-linked lists) are represented by hierarchical forest automata. In general, the hierarchical forest automata are able to represent more data structures down to their higher expressive power.

Let us formalize the idea given above. We denote a pointer selector by $PSel$, a data selector by $DSel$ and data domain by $\mathbb{D}$. A single heap configuration is an io-graph g_{st} over the ranked alphabet of the structured labels from 2^{Γ} with the sub-labels from the ranked alphabet $\Gamma = PSel \cup (DSel \times \mathbb{D})$ having the ranking function that assigns 1 to the pointer selectors and 0 to the data selectors. The values of data selectors are stored in the structured labels as the sub-labels from $DSel \times \mathbb{D}$. The internal structure of a memory cell in the heap

is reflected by the structured label $l_g(v)$ of a node v representing the memory cell. A null value is represented by a node `null` with $l_g(\texttt{null}) = \emptyset$. The undefined selectors of a memory node v have no special syntax so they are not presented in the structured label $l_g(v)$.

Not only the allocated memory on the heap is modeled by forest automata in the verification procedure but also a *stack frame* of the actually analyzed function of the original program is represented by forest automata. Particularly, the io-graphs and forest automata have from their definitions exactly one (index of) input port and just the one input port *sf* keeps the information about the stack frame.

Example 3.1.1. *We illustrate described principle of heap representation with an example taken from [13]. We consider a singly-linked list whose items are data structures containing pointers to a next item and also integer data variable. Written in C:*

```
struct SLL {
 struct SLL* next;
 int data;
};
```

Consider an io-graph g representing a heap and an allocated cell of this singly-linked list on the heap with value 13 in `data` *variable and pointer to a cell named s_{next} in the* `next` *variable. The memory cell can be represented by a node s with following label $l_g(s) = \{next_g(s_{next}), (data_g, 13)()\}$. The pointer selector $next_g \in PSel$ representing the pointer variable* `next` *from the* `SLL` *structure has really the rank 1 and the one successor state is s_{next}. The sub-label $(data_g, 13) \in DSel \times \mathbb{D}$ representing* `data` *variable from* `SLL` *structure has the rank 0 and has no successor what is denoted by ().*

3.2 Symbolic Execution

So far we described the representation of heap configurations using forest automata. Now we will describe how the forest automata representation is computed for each point of the analysed program. FA-based verification procedure is a standard *abstract interpretation* [6]. The concrete domain is a set of heap graphs. To each program location can be assigned a set of pairs (σ, H) where H is a single heap configuration and σ is a mapping that maps variables to the nodes in H, to `null` or to an *undefined* value. The abstract domain is a set of forest automata. The verification procedure computes for each program location a finite set of pairs (σ, F) (called *abstract configuration*) where F is a hierarchical forest automata of higher level respecting canonicity and σ maps again each variable to `null` or to undefined value or to an index of TA in F. Since FA are not closed under union, it is not possible to represent the sets of heaps by single FA. Therefore a set of FA is needed to represent the set of all reachable heap configurations at one program location.

Let us now describe the notion of abstract transformers. The function $f_{\texttt{op}}(g_{st})$ is related to an operation `op` of the analysed program. This function models semantics of `op` in the concrete domain in such way that $f_{\texttt{op}}(g_{st})$ transforms an io-graph representing the heap configuration before and after the execution of the concrete operation `op`. The abstract transformers $\tau_{\texttt{op}}$ are defined for each concrete operation `op` reflecting the semantics

of $f_{\texttt{op}}(g_{st})$. They transform a FA S representing the heap in abstract domain to a resulting FA $S' = \tau_{\texttt{op}}(S)$ such that $\bigcup_{F' \in S'} [\![F']\!] = \{ f_{\texttt{op}}(g_{sf}) \mid g_{sf} \in [\![F]\!] \wedge F \in S \}$.

The verification starts from an initial abstract configuration that consists of an FA for the initial heap configuration representing an io-graph g_{sf} where g consists of two nodes. The first one is `null` and the second one is the empty stack frame sf with $l_g(sf) = \emptyset$. The sequence of *abstract transformers* related to the program statements is then iteratively applied to the abstract configurations. The process of applying abstract transformers over the abstract domain is called *symbolic execution.*

The abstract transformers related to a program location are applied iteratively until abstract configurations reach fixpoint. Each iteration consists the following steps:

1. The sets of abstract configurations at each program point are updated by applying the abstract transformers following these steps:
   (a) Some boxes of FA in abstract configuration are unfolded to materialize the accessed parts of heaps by abstract transformers. This step is called *unfolding.*
   (b) The abstract configuration is updated by the abstract transformer.
   (c) New boxes are *learnt* by the method described in [15] and boxes are applied again. This is repeated until no new boxes are found.
   (d) FA are transformed to the canonicity respecting form.
2. At junctions corresponding to the beginning of the loops the union of the possible heap configurations of the particular branches is followed by *widening.* This requires checking language inclusion between sets of FA to test whether a fixpoint has been reached. It is necessary to transform the FA to canonicity respecting form before inclusion checking.

Widening currently consists of repeating the following two steps to each F in the abstract configurations of a junction point until the fixpoint is reached:

1. The boxes of F are folded.
2. Abstraction — currently based on the framework of *abstract regular (tree) model checking* [3].

We will further describe some parts of the verification procedure in detail. *Normalization* transforming FA to canonicity respecting FA is described in Subsection 3.2.1. Subsection 3.2.2 describes how the boxes are used in the verification procedure. Finally, the methods of abstraction over FA are described in 3.2.3.

3.2.1 Normalization

Normalization performs transformation over a FA $F = (A_1 \cdots A_N, \pi)$ and its result is canonicity respecting FA. Normalization consists of the following steps:

- We merge TA of F into a form where the roots of forests accepted by F are cut-points only. Consider there is a TA A which accepts trees with roots that are not cut-points of the related graph from $L(F)$ and a TA B that contains the root references to A. A is then connected to B at the referenced places and so a new TA $B_A = (Q_A \cup Q_B, \Gamma, \Delta_{A+B}, R_B)$ is created. The set of transition rules Δ_{A+B} is $\Delta_A \cup \Delta_B$ where the transitions $q \to \overline{a}(q_1 \cdots q_i \cdots q_n) \in \Delta_B$ are substituted by $q \to \overline{a}(q_1 \cdots q_a \cdots q_n) \in \Delta_B$, where q_i is a root reference to A and q_a is the root of A.

- TA of F are reordered by canonical ordering of cut-points.
- We obtain F in form in which the roots of trees from the forest language correspond to the cut-points respecting the canonical unique ordering. It meas that $\forall i \in \{1, \ldots, n\}$ and for all accepted forests $f = (t_1, \ldots, t_n)$ holds one of the following conditions:
  - The root of t_i is the j-th cut-point in the canonical ordering of the cut-points of the graph of the accepted forest $\Rightarrow$ It is the j-th cut-point in the canonical ordering of all accepted forests.
  - Otherwise, the root of f_i is not a cut-point of any accepted forest.

3.2.2 Boxes

There can be infinitely many cut-points in the graphs representing the heaps. For instance each node of doubly linked list is a cut-point. These heaps cannot be represented by the plain forest automata since the tree decomposition of such graph is not possible. This can be resolved by employing the boxes (box is a symbol of alphabet of forest automata of higher level which is also forest automata as it was defined in Chapter 2). Since it is possible for FA to use other FA as symbols the repeating sub-graphs of the heap are folded to the boxes. They are further used as labels and thus bound cut-points. When an abstract transformer needs to access a graph described by a FA hidden in the box, then the unfolding is done.

The boxes can be learnt during the analysing of a program automatically by the method proposed in [15] or it can be given to the analyzer manually by a user [10].

3.2.3 Abstraction over Forest Automata

Forest automata are able to handle the infinite state space rising from the possible unboudness of the mentioned dynamic data structures by representing them in a finite way. The finite representation is possible due to abstraction over forest automata. The abstraction merges some states and so creates the cycles in transitions what make forest automaton able to represent an infinite state space. Moreover, the abstraction increases the probability of the termination and accelerates the method. The abstraction over forest automata is based on the framework of *abstract regular tree model checking* [3].

The abstraction merges states of forest automaton that are equivalent according to a given equivalence relation over the states of forest automaton. Since the abstraction is defined for a single tree automaton it is performed component-wise over a forest automaton. Formally, the abstraction α over a tree automaton $M = (Q, \Sigma, \Delta, R)$ is a function $\alpha : Q \rightarrow Q/_{\sim_M}$ such that $\alpha(q) = [q]_{\sim_M}$ where $\sim_M \subseteq Q \times Q$ is an equivalence relation. We denote $\alpha(M)$ the tree automaton obtained by applying α to Q. It holds that $|Q/_{\sim_M}| \leq |Q|$ and also that $L(M) \subseteq L(\alpha(M))$.

The forest automata based verification is currently implemented (in the Forester tool) employing *height abstraction*. This abstraction merges the states with the equivalent languages upon to a given tree height. The range of abstraction function is the set of the equivalence classes of the relation $\sim_M^n$. The relation $\sim_M^n$ is therefore defined as follows: $\forall q_1, q_2 \in Q : q_1 \sim_M^n q_2 \Leftrightarrow L^{\leq n}(M, q_1) = L^{\leq n}(M, q_2)$ where $L^{\leq n}(M, q)$ is a language containing sub-trees obtained from trees of $L(M, q)$ by their restriction up-to height $\leq n$.

The abstraction over forest automata overapproximates the set of the reachable configurations of a program (formally, $L(M) \subseteq L(\alpha(M))$). This it can led to a detection of a spurious counterexample which is not present in an original program but it is caused by

a heap configuration created by abstraction. In [4], the *counterexample-guided abstraction refinement* (CEGAR) for avoiding a spurious counterexample was proposed. It consist two steps. The first is detection of spuriousness of an error found by the analysis. When it is proved that the error is spurious then the second step is refinement of the abstraction α to abstraction α' such that $L(\alpha'(M)) \subseteq L(\alpha(M))$. The program analysis is restart using the refined abstraction α' what may prevent reaching the detected spurious counterexample again.

In the case of height abstraction this refinement is done by increasing the height n. However the refinement does not guarantee that the detected spurious counterexample will not be detected again the next run. Moreover, no automatic refinement and also not even spurious counterexample detection have been implemented in the Forester tool yet. This will be resolved by implementing backward run and predicate abstraction what is described later in Chapter 6. Predicate abstraction is refined by the predicates learnt from backward run what prevents reaching the spurious error again and so provides finer refinement then height abstraction.

Chapter 4

The VATA Library and Forester

As we mentioned in the introduction, FA based verification is implemented by the tool Forester. Since FA are closely related to TA, as it was shown in Chapter 2, Forester also depends on an implementation of TA. It currently has its own implementation of TA providing the operations needed during the verification procedure. This library has the strong dependencies with the rest of code. The other classes depend to the library data types, manipulates its data members and so on. The changes in this library are very difficult due to these dependencies. Encapsulating of the library to a module with an clearly defined interface can improve the overall quality of Forester code. It is also not practical to maintain and further develop such one purpose library. It is not general to be used anywhere else. But the development in algorithms for TA, especially the one for checking inclusion of TA languages, goes on and an implementation of the new algorithms brings the greater efficiency. A new efficient algorithm also often achieves greater efficiency than an implementation optimization of an existing algorithm. Therefore the replacement of library by a general purpose one can bring the advantages such as maintainability and modularity.

With respect to the mentioned facts we decided to replace the mentioned Forester TA implementation by the VATA library. The VATA library is a very efficient, general purpose, library which provides implementation of the standard operations over TA like union or intersection. It mainly aims at implementing the state-of-the-art algorithms [1] for language inclusion checking efficiency of which is crucial for performance of Forester.

This chapter provides a brief description of the VATA library and the Forester tool.

4.1 The VATA Library

The VATA library is an open source library for nondeterministic tree automata implemented in C++. Its main application is in the field of formal verification. VATA is licensed under GPL, version 3, and it can be obtained from its official website [25]. To the our best knowledge, it is the only library implementing the state-of-the-art algorithms for checking inclusion of NTA languages, which makes it the only suitable choice for use as the backend library of Forester.

4.1.1 Design

The VATA library currently provides explicit encoding and also semi-symbolic (top-down and bottom-up) encoding using *multi-terminal binary decision diagrams (MTBDD)* of NTA. It has been designed to be easily extensible by other encodings. The library provides API for

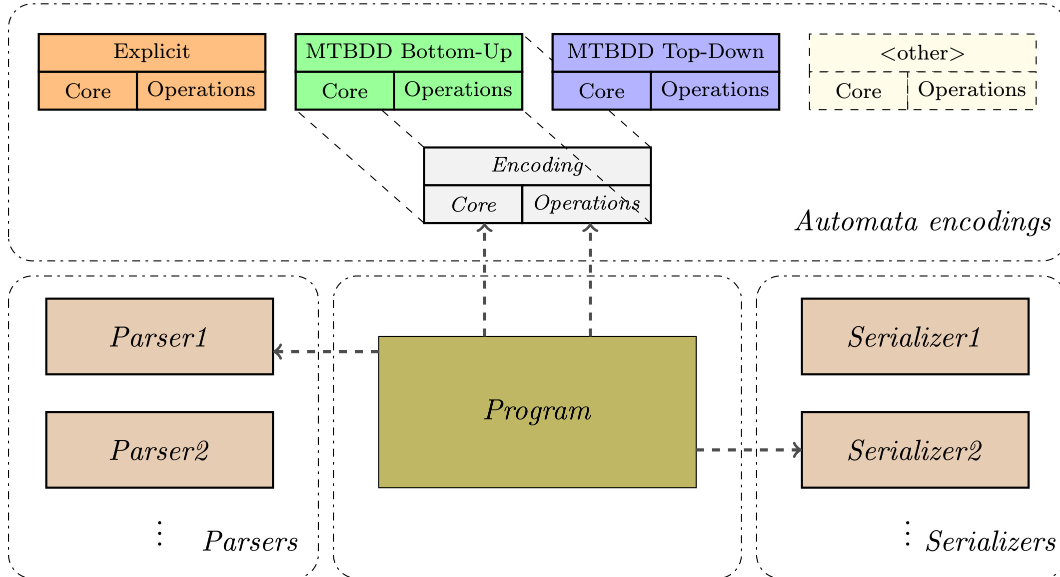


Figure 4.1: The main concept of the VATA library. Figure is taken from [20].

creating and manipulating NTA and also the command line interface (cli) build around the API for experimenting with tree automata defined in a text format directly from command line. The main concept of the design of the library is given in Figure 4.1. There are the three main parts in the library design:

1. *Parsers* — Parsing an input automaton from a text file. Timbuk [26] is currently the only one supported format for parsing the input automata.

2. *Serializers* — Serializing an automaton to a text file. Timbuk format is again the only supported format.

3. *Automata encodings* — The particular encodings of NTA. An encoding should consists of a core module implementing NTA representation and also the operations over NTA in this encoding.

A *program* (e.g., cli of VATA) employing the three parts of the VATA library works as follows. An input automaton is loaded by one of the parsers to an intermediate representation. The wrapping program chooses an internal encoding of NTA into which the automaton is translated from the intermediate encoding (note that it is also possible to create automaton in the chosen encoding directly using API provided by VATA). The automaton is then processed by applying the operations implemented by a module of the chosen encoding. Finally, the automaton can be serialized to an output format. When one wants to add her own encoding, then she needs to implement only the core of the encoding (with an API for creating the automaton itself) and needed operations over TA, and can employ already implemented parses and serializers of VATA.

4.1.2 Implemented NTA Encodings

We will now describe the two encoding of TA in the following subsection.

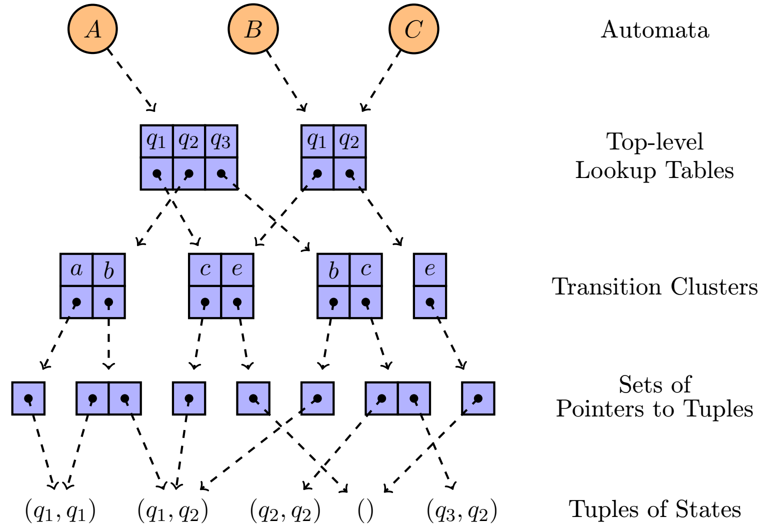
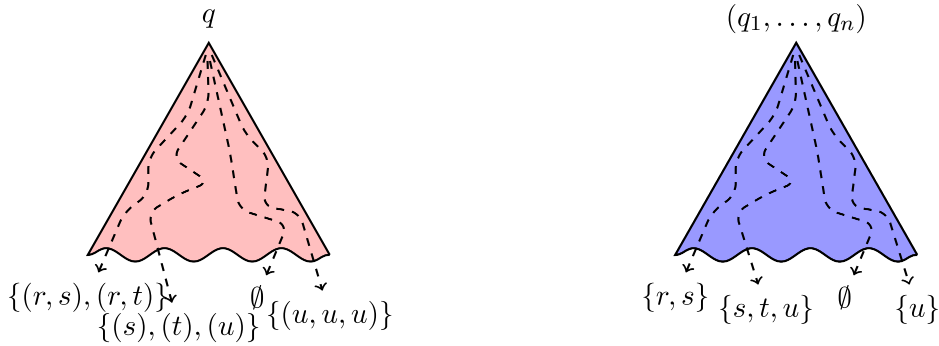


Figure 4.2: Explicit representation of NTA in the VATA library. Figure is taken from [20].



(a) MTBDD Top-down representation of a NTA. Image is taken from [20].

(b) MTBDD Bottom-up representation of a NTA. Image is taken from [20].

Figure 4.3: Semi-symbolic encoding of tree automata.

The *explicit encoding* of NTA transition relation is a hierarchy of hash tables, as it is shown in Figure 4.2. The first level of the hash tables hierarchy (*top-level look-up tables*) maps each state q to the second level of the hash tables hierarchy (*transition cluster*). It stores symbols that appears at a transition with q as the left-handed side. Each symbol a in a transition cluster is mapped to a pointer to a set in the third level of hierarchy (*sets of pointers to tuple*). The set contains pointers to the tuples which are at the right-handed sides of the transitions with q at the left-handed side and with a as the symbol. The tuples are stored at a set where every tuples is stored only once. This hierarchy allows one to share the part of the transition relation between different automata what brings the higher space efficiency. Module for explicit encoding also stores explicitly the set of the final states of the NTA. On the other hand, it does not explicitly store a set of all states because it can be obtained from the transition relation.

The other encoding is the *semi-symbolic* one, based on VATAs own of the MTBDD package. This encoding is efficient mainly for TA with large alphabets. The main principle of the semi-symbolic encoding is shown in Figure 4.3. First of all it is necessary to dis-

Operation	Explicit	Semi-symbolic	
	top-down	bottom-up	top-down
Union	+	+	+
Intersection	+	+	+
Complement (experimental)	+	+	+
Removing useless states	+	+	+
Removing unreachable states	+	+	+
Simulation	+	−	−
Downward Inclusion	+	+	−
Upward Inclusion	+	−	−
Simulation over Labeled Transitions System	+	−	−

Table 4.1: Table of supported operations over NTA by particular encodings implemented in the VATA library. Table is taken from [16].

tinguish between (a) top-down and (b) bottom-up variants of this encoding. The first one maps each state q of a NTA using MTBDD to the sets of the tuples of states such that that it is possible to make transition from q under a symbol a to a tuple in appropriate set (each set of tuples is dedicated to one symbol under which it is possible to make transition from q). The former one symmetrically maps each n-tuple $(q_1 \cdots q_n)$ of a NTA using MTBDD to the sets of states. Each such a set S is dedicated to a symbol a of the NTA and it contains states such that there exists a transition with $(q_1 \cdots q_n)$ at the right-handed side and symbol a and state from the set S at the left-handed side. The final state set of a NTA is again represented by explicit set in the both variants. A state set is not stored explicitly because one can obtain it from the transition relation. The symbols are encoded (as binary strings) in MTBDD. The more detailed description of theory and implementation of semi-symbolic encoding and MTBDD can be found in [19].

All of the mentioned encodings currently support efficient language inclusion checking using the algorithms from [1]. On the other hand, the other operations are not currently implemented by all encodings. The full enumeration of the supported operations for the particular encodings is given in Table 4.1.

4.2 Forester

Forester is an open source for verification of programs manipulating complex dynamic data structures tool written in C++. It currently supports the analysis of programs in the C language. Forester is distributed as a *GCC* plugin under GPL license, version 3, and it can be obtained from its official website [23].

4.2.1 Design

Forester is implemented as a GCC plugin however it does not directly analyze intermediate code of GCC called GIMPLE. It uses the Code Listener infrastructure [8] that provides a fronted over the GIMPLE format.

Let us describe the verification procedure done by Forester (shown in Figure 4.4) and the high level conceptual design of Forester (which is shown in Figure 4.5) and the relations

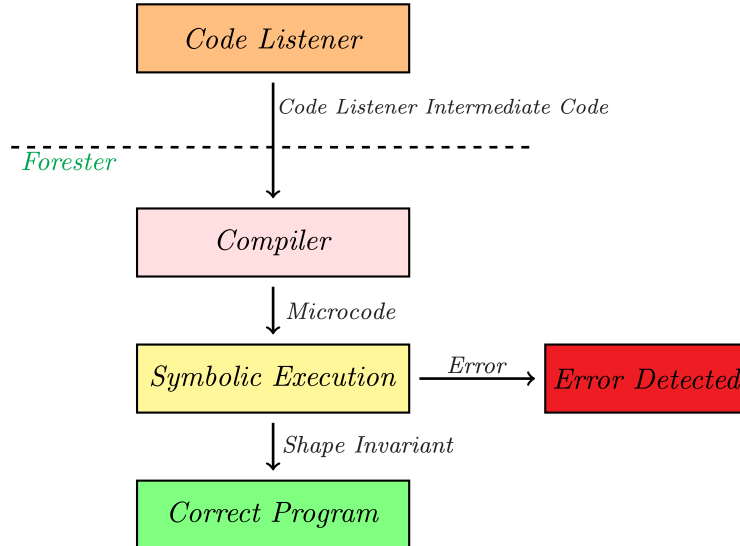


Figure 4.4: High level overview of Forester program analysis.

between its modules. The implementation of Forester is not explicitly separated into stand-alone compilation modules. The notion of the modules used in the following text is rather an abstract one, serving to a reader basic summary of the Forester design. A module typically is e.g., a set of closely related classes with a similar purpose. Forester starts the analysis by translation of the intermediate code of GCC in representation provided by *Code Listener* to its own microcode. This is done mainly by Forester *Compiler* module. The microcode instructions of Forester are implemented by module *Microcode Instructions*. Each program statement is represent by one or several *Microcode Instructions* associated with the abstract transformers. *Compiler* creates a list of *Microcode Instructions* over which the *Symbolic Execution* is performed. *Symbolic execution* run the microcode instructions (abstract transformers) which manipulates *Symbolic State* and *Forest Automata* included in the symbolic states. When Forester detects an error the symbolic execution is aborted and the analyzed program is claimed incorrect. During the symbolic execution, Forester checks whether there is no left garbage. If the program is garbage free and without errors then a shape invariant has been found and the analyzed program is determined as correct.

Symbolic execution requires *Symbol Context* which is created for each function and also for global space. It keeps information about the variables used in the current scope, the function arguments and the stack frame layout. A symbolic state provides information about the state of the heap, represented by FA, and it also keeps the information about the corresponding microcode instructions.

Forest Automata module provides the methods for manipulation of FA during the verification procedure. The operations such as normalization or abstraction over FA are not the part of the module containing FA implementation but are provided as classes taking FA as parameters. So these operations can be understood as another module *Operations over Forest Automata*. Forester currently has its own implementation of *Tree Automata*. It is very lightweighted and contains the operations optimized for the purposes of Forester. One of them is an operation for language inclusion checking. It uses the simulation relation for acceleration because the efficiency of inclusion checking is crucial for the performance. The advantage of this implementation is its simplicity and the efficiency raising from the

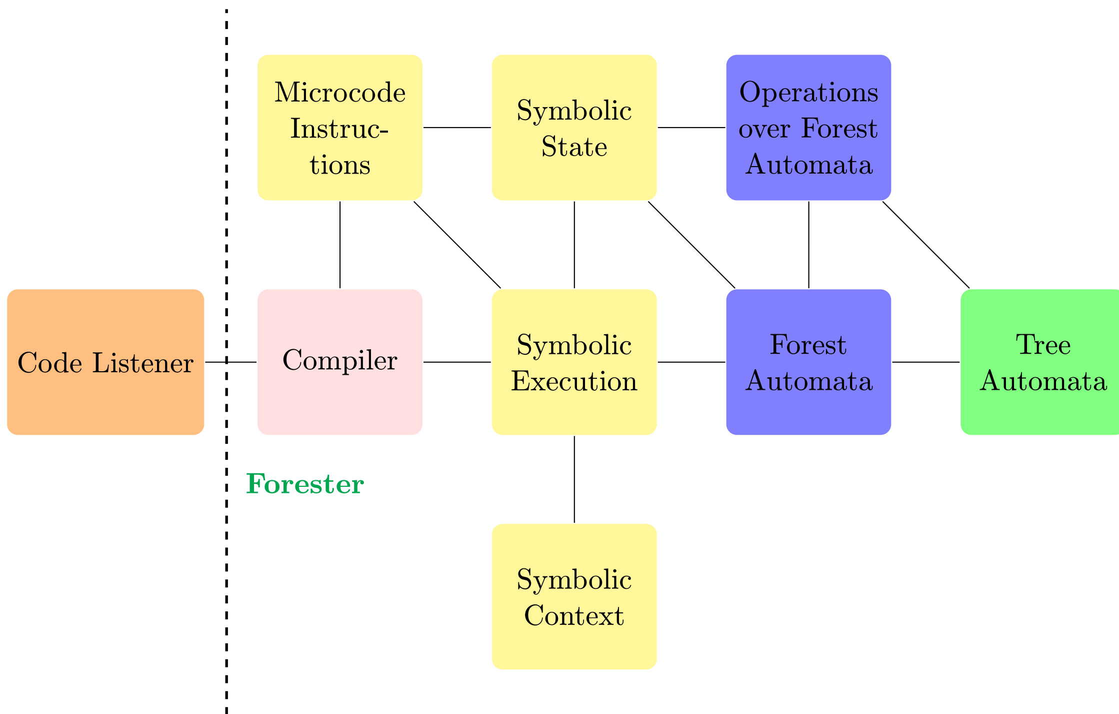


Figure 4.5: Conceptual design of Forester.

optimization the implementation to suit the needs of Forester.

A real implementation is much more complicated (e.g. Forest automata are implemented by two classes: *FA* and *FAE*) and full of the technical details. A full description of the implementation is also not the aim of this text therefore we provide just this conceptual view of the design.

It was already mentioned that the substitution of the underlying TA implementation with the VATA library could bring some improvements. The first one is the easier maintenance of the library where the state-of-the-art algorithms are implemented and optimized in comparison with the one purpose implementation. Since VATA and Forester are developed by the same developers it is also easy to add to VATA the operations needed by Forester. The clearly defined interface between TA library and the rest of Forester also improves the code quality of Forester in the sense of modularity, maintainability and code organization. These benefits led us to implement the version of Forester using VATA, further described in Section 5.

4.2.2 Implementation of Forest Automata Concepts

This subsection describes the implementation of the forest automata based verification procedure in Forester. The core of the forest automata module is class `FA`. The implementation corresponds with the definitions given in Chapter 2. Class FA has a data member representing a vector of tree automata $(t_1 \cdots t_n)$. It also has a data member keeping a reference to an automaton representing the global variables and a reference to the automata representing the local variables of the actually executed function. The basic operations over an

FA (like adding or removing a TA from an FA) are encapsulated in class FA. The other class providing operations over FA is class `FAE` (Forest Automata Extension).

The labels of the TA transitions are represented by the structure `NodeLabel`. It is defined as a tuple $NL = (T, V)$. This structure contains a data member T determining the type of the root node of a transition (whether it is a general node, a data node or a node containing a vector of data). A value V related to the type of the node is also stored as a data member of the structure.

As it was already mentioned, the forest automata are further used in the representation of a symbolic state. An implementation of the representation of a symbolic state is in class `SymState`. This class keeps a reference to a Forester microcode instruction I which is executed at the symbolic state. It has a data member that is a pointer to a forest automaton F (an instance of class `FAE`) representing the actual memory shape in the symbolic state. Another data member of class SymState is a set of registers $\{r_1, \ldots, r_m\}$. We denote set $\{r_1, \ldots, r_m\}$ by $\mathbb{G}$. A register can be local or global one. There are two global registers. The first one contains a reference to the forest automata modelling the global state, e.g., the block of global variables in the symbolic state. The other then represents the allocated memory of the actually executed function. The local registers serve as temporary memory used in the microcode instructions for data manipulation. Putting together the data members of class `SymState`, we define symbolic state formally as a triple $S = (F, \mathbb{G}, I)$.

The values of the registers are instances of a structure `Data`. We formally define structure Data as $D = (T, S, V, MB)$ where T is a data type, $S \in \mathbb{N}$ is a size of data, V is a value of type T, $MB = (D_1 \cdots D_n)$ are nested data when D is a data structure. T can be one of the following data types: pointer, reference to a TA, integer, boolean, structure or generally some other data type. T can be also undefined or it can be a state with the type that is unknown. We describe in detail one of the types — the root reference. Its values are defined as $RR = (ROOT, DISPL)$ where $ROOT \in \mathbb{N}$ is an index of a tree automaton in F and the displacement $DISPL \in \mathbb{N}$ is an index of a selector in a structured label of the transition with RR at the left-handed side. The displacement $DISPL$ is given by the sum of sizes of the preceding selectors so it determines the precise position of the selector in label. The operations for manipulation with a symbolic state are partially provided by class `SymState` and some additional operations are provided by class `VirtualMachine`.

We already described described the forest automata implementation in Forester. The symbolic execution using this data structures is realized by gradually appending and removing the symbolic states from a queue. The implementation of symbolic execution is in class *SymExec*. When a symbolic state is taken from the queue a microcode instruction contained in the symbolic state is executed. Some of the instructions during its execution append a new symbolic state to the queue. Symbolic execution finishes when the queue is empty. The microcode instructions executed during the symbolic execution are described in the following section.

4.2.3 The Microcode Instructions

This sections provides a description of the Forester microcode instructions which correspond to the abstract transformers in abstraction interpretation. First, we introduce some examples how the C statements are translated to the corresponding Forester microcode instructions, then we define the instructions more formally and we give a complete list of them. Consider the following data structure that is an implementation of singly-linked list

```
struct T {
      struct T* next;
      int data;
};
```

and the declared variables `struct T *x, *y` used in the following examples. All constants used in the examples are dependent on a concrete Forester run and they are chosen randomly in this text. The syntax of the instructions in the examples is following:

$$op(r_0,\ r_1,\ r_2)$$

where op is a name of a instruction, r_0, r_1, r_2 are operands such that r_0 is a destination register and r_1 and r_2 are the source registers. But only some of the instructions require all three parameters, the other instructions take just two or less parameters. A constant may be also used in place of a register. When $[r + c]$ is used as an operand then the instruction dereferences a root reference in register r. More precisely, the instruction accesses directly a selector with the displacement c in a label of the transition with the root referenced by r at the left-handed side. We will further consider the selector in the label of a transition with the referenced root at left-handed side if it is not stated otherwise. In the following text, the registers $r_1, \ldots, r_n$ are local ones. *GLOB* and *ABP* are global registers such that *GLOB* register contains reference to the FA representing global variables and *ABP* register contains reference to the FA representing local memory.

Example 4.2.1. *The statement* `x = (struct T*) malloc(sizeof(struct T));` *is translated to these microcode instructions:*

```
1: mov_reg r0, (int)4
2: alloc r0, r0
3: node_create r0, r0, next[0:4:+0]
4: mov_reg r1, ABP + 0
5: mov_reg [r1 + 12], r0
6: check
```

First, the size of the newly allocated data (in this case the size is 4*) is stored in the register* 0 *on line* 1*. Then the new instance of the structure* `Data` *is created and stored to register* 0 *on line* 2*. A new TA t is created on line* 3*. The TA t has one transition from the root with a label that consists of one pointer selector* `next` *whose displacement in the label is* 0 *and the size of the allocated node is* 4*. The instruction* `node` *stores a root reference to t to register* 0*. Finally lines* 4 *a* 5 *add a root reference pointing to t to a FA representing a memory state of the actually executed function (which is in this case the function* `main`*). In this case a selector related to the pointer x has displacement* 12 *in the label of TA representing the function memory state. The instruction on line* 6 *then checks whether no garbage was created by the previous instructions.*

Example 4.2.2. *The statement* `y = x->next;` *is symbolically executed by the following instructions:*

```
1: mov_reg r0, [ABP + 12]
2: acc_sel [r0+0]
3: mov_reg r0, [r0 + 0]
4: mov_reg r1, ABP + 0
5: mov_reg [r1 + 16], r0
6: check
```

The first instruction stores a root reference to a TA representing heap pointed by the pointer `x` *to register* 0*. In this case, a node with a root reference is pointed by the selector with the displacement* 12 *in a label in TA referenced by register ABP. Because the pointer* `y` *will point also to the this memory after the execution of statement, the new cut-point will be created at this node. The special TA representing the heap reachable from* `y` *will be needed. This is done by the second instruction that makes so-called* isolation *of the selector with displacement* 0 *in the TA referenced by register* 0*. This creates a new TA if* `x->next` *points to an allocated memory, otherwise the instruction does not do anything. The instruction on line* 3 *loads to register* 0 *a root reference from a selector of TA referenced by register* 0*. The instruction on line* 4 *moves a reference to TA that represents a local memory state to register* 1*. Finally, the instruction on line* 5 *copies the root reference from register* 0 *to a TA node pointed by pointer* `y`*. The last instruction checks whether no garbage is present.*

Example 4.2.3. *The statement* `free(x);` *is implemented by these instructions:*

```
1: mov_reg r0, [ABP + 12]
2: acca_sel [r0]
3: free r0
4: check
```

The first instruction loads to register 0 *the root reference to TA representing the heap reachable from* `x`*. The reference to this TA is stored in a selector with displacement* 12 *in TA referenced by register ABP. The second command isolates all selectors of the TA referenced by the register* 0*. The isolation has the same meaning as in the previous instruction. The isolation of the selectors to the particular TA is done in order to enable freeing right the memory represented by the TA. Then the third instruction removes TA pointed by register* 0 *from FA and then it invalidates all references to it. The last instruction again checks whether there is no garbage created by executing this instruction.*

Example 4.2.4. *The statement* `x->data = y->data;` *is done by these instructions:*

```
1: mov_reg r3, [ABP + 16]
2: acc_sel [r3 + 4]
3: mov_reg r3, [r3 + 4]
4: mov_reg r1, [ABP + 12]
5: acc_sel [r1 + 4]
6: mov_reg [r1 + 4], r3
7: check
```

The first instruction loads to register 3 *a root reference pointed by a selector with displacement* 16 *in TA referenced by register ABP (which is the TA modelling the heap of the actually executed function). Then the data selector* `x->data` *(which has displacement* 4 *in label in TA reference by register* 3*) is isolated. A data value of selector is loaded to register r by the instructions on line* 2 *and* 3*. The instructions on line* 4 *and* 5 *prepare the data selector* `y->data` *in the same way as has been prepared the selector* `x->data`*. Finally, on line* 6*, the data value in register* 3 *is copied to the TA node pointed by the selector* `y->data`*. The last operation is again checking whether no garbage is left.*

Example 4.2.5. *The statement* `x == y` *translates Forester into the following list of the instructions:*

```
1: mov_reg r0, [ABP + 12]
2: mov_reg r1, [ABP + 16]
3: eq r5, r0, r1
```

The instruction on line 1 loads the root reference pointed by a selector with displacement 12 in TA referenced by register ABP related to a pointer x to register 0. The second instruction does the same thing as the previous one but for a pointer y *and register 1. The third operation then compares the content of the registers 1 and 2 and stores the result to register 5.*

These examples provide an overview of the concept of the microcode instructions. We will further describe a more detailed and formal description of them. We use $r_s \in \mathbb{G}$ to denote a source register, $r_d \in \mathbb{G}$ to denote the destination register, $r_l \in \mathbb{G}$ to denote a local register and $r_g \in \mathbb{G}$ to denote a global register. Note that the source and destination registers can be the local as well as the global ones. The source and destination registers are the parameters of some instructions. We use $\mathbb{G}[r := r']$ to denote the set G of the registers with register r substituted by the register r'. A symbol I_n denotes then next instruction following an actually executed instruction. We define the semantics of each instruction formally as a function $f : \mathbb{I} \times \mathbb{S} \to \mathbb{S}$ where $\mathbb{S}$ is a set of all symbolic states and $\mathbb{I}$ is a set of all instructions. When the instruction is clear from context, we omit the first parameter of f. The function f defines the effect of the instruction as a transformation of the current symbolic state s to a new symbolic state s'.

We now present a list of the microcode instructions used in Forester with the description of their semantics. Each instruction is described informally and then its effect is defined formally by the function f. The instructions are divided into two parts. The first part contains the instructions modifying forest automata and the second part consists of the remaining instructions that do not change forest automata.

The Instructions Modifying Forest Automata

- `acc_sel(`r_d`)` isolates the i-th selector of a label in the transition from the root of a given TA. The root of TA is defined by the destination register r_d. This instruction checks whether a selector pointed by the destination register selects a node that represents allocated memory node or undefined pointer. In the first case, the node is isolated to a new TA and the original node in the original TA is replaced by a root reference to the new TA. In the other case nothing changes. Note that this basically corresponds to the identification of the cut-points and their separation to a new TA.

  $f((F, \mathbb{G}, I)) = (F', \mathbb{G}, I_n)$ where F' is a new FA obtained from F by isolating the i-th selector of TA t. TA t and the related selector are determined by a root reference $RR = (Root, Displ)$ in the register r_d such that $Root$ references to t and $i = Displ + o$ where o is a offset which is a parameter of this instruction.

- `acc_set(`r_d`),acc_all(`r_d`)` instruction does the same thing as the previous one but for a set of selectors or for all selectors of a given root of TA.

- `node_create(`r_d`,`r_s`)` instructions creates a new node (and hereby also a new TA) which is the root node of the new TA. A reference to the new TA is stored to the destination register r_d. A type info of the new node and its size is obtained from source register r_s. Note that the reference to t is not added to a TA in F in this step but it is done by instruction assigning a register value to a node of TA.

  $f((F, \mathbb{G}, I)) = (F', \mathbb{G}[r_d := x], I_n)$ where $F = (T_1 \cdots T_N, \phi)$, $F' = (T_1 \cdots T_N, T_{new}, \phi)$, T_{new} is the newly created TA, and x is a reference to T_{new}. The selectors in the label of T_{new} are specified in r_s.

- `node_free(`r_s`)` instruction deletes a node in FA (referenced by register r_s) and invalidates all references to it.

$f((F, \mathbb{G}, I)) = (F', \mathbb{G}, I_n)$ where F' is obtained from F by removing the TA t referenced by r_s and removing all reference to t from TA of F.

- `store(`r_d, r_s, o`)` instruction stores a value from the source register r_s to a TA t pointed by the destination register r_d. The location is selected by a selector with displacement o in a label of t.

  $f((F, \mathbb{G}, I)) = (F', \mathbb{G}, I_n)$ where F' is a FA obtained from F by an assignment $S_T^{displ} = r_s$, where S_T^{displ} is a node of the TA T referenced by the root reference $RR = (Root, Displ)$ stored in r_d register. The node is pointed by the selector with $displ$, such that $displ = Displ + o$ and o is an offset which is a parameter of this instruction.

- `stores(`r_d, r_s, o`)` instruction does the same as the previous one but manipulates the structures. Formal definition is also same but the result is a structure.

- `abs(), fix()` instructions computes fixpoint with (abs) or without (`fix`) abstraction.

  $f((F, \mathbb{G}, I)) = (F', \mathbb{G}, I_n)$ where F' is a new FA obtained from F by computing fixpoint with or without abstraction.

- `check()` instruction checks whether there is no a garbage. It alternatively removes all unreachable TA.

  $f((F, \mathbb{G}, I)) = (F', \mathbb{G}, I_n)$ where F' is obtained from F by removing all unreachable TA.

The Instructions Not Modifying Forest Automata

- `alloc(`r_d`,`r_s`)` instruction creates a new instance of structure `Data` $D = (T, S, V, MB)$ that represents allocated memory. The size S is determined by an integer value in the source register r_s. The created instance will be further used for creation of a new TA and it is yet stored to the destination register.

  $f((F, \mathbb{G}, I)) = (F, \mathbb{G}[r_d := x], I_n)$ where $x = (\texttt{void_ptr}, \texttt{undef}, S', \emptyset)$ is an instance of structure Data such that $S' \in \mathbb{N}$ is a size stored in r_s.

- `load_cst(`r_d`,`c`)` instruction loads a constant c to the destination register r_d. This creates a new symbolic state which differs only in register contents.

  $f((F, \mathbb{G}, I)) = (F, \mathbb{G}[r_d := c], I_n))$ where c is a constant which is parameter of this instruction.

- `move_reg(`r_d`,`r_s`)` instructions creates a new symbolic state where a value from the source register r_s is moved to the destination register r_d.

  $f((F, \mathbb{G}, I)) = (F, \mathbb{G}[r_d := r_s], I_n)$.

- `bnot(`r_d`), inot(`r_d`)` instructions negates a value in the destination register and it creates a new symbolic state with the negated value in the destination register. `bnot` negates integer and `inot` negates the boolean value.

  $f((F, \mathbb{G}, I)) = (F, \mathbb{G}[r_d := \neg r_d], I_n)$.

- `move_reg_offs(`r_d`,`r_s`,`o`)` instruction accesses a tree automaton reference in the source register r_s and increases its displacement value by the given offset o. The new value is then stored in the destination register r_d.

  $f((F, \mathbb{G}, I)) = (F, \mathbb{G}[r_d := (Root, Displ + o)], I_n)$ where o is offset which is a parameter of this instruction and $r_s = (Root, Displ)$.

- `move_reg_inc(`r_d`,`r_{s_1}`,`r_{s_2}`)` instruction does the same thing as the previous one but moreover it increases a displacement by a value in the second source register r_{s_2}.

  $f((F, \mathbb{G}, I)) = (F, \mathbb{G}[r_d := (Root, Displ + o\ + r_{s_2})], I_n)$ where $r_{s_1} \in \mathbb{G}, r_{s_2} \in \mathbb{G}$ are the source registers such that $r_{s_1} = (Root, Displ)$, r_{s_2} is integer value.

- `get_greg(`r_{ld}`,`r_{gs}`)` loads a value from the global, source register r_{gs} to the local, destination register r_{ld}.

  $f((F, \mathbb{G}, I)) = (F, \mathbb{G}[r_{ld} := r_{gs}], I_n)$.

- `set_greg(`r_{gd}`,`r_{ls}`)` loads a value from the local, source register r_{ls} to the global, destination register r_{gd}.

  $f((F, \mathbb{G}, I)) = (F, \mathbb{G}[r_{gd} := r_{ls}], I_n)$.

- `get_`ABP`(`r_d`,`ABP`, `o`)` loads a root reference from the register ABP to the destination register r_d. The instruction also adds the offset o to a displacement in the root reference.

  $f((F, \mathbb{G}, I)) = (F, \mathbb{G}[r_d := (Root, Displ + o)], I_n)$ where $r_{ABP} = (Root, Displ)$ is the register containing root reference to FA representing local memory and o is an offset which is a parameter of this function.

- `get_`$GLOB$`(`r_d`,`$GLOB$`,`o`)` loads a root reference from the register ABP to the destination register r_d and adds the offset o to a displacement in the root reference.

  $f((F, \mathbb{G}, I)) = (F, \mathbb{G}[r_d := (Root, Displ + o)], I_n)$ where $r_{GLOB} = (Root, Displ)$ is the register containing root reference to FA representing the global memory state and o is an offset which is a parameter of this instruction.

- `load(`r_d`,`r_s`,`o`)` loads a value from a TA t pointed by a root reference in the source register r_s to the destination register r_d. The value is stored in a selector with displacement o in label of t.

  $f((F, \mathbb{G}, I)) = (F, \mathbb{G}[r_d := x], I_n)$ where $r_s = (Root, Displ)$, x is a value of the selector of T with displacement $displ$ T is a TA of F referenced by $ROOT$, $displ = Displ + o$, o is the offset which is a parameter of this instruction.

- `loads(`r_d, r_s, o`)` instruction does the same as `load` instruction but manipulates the structures.

  Formal definition is the same but the result stored in the selector is a structure.

- `load_`ABP`(`r_d`,`r_s`,`o`), load_`$GLOB$`(`r_d`,`r_s`,`o`)` instructions do the same as the previous instruction but loads a value from a TA pointed by r_{ABP} or r_{GLOB} register.

  Formally it could be defined as the previous instruction but ABP and $GLOB$ are used instead of r_s.

- `push_greg(`r_s`)` instruction creates a new global register r_g and fills it with a value in the source register r_s.

  $f((F, \mathbb{G}, I)) = (F, \mathbb{G} \cup \{r_g\}, I_n)$ where r_g is a new global register such that $r_g = r_s$.

- `pop_greg(`r_g`,`r_s`)` instruction takes a value in the last created global register and stores it to the source register. The global register is then deleted.

  $f((F, \mathbb{G}, I)) = (F, \mathbb{G} \setminus \{r_g\}[r_s := r_g], I_n)$ where r_g is the lastly created global register.

- `cond(`r_s`,`I_t`,`I_e`)` instruction represents a condition in the original code. It creates a new symbolic state with the same forest automaton and appends it to the queue. The new state can contain instruction I_t when the condition is true and *then* branch is performed. Otherwise, instruction I_e is used in the next state for case when the condition is false. The value of condition is stored in the source register r_s.

  $f((F, \mathbb{G}, I)) = (F, \mathbb{G}, r_s?I_t : I_f)$ where I_t is instruction used when r_s has true value and I_f is instruction used otherwise.

- `iadd(`r_d`,`r_{s_1}`,`r_{s_2}`), imull(`r_d`,`r_{s_1}`,`r_{s_2}`)` instructions do the integer addition and multiplication of the numbers in the source registers r_{s_1} and r_{s_2} and stores the result to the destination register r_d.

  $f((F, \mathbb{G}, I)) = (F, \mathbb{G}[r_d := r_{s_1} \otimes r_{s_2}], I_n)$ where $r_{s_1} \in \mathbb{G}, r_{s_2} \in \mathbb{G}$, $\otimes \in \{+, *\}$ are two source registers with values of integer type.

- `eq, neq, ge, gt, le, lt(`r_d`,`r_{s_1}`,`r_{s_2}`)` instructions makes a corresponding comparison of the values in the source registers r_{s_1} and r_{s_2}. The result of the comparison stores to the destination register r_d.

  $f((F, \mathbb{G}, I)) = (F, \mathbb{G}[r_d := r_{s_1} \otimes r_{s_2}], I_n)$ where $r_{s_1} \in \mathbb{G}, r_{s_2} \in \mathbb{G}$ are two source registers and $\otimes \in \{=, \neq, <, >, \leq, \geq\}$.

- `build_struct(`r_d`,`r_s $i_1 \cdots i_n$`)` instruction creates a structure from a content of the source registers $r_s, r_{s+i_1}, r_{s+i_2}, \ldots, r_{s+i_n}$ and the result stores to the destination register r_d. The source registers are defined by the index of the first register r_s and the vector of offsets $i_1 \cdots i_n$ from this register.

  $f((F, \mathbb{G}, I)) = (F, \mathbb{G}[r_d := x], I_n)$ where x is a structure created from the values of the registers $r_s, r_{s+i_1}, r_{s+i_2}, \ldots, r_{s+i_n}$. The values $i_1, i_2, \ldots, i_n$ are the offsets defined as the parameters of this instruction.

- `abort()` instruction aborts program execution.
- `assert(`r_s`,`c`)` instruction checks whether the value in the source register is the same one as the constant c.

  $f((F, \mathbb{G}, I)) = (F, \mathbb{G}, I)$ if $r_s = c$, otherwise symbolic execution is aborted. Constant c is a parameter of this instruction.

- `error()` instruction throws an exception representing a local error in program.
- `noret()` instruction quits program symbolic execution when no return function end is reached.

Chapter 5

Forester with VATA

This chapter describes the first enhancement of Forester done as the part of this thesis — replacing the underlying tree automata library by VATA. We discuss its difficulties and technical issues we had to handle during the design and implementation phase.

We decided to use the VATA library implementation of explicit encoding of TA because it is currently the only one that support the most of needed operations over TA, including language inclusion checking or removing useless states of TA. The semi-symbolic encoding is designed to tackle automata with large alphabets and since no large alphabets are used during the verification procedure the advantages of the semi-symbolic encoding would not be fully utilized here.

Forester implementation is currently far from maturity. The high structural dependencies is one of the bottlenecks. It makes difficult to change a module because the other modules are dependent on it internals such as the used data types. Therefore we first focused on reduction of dependencies between the classes (described in Section 5.1). Then we apply the design pattern *adapter* [9] (described in Section 5.2) to create an interface between Forester and the VATA library (implementation itself is described in Section 5.3). Application of adapter design pattern makes it possible to include VATA without the need of rewriting Forester to the names of methods and the data members used in VATA. It also creates only one particular place (adapter class) connecting Forester and VATA instead of including VATA into Forester classes. This prevents and removes a need of creating too strong relations between Forester and VATA.

5.1 Refactoring of Forester

The original Forester implementation of tree automata library has the strong dependencies to the other classes. Hence it was needed to refactore the implementation before creating adapter class for VATA API. The core class of the original tree automata is class *TA*. There are also the other related classes, e.g., class *TT* for representation of the transitions or class *Antichain* for language inclusion checking using the Antichain algorithm from [1]. This set of classes realizing original tree automata library will be further referred as *tree automata module*.

The used principles and patterns for refactoring are inspired by *generic programming* [28]. The refactoring is mainly based on reduction of a number of data types and data members declared to be *public* (in sense of the C++ programming language). This is done by exploiting the features provided by C++11 [29] which allows auto deduction of the mentioned data types in compilation time (using keyword *auto*). That enables us to make some of the data types of the original tree automata module *private*. *Iterator* is another concept often used in combination with auto deduction of types to bring higher genericity and reduce the number of dependencies to a concrete implementation. The combination of these two patterns are used for example for iteration over of all transitions of tree automata or over all transitions with the same state at the left-handed side. These kinds of iterations are quite common in Forester.

Another part of the refactoring consists of replacing a direct access to the class data members by the corresponding getters and setters. Reducing the structural dependencies is also done by

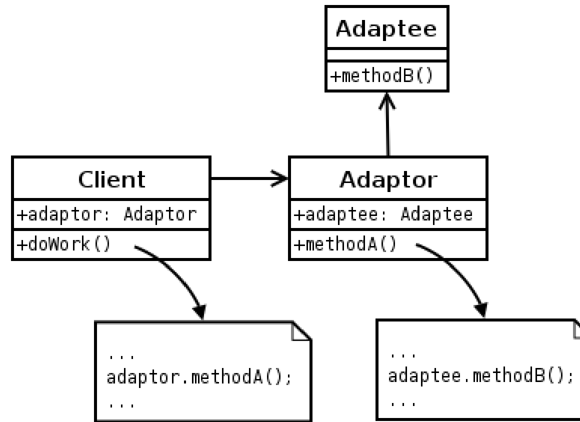


Figure 5.1: Adapter design pattern expressed in UML. Picture is taken from [31].

emphasis on application of *Law of Demeter* [21] which practically means that classes using TA explicitly should implement methods providing information about TA instead of providing instance of TA object itself. For instance, when one class wants to know whether a given state is a final state of a TA of a FA then the implementation of FA should implement a method providing this information instead of providing access to its private TA data member. That helps us to make the interface for TA module smaller.

5.2 Adapter Design Pattern

Adapter is a structural design patter [9] used for creating interface between incompatible classes. An UML diagram describing Adapter is shown in Figure 5.1. Adapter consists of a class *Adaptee* which is the class that we want to make compatible with another class *Client*. Client uses the methods of Adaptee. The *Adaptor* class is the one providing the connection between Client and Adaptee. There are two ways how to implement an Adaptor. The first one is to implement Adaptor as an inherited class from Adaptee and to employ the concept of inheritance to redirect method calls to its parent (with some possible preprocessing). The other implementation is composing Adaptee to Adaptor and using Adaptor like an interface to Adaptee. Adaptor could also add a new method that combines the methods of Adaptee to implement the wanted operations.

Applied to the case of Forester and VATA, Adaptee is the API of the VATA library, specifically, it is the class *ExplicitTreeAut* implementing the representation of tree automata in explicit encoding. Client is not only one class but the set of the Forester classes using TA library. Adaptor is a newly implemented class *VATAAdapter* described in the following section.

5.3 Implementation

The main part of our implementation of the adapter pattern is the new class *VATAAdapter* having the role of Adaptor. We decided to use such approach to Adaptor preferring the composition over the inheritance, since we often needs to rename methods. For instance, the name of a method in Forester differs from the name of the method in VATA with the same functionality. A conversion of a data type of a parameter of a tree automata operation is also sometimes needed, e.g. from vector to set.

The class *VATAAdapter* instantiates class *ExplicitTreeAut* from VATA as its private data member and redirects to this instance method calls from Forester (the names of methods of *VATAAdapter* are the same as they were in the original TA library). *VATAAdapter* also sometimes performs mentioned conversion of the data types. There are also the methods implemented by adapter not

presented in VATA, like the method *unfoldAtRoot*, performing unfolding. These methods are very Forester specific and so it is not sensible to add them to a general purpose library such as VATA and it is better to implement them in an interface like class *VATAAdapter*.

Originally, we were planning keep the original TA module along VATA adapter in order to be able to easily switch between the two implementations of TA. However, it has proven that it would bring high overhead in situations such as a conversion of some data types which is not necessary to do when the implementation using the data types compatible with VATA is used directly. Hence we decided to remove the original tree automata module and from this point, Forester supports the VATA library only.

Chapter 6

Backward Run and Predicate Abstraction in Forester

When an error in the analysed program is detected, it is necessary to check whether is it real or spurious. A spurious error can be caused by an overapproximating abstraction over foster automata. The analysis of spuriousness can be done by a *backward run* which, however, has not yet been designed and implemented for forest automata based shape analysis. An intersection of forest automata is crucial for the backward run but it has not been also developed yet. After proving that the error is spurious the abstraction is refined to avoid the spurious error detection by the same run again. The analysis is then restarted with the refined abstraction. This principle of gradual refinement of abstraction is based on the CEGAR framework [4].

As one of the main contributions of this work, we design and implement backward run including the intersection of forest automata and predicate abstraction [3] for forest automata based verification. Predicate abstraction can be refined by *predicates* obtained from backward run. It can be done is such way that the predicates are created directly to avoid the abstraction that caused the error and so the error will not be detected by the same run again. We restrict ourselves only to basic forest automata, not the hierarchical ones to keep this work within the bounds of master thesis.

The structure of this chapter is the following. The first Section 6.1 describes the backward run and predicate abstraction in general. Section 6.2 describes the algorithm for the intersection of non-hierarchical forest automata which is important part of the backward run. Section 6.3 contains the design of backward run and Section 6.4 describes the design of predicate abstraction for forest automata. Section 6.5 covers the implementation.

6.1 Backward Run and Predicate Abstraction

We described in Section 3 that the abstract transformers τ_{op} related to the concrete program operations are gradually applied to forest automata starting from a forest automaton modelling an empty heap. This gradual application of abstract transformers in the order of the concrete program operations is called *forward run*. Formally, a forest automaton representing the memory state at program point n is obtained by $\tau_{op}^{n}(\tau_{op}^{n-1}(\ldots\tau_{op}^{1}(F_{Empty})\ldots))$ where τ_{op}^{i} is an abstract transformer related to the concrete program operation at the i-th point of the program execution and F^{empt} is a forest automaton modelling the (initial) empty heap. Normalization, (un)folding, and abstraction are done along with the abstract transformers at some program locations, as described in Chapter 3.

Consider that an error is detected at the n-th program location during the forward run. The verification of non-spuriousness of the error is done by a gradual application of *reverse abstract transformers* τ_{-op} over forest automata. The backward run starts from the error line and then checks whether an intersection of the languages of forest automata from forward and backward run is non-empty at each program line. When the intersection is non-empty the backward run continues using the automaton representing the intersection of the languages. If the backward run reaches

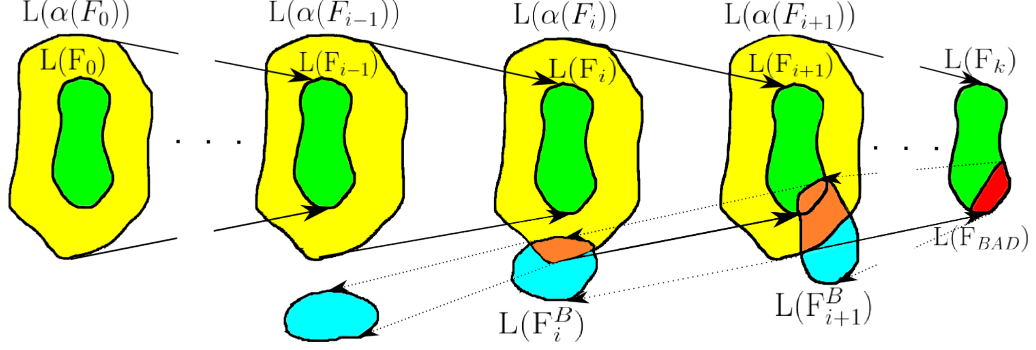


Figure 6.1: Figure is taken from [3]. Figures illustrates principle of forward and backward run and a detection of a spurious counterexample.

the beginning of the program without detection of an empty intersection of the languages, the error is real, otherwise is spurious. This process is called *backward run*. We use $F_A \cap F_B$ to denote a forest automaton with language $L(F_A) \cap L(F_B)$. Formally, we compute forest automata by application $\tau^1_{-op}(\tau^2_{-op}(\dots\tau^n_{-op}(F_n \cap F_{BAD})\dots \cap F_3) \cap F_2)$ from the n-th point back to the beginning where F_i is the forest automaton at the i-th point of the forward run and F_{BAD} is the forest automaton representing heap configuration causing the detected error. It is checked whether $\forall\, 0 \leq i < n : L(\tau^1_{-op}(\dots\tau^i_{-op}(F_i \cap F_i^B)\dots \cap F_2)) \cap L(\tau^i_{op}(\dots\tau^1_{op}(F_{Empty})\dots)) \neq \emptyset$ where F_i^B is the forest automaton at i-th point of the backward run. When the intersection is empty the found error is reported as spurious and if no empty intersection is found the error is real.

This principle is illustrated in Figure 6.1. Figure shows how a forest automaton (and its language) is gradually changed by abstract transformers until an error is found. The backward run starts from the error configuration and reverses the abstract transformers until the empty intersection between the forward and backward FA languages is get at the point $i-1$. Then the error is reported as spurious. Forest automata are in the figure represented by their languages which are shown as the colored areas. The figure shows FA of a symbolic states in the forward run using the green (language of FA before abstraction) and the yellow (language of FA after abstraction) areas. There are k symbolic states in the figure. Each step of the forward run does abstraction of FA, e.g., FA $\alpha(F_i)$ is obtained by the abstraction of FA F_i. Then the abstract transformer is applied to the abstracted FA and the next symbolic state with FA F_{i+1} is obtained. An abstract transformer is illustrated by the arrows connecting the yellow area of one state with the green area of the next state. The error configuration in language $L(F_{BAD})$ is shown as the red area. The backward run starts from the error configuration and applies the reverse abstract transformer (illustrated by the dotted arrows) to FA representing the intersection (the orange areas) of the languages of the forest automata from the forward (the green and yellow areas) and backward run (the blue areas). The result of a reverse abstract transformer is a new backward run state with the FA having language shown as a blue and an orange area. It is found that the error is spurious because the intersection is empty at the level $i-1$.

Predicate abstraction has not yet been used in forest automata based verification because it is meant to work with backward run which detects the predicates to refine predicate abstraction in the case of a spurious counterexample detection.

The used predicate abstraction is defined over a TA as in [3] and its extension to FA will be described later. The predicate abstraction is parameterized by the set of predicates $\mathcal{P} = \{P_1, \dots, P_n\}$. A predicate $P \in \mathcal{P}$ is a language represented by a tree automaton. As we described in Section 3.2.3, the abstraction function α over a set of states Q of TA $A = (Q, \Sigma, \delta, R)$ basically merges states in the same equivalence class of an equivalence relation. The predicate abstraction function is denoted by $\alpha_{\mathcal{P}}$. The predicate abstraction introduces an equivalence relation $\sim_{\mathcal{P}}$ where the states of tree automaton are equivalent when their languages have an non-empty intersection with the same predicates. Formally, $\forall q_1, q_2 \in Q : q_1 \sim_{\mathcal{P}} q_2 \Leftrightarrow (\forall P \in \mathcal{P} : L(A, q_1) \cap P \neq \emptyset \Leftrightarrow L(A, q_2) \cap P \neq \emptyset)$.

The refinement of predicate abstraction is done by extending the set of predicates by adding

tree automata languages from a FA in backward run that has empty language intersection with the related FA from the forward run. Then the new forward run is started using the extended predicate set for the abstraction. The new predicates will prevent merging the states at the points that caused detection of the spurious error again. When the refinement from [3] is used then the termination of the method is guaranteed.

6.2 Intersection of Forest Automata

As an important part of the backward run, we design and implement the algorithm for intersection of languages of non-hierarchical forest automata. The algorithm is based on the algorithm for the top-down intersection of tree automata and extends its idea to forest automata.

The main part of the method is given in Algorithm 1. The algorithm takes two non-hierarchical forest automata $\mathcal{F}_\mathcal{A}$ and $\mathcal{F}_\mathcal{B}$ with the same number of port indices as its input and outputs a forest automaton $\mathcal{F}_\mathcal{C}$ such that $\mathcal{L}(\mathcal{F}_\mathcal{A}) \cap \mathcal{L}(\mathcal{F}_\mathcal{B}) \subseteq \mathcal{L}(\mathcal{F}_\mathcal{C})$. The reason why $\mathcal{F}_\mathcal{C}$ overapproximates language of the intersection is given after the description of the algorithm.

The algorithm starts with the initialization of the auxiliary data structures *processed* (where the processed roots of the resulting FA are stored), *cnt* (what is counter of TA of the resulting FA) and the data structure for the resulting automaton F on lines 1-3. Then it iterates over all port indices of the input FA and over all roots of TA at the positions in the input FA given by the port indices and it calls the function *intersect* over the root states (one from each TA) and their TA.

The function *intersect* takes as the input parameter these two TA and one state from each of TA. We use *lhs* to denote one of the inputs, *lhs.aut* to denote the TA itself, and *lhs.state* to denote its state. The same notation is used for the second input parameter *rhs* of the function. The function *intersect* constructs TA of the resulting forest automaton. The states of the resulting automaton are so-called *product states* — pairs (r, s) where r is a state of $\mathcal{F}_\mathcal{A}$ and s is a state of $\mathcal{F}_\mathcal{B}$. The function returns the index of the newly created TA in the resulting FA.

The function *intersect* starts with creating the root point (lhs, rhs) and adding it to the set of processed root points if it is not already processed. Otherwise it returns the index of TA with such root point as the root. This is done on lines 1-3. Then the counter *cnt* of the created TA of the resulting FA is increased and the stack *workset* is initialized to serve as an auxiliary data structure where the product states are stored to be processed later on lines 4-8.

The function *intersect* creates a new TA with the only product state $(lhs.state, rhs.state)$ and with the empty transitions set on line 8. The product states from the workset are gradually processed in the cycle beginning on line 9. We denote (r, s) the actually processed product state. We call *referencing* the states that are at the left-handed side of transitions labeled by root references. The *normal* states are states that are not referencing states. There are several cases of how the product state is managed by.

Algorithm 1: Intersection for non-hierarchical forest automata

---

Input: $\mathcal{F}_\mathcal{A} = (\mathcal{A}_1 \cdots \mathcal{A}_n, \pi^\mathcal{A}_k \cdots \pi^\mathcal{A}_k)$, $\mathcal{F}_\mathcal{B} = (\mathcal{B}_1 \cdots \mathcal{B}_m, \pi^\mathcal{B}_k \cdots \pi^\mathcal{B}_k)$
Output: $\mathcal{F}_\mathcal{C} = (\mathcal{C}_1 \cdots \mathcal{C}_p, \pi^c)$ such that $\mathcal{L}(\mathcal{F}_\mathcal{A}) \cap \mathcal{L}(\mathcal{F}_\mathcal{B}) \subseteq \mathcal{L}(\mathcal{F}_\mathcal{C})$

```
1 processed := ∅;
2 cnt := 0;
3 F := empty FA;
4 for i := 1 to k do
5     forall the root states r of A_{π_i^A} do
6         forall the root states s of B_{π_i^B} do
7             intersect((r, A_{π_i^A}), (s, B_{π_i^B}));
8 return F;
```

Function intersect(lhs, rhs)

Input: $lhs = (lhs.aut, lhs.state)$ and $rhs = (rhs.aut, rhs.state)$ are pairs of an automaton and a state from the automaton

Globals: F: an FA; *processed*: set of processed root points; *cnt*: counter of components of F

Output: Index of the resulting component in F

```
1  if (lhs, rhs) ∈ processed then
2      return index of the component for (lhs, rhs)
3  processed := processed ∪ {(lhs, rhs)};
4  index := cnt;
5  cnt := cnt + 1;
6  workset := createStack();
7  workset.push((lhs.state, rhs.state));
   // start creating the new TA
8  Q := {(lhs.state, rhs.state)}; Δ := ∅;
9  while ¬workset.empty() do
10     (r, s) := workset.pop();
11     if r --ref(i)--> then // r is a root reference
12         if s --ref(j)--> then // both r and s are root references
13             cp := intersect((i, root(i)), (j, root(j)));
14         else // only r is a root reference
15             cp := intersect((i, root(i)), (rhs.aut, s));
16         Δ := Δ ∪ {(r, s) --ref(cp)-->};
17     else if s --ref(j)--> then // only s is a root reference
18         cp := intersect((lhs.aut, r), (j, root(j)));
19         Δ := Δ ∪ {(r, s) --ref(cp)-->};
20     else // for normal states
21         foreach a ∈ Σ (in the reverse order than given by ⪯_Σ) do
22             foreach (r_1, ..., r_#a) ∈ post_a(r), (s_1, ..., s_#a) ∈ post_a(s) do
                   // the ordering should not be important here
23                 Δ := Δ ∪ {(r, s) --a--> ((r_1, s_1), ..., (r_#a, s_#a))};
24                 for i := #a to 1 do // push in the reverse order to keep DFT
25                     if (r_i, s_i) ∉ Q then
26                         Q := Q ∪ {(r_i, s_i)};
27                         workset.push((r_i, s_i));
28 F.append((Q, Σ ∪ ℕ, Δ, {(lhs.state, rhs.state)}));
29 return index;
```

1. Both r and s are referencing states. The function *intersect* is called recursively on the referenced root states and creates a new TA with the index *cp*. Finally a transition with the product state (r, s) at the left-handed side, that is labeled by the root reference to a TA with index *cp*, is added to the transition set of the new TA. This is on lines 11-13.

2. The one of states r or s is a referencing state and the other one is normal. The function *intersect* is called with the referenced root state as one parameter and the second parameter is the normal state. The result of this function call is the position *cp* of a new TA at the resulting FA created by this call. Then a transition with the product state (r, s) at the left-handed side that is labeled by the root reference to a component at the position *cp* is added to the new TA. These cases in the algorithm correspond to lines 14-16 and 17-19.

3. The both r and s are not root references but the normal states. Then it is iterated over all

$a \in \Sigma$ with $\#(a) = n$ and a new transition t is added to the transition relation Δ, where $t = (r, s) \xrightarrow{a} ((r_1, s_1) \cdots (r_n, s_n))$ such that $(r_1 \cdots r_n) \in post_a(r)$, $(s_1 \cdots s_n) \in post_a(s)$ and $post_a(q) = \{(q_1 \cdots q_n) \in Q^n \mid q \xrightarrow{a} (q_1 \cdots q_n) \in \Delta\}$. Finally, when the product state (r_i, s_i), $1 \leq i \leq n$, is a new one, then it is added to Q and appended to the workset for a further processing. This last case corresponds to lines 20-27 of the function *intersect*.

The function *intersect* finally adds the created TA with the root $(lhs.state, rhs.state)$ to the resulting FA and returns the index of this TA in the resulting FA, which is done on lines 28-29.

Algorithm 1 returns the FA $\mathcal{F}_\mathcal{C}$ overapproximating the intersection $\mathcal{L}(\mathcal{F}_\mathcal{A}) \cap \mathcal{L}(\mathcal{F}_\mathcal{B})$. The over-approximates comes from non-determinism causing that a referencing state $r_\mathcal{A}$ from $\mathcal{A}$ may form more then one product state with the different states $r_\mathcal{B}$ from $\mathcal{B}$. This ambiguity in mapping among the states can led to a generation of a graph where one cut-point appears more then once what is not possible. It can be resolved by splitting FA $F_\mathcal{C}$ to several FA. For each possible bijection between the referencing states of $\mathcal{F}_\mathcal{A}$ and the states of $\mathcal{F}_\mathcal{B}$ it is created a particular FA. Each of FA corresponds to one maximum bijection states. The product states that contract the chosen bijection are removed from FA. The useless states are then removed from each FA. We continue for each FA separately.

It is also necessary to care about the case when a referencing state r of one FA is paired with a normal state s of the second FA and the product state (r, s) appears more then once in a run of TA or in more then one TA. This would mean that the product state, which is the root, is referenced more then once what cannot happen since s is a normal state. To guarantee that this will not happen we create the new FA $F = (T_1 \cdots T_n, \pi)$ in the following way. For each $1 \leq i \leq n : T_i$ and for each (r, s) in T_i there is created FA $F = (T_1 \cdots T_n, \pi)$ such that (r, s) is not in any other TA T_j, where $i \neq j$, and (r, s) is not in any run of T_i more then once. We remove useless states in every FA F and throw away FA with the empty language. We get as a result of the intersection of FA the set of FA representing the resulting language.

6.3 Designing Backward Run over Symbolic States

We describe now the design of the backward in Forester. A general overview of the backward run was given in the beginning of this chapter.

It was mentioned that abstract transformers in Forester implements the instructions of its microcode in an abstract domain. The input C program is translated to microcode instructions $T = I_1^F, I_2^F, \ldots, I_{n+1}^F$.

The *forward run* is done by going through T in the forward order and executing the abstract transformers implementing the instructions in T over symbolic states. It generates the *forward trace* of symbolic states $FT = S_0^F, \ldots, S_n^F$. The state S_i^F is the input of the abstract transformer τ_{op}^{i+1} where $0 \leq i \leq n$ and S_{i+1}^F is the resulting symbolic state.

Assume that an error is detected in the symbolic state S_n^F. Then the backward run starts from this symbolic state and it is done by going through T in the reverse order executing the reverse transformers τ_{-op}^i over the intersection S_i^B and S_i^F obtaining S_{i-1}^B where $0 < i \leq n$. The intersection of S_i^B and S_i^F is a symbolic state that is identical to S_i^B but has an FA representing a language that is an intersection of languages of FA of both states. The backward run generates the *backward trace* $BT = S_0^B, \ldots, S_n^B$. Each τ_{-op}^i implements *reverse instruction* I_j^B and it is reversion of τ_{op}^i which implements the instruction I_j^F.

It is not necessary to compute FA with the language $L(\mathcal{F}_i^F) \cap L(\mathcal{F}_i^B)$ to get the next symbolic state in the backward run (and to check spuriousness of an error) after the execution of each reverse instruction in the backward run. Indeed, it is sufficient to compute intersection only for the microcode instructions `FI_abs(), FI_fix()` because only these instructions do an abstraction of FA which can lead to a spurious error. The other instructions are precise and elementary enough that they could be reversed without need of intersection.

The next sections provides the description of the inverse microcode instructions. We describe each reverse instruction first informally and then define it formally as the function $g : \mathbb{S} \times \mathbb{I} \rightarrow \mathbb{S}$.

The function g should have the reverse semantics to the forward function f from Section 4.2.3. The function models behavior of the reverse transformer implementing a reverse instruction with a given symbolic state as an input. We further use I_P to denote an instruction I_{i-1} that precedes the instruction I_i. The symbolic states, their registers and instructions have the same role as they have in Section 4.2.3.

The Instructions With Special Reverse Semantics

This section provides a description of instructions such that the (forward) abstract transformers implementing them manipulates forest automata or do an operation that is not similar to any other instruction.

- `FI_node_create(`r_s`, `r_d`)` does the reversion the C function `malloc`. The instruction gets a value from the source register r_s in S_i^F where i is order of this instruction in T. If the value is a reference or a null pointer then the new state is just copy of S_i^B.

  Otherwise the value type is void pointer. The new symbolic state registers have the same values in the registers as the state S_i^B with the exception of the register r_d. The value of this register is substituted by the value r'_d of S_i^F. The substitution is done because the register r_d contains reference to the TA t representing the heap pointed by the allocated pointer. TA t is then removed from FA and all references to it are invalidated.

  $g((F, \mathbb{G}, I^B)) = (F', \mathbb{G}[r_d := r'_d], I_P^B)$ where $(F, \mathbb{G}, I^B) = S_i^B$, $r_d \in S_i^B$, $r'_d \in S_i^F$ and F' is obtained from T by removing TA t referenced by the register r_d and invalidating all references to it.

- `FI_node_free` reverses freeing the memory. The removed TA representing a heap pointed by a freed pointer is referenced by the root reference RR in the register r_d. This TA is copied from FA F^F of S_i^F to FA F^B of S_i^B into the corresponding position. It is also necessary to relabel the selectors of F^B according to F^F to replace undefined values created after free by the root reference to the renewed FA.

  $g((F, \mathbb{G}, I^B)) = (F', \mathbb{G}, I_P^B)$ where $(F, \mathbb{G}, I^B) = S_i^B$ and $F' = l(F \cup \{T\})$ where T is the TA referenced by RR in the register r_d of S_i^F and l performs relabeling such that if $r \xrightarrow{undef} () \in F \land r \xrightarrow{RR} () \in F^F$ then $r \xrightarrow{undef} ()$ is replaced by $r \xrightarrow{RR} ()$.

- `FI_store(`r_d`)` creates a new symbolic state identical to the S_i^B. Then the root reference $RR = (Root, Displ)$ is loaded from the register r_d of S_i^B. The value V from a node pointed by the selector with displacement $Displ$ in FA of S_i^F referenced by $Root$ is loaded. Finally, the value V is stored to the node of FA of the new symbolic state. The node is pointed by the selector referenced by RR.

  $g((F, \mathbb{G}, I^B)) = (F', \mathbb{G}, I_P^B)$ where $(F, \mathbb{G}, I) = S_i^B$ and F' is obtained by the described operation of storing value to the node of FA.

- `FI_stores` has the same semantics as the previous one but for storing a structure to a FA.

- `FI_abs()`, `FI_fix()` creates a new symbolic state identical to the state S_i^B. Then a new FA created by the intersection of FA from S_i^B and S_i^F is used for the new symbolic state.

  $g((F, \mathbb{G}, I^B)) = (F', \mathbb{G}, I_P^B)$ where $(F, \mathbb{G}, I) = S_i^B$, $F' = F \cap F_{FWD}$ and F_{FWD} is FA of S_i^F.

- `FI_push_greg(`r_g`)` creates a new symbolic state identical to the state S_i^B but the last global register added S_i^F is removed from the new state.

  $g((F, \mathbb{G}, I^B)) = (F, \mathbb{G} \setminus \{r_g\}, I_P^B)$ where $S_i^B = (F, \mathbb{G}, I)$ and r_g is the last global register added to S_i^F.

- `FI_pop_greg(`r_g`)` creates a new state identical to the S_i^B, then the register r_g from S_i^B is moved to the forest automaton of the new state and finally the original value of r_g in S_i^F is stored to the corresponding register in the new state.

  $g((F, \mathbb{G}, I^B)) = (F, \mathbb{G} \cup \{r_g\}, I_P^B)$ where $S_i^B = (F, \mathbb{G}, I)$ and r_g is the last global register removed from S_i^F.

- `FI_set_greg(`r_g`)` creates a new symbolic state identical to the S_i^B. It loads an old value of the global register r_g from S_i^F and assigns it to the register r_g.
  $g((F, \mathbb{G}, I^B)) = (F, \mathbb{G}[r_g := r'_g], I_P^B)$ where $S_i^B = (F, \mathbb{G}, I)$ and r'_g is the register of S_i^F.
- `FI_abort` is not reversible.
- `FI_acc_sel` has an empty semantics. The original instruction isolates a selector and so it can create a new TA. It is not necessary to reverse this operation since it does not change the language of FA of the symbolic state or anything else in the symbolic state.
  $g((F, \mathbb{G}, I)) = (F, \mathbb{G}, I)$
- `FI_acc_set,FI_acc_all` are the same as the previous instruction and so the semantic of these instructions is empty.
  $g((F, \mathbb{G}, I)) = (F, \mathbb{G}, I)$

Register Assignment Instructions

The instructions in the list bellow perform simply an assignment to the register. Reversing them consists of creating a symbolic state S' with the same values of registers as the symbolic state S_i^B and substituting of a value of the original destination register r_d of S' by the value of the same register in the state S_i^F (denoted by r'_d). Formally, $g((F, \mathbb{G}, I^B)) = (F, \mathbb{G}[r_d := r'_d], I_P^B)$ where $(F, \mathbb{G}, I^B) = S_i^B$ and $r'_d \in S_i^F$.

- `FI_load_cst`
- `FI_move_reg`
- `FI_bnot, FI_inot`
- `FI_move_reg_offs`
- `FI_move_reg_inc`
- `FI_get_`ABP
- `FI_get_`$GLOB$
- `FI_load`
- `FI_load_`ABP`, FI_load_`$GLOB$
- `FI_get_greg`
- `FI_loads`
- `FI_alloc`
- `FI_iadd, FI_imull`
- `FI_eq, FI_neq, FI_ge, FI_gt, FI_le, FI_lt`
- `FI_build_struct`

Void Instructions

The following instructions have reverse semantics that is identity to the input symbolic state because they do not change the symbolic state in forward run. The symbolic state obtained by the reverse instruction is the same as the symbolic state S_i^F. Formally written, $g((F, \mathbb{G}, I^B)) = (F, \mathbb{G}, I_P^B)$ where $S_i^B = (F, \mathbb{G}, I^B)$.

- `FI_cond`
- `FI_check`
- `FI_assert`
- `FI_error`
- `FI_noret`

6.4 Designing Predicate Abstraction over Forest Automata

The predicate abstraction from [3] defined for tree automata was already introduced. We extend the concept to forest automata in automata component wise way. This means that the abstraction is applied to each tree automaton of forest automaton in the current symbolic state separately. Formally, let $\alpha_{\mathcal{P}}^{\mathcal{F}} : \mathcal{F} \rightarrow \mathcal{F}$ be an abstract function where $\mathcal{F}$ is a domain of forest automata and $\mathcal{P} = \{P_1, \ldots, P_n\}$ is a set of predicates. The function $\alpha_{\mathcal{P}}$ is defined as follows. Let $F_1 = (A_1 \cdots A_n, \phi^A)$ and $F_2 = (B_1 \cdots B_n, \phi^B)$ be two forest automata and let $\alpha_{\mathcal{P}}$ be an abstraction function merging states of a tree automaton according to the equivalence relation $\sim_{\mathcal{P}}$ defined in Section 6.1. Then $\alpha_{\mathcal{P}}^{\mathcal{F}}(F_1) = F_2 \Leftrightarrow \forall i \in \{1, \ldots, n\} : B_i = \alpha_{\mathcal{P}}(A_i)$.

The states are merged only in one tree automaton and not across the different tree automata of forest automata using the proposed method. However, merging the states of different tree automata can be more efficient (since a forest automaton with smaller number of states is created). This work proposes the first version of predicate abstraction and an optimization of the new method could be done as the future work.

The predicates from $\mathcal{P}$ are languages represented by tree automata. We use compressing representation from [3]. A tree automaton $A = (Q, \Sigma, \Delta, R)$ represents a set of predicates where the predicates are the languages $\mathcal{L}(A, q)$ of the states $q \in Q$. Checking whether a state of a tree automaton has non-empty intersection with the same sets of predicates as another state of the automaton is done by the following method. Consider a TA A and the set of predicates $\mathcal{P}$ that are represented by the set of TA $T = \{A_1^{\mathcal{P}}, \ldots, A_m^{\mathcal{P}}\}$. Recall that the TA can represent a set of predicates. We create the product automata $A \times A_1^{\mathcal{P}}, \ldots, A \times A_m^{\mathcal{P}}$ in the bottom-up way manner [5].

Each of product automata $A \times A_i^{\mathcal{P}}$, where $1 \leq i \leq n$, has the state set consisting of the product states of the form $(p, q) \in Q_A \times Q_{A_i^{\mathcal{P}}}$. To easier the construction of $\sim_{\mathcal{P}}$, we introduce the auxiliary function m that labels each state of A by the states of the predicate tree automata. Formally, the function $m : Q_A \rightarrow 2^{Q_{A_1^{\mathcal{P}}} \cup \ldots \cup Q_{A_m^{\mathcal{P}}}}$ is defined as follows: $m(p) = \{q \in Q_{A_1^{\mathcal{P}}} \cup \ldots \cup Q_{A_m^{\mathcal{P}}} \mid \exists A \times A_i^{\mathcal{P}} : (p, q) \in Q_A \times Q_{A_i^{\mathcal{P}}}\}$. The equivalence relation $\sim_{\mathcal{P}} \subseteq Q \times Q$ is then constructed such that $(p_1, p_2) \in \sim_{\mathcal{P}} \Leftrightarrow m(p_1) = m(p_2)$. The application of the function $\alpha_{\mathcal{P}}^{\mathcal{F}}$ can lead to merging the states of A.

When a found error is considered to be a spurious one, then the abstraction needs to be refined by creating the new predicates and adding them to the current set of the predicates $\mathcal{P}$. The new predicates are created as follows. Consider a point of an analysed program where the spurious error is detected at the $(i-1)$-th step of the backward run because $\mathcal{L}(F_{i-1}^F) \cap \mathcal{L}(F_{i-1}^B) = \emptyset$ where FA F_{i-1}^F is from the forward run and FA F_{i-1}^B is from the backward run. We create a normalized FA $F_i^{FN} = (A_1 \cdots A_n, \phi^A)$ and a normalized $F_i^{BN} = (B_1 \cdots B_m, \phi^B)$ by normalization of F_i^F and F_i^B from the i-th step. It can be supposed that $n = m$ because normalization transforms the both FA to an uniform form. Then the set of the new predicates $\mathcal{P}_{new} = \{B_i \in F_B^N \mid \mathcal{L}(A_i) \cap \mathcal{L}(B_i) = \emptyset\}$ is obtained. The symbolic execution is then restarted again with the $\mathcal{P} = \mathcal{P} \cup \mathcal{P}_{new}$.

6.5 Implementation

This section provides the description how the described methods have been implemented in the Forester tool. We describe the implementation of backward run in Section 6.5.1, the implementation of intersection of symbolic states in Section 6.5.2, and predicate abstraction in Section 6.5.3.

6.5.1 Backward Run

The backward run can be enabled by setting the macro `FA_BACKWARD_RUN` in file *config.h* to 1. If this macro is set to value 1, then the backward run is executed by class *SymExec* on an error detection.

The main method of backward run is called *isSpuriousCE*. It checks whether a given error is a spurious or not. This method is implemented in class *BackwardRun* — the wrapper class of backward run.

As it was described in Section 6.3, the backward run is performed by going through the given forward trace in the backward order and executing reverse abstract transformers implementing the reverse instructions. The method *reverseAndIsect* has been added to class *AbstractInstruction* for modelling a reversion of instruction. The class *AbstractInstruction* is the parent class of the inheritance hierarchy of the classes implementing microcode instructions. Each microcode instruction has to implement this method if the parent class does not already implement it. The method *reverseAndIsect* basically implements the reverse abstract transformer of a given instruction. Despite the name of the method, the intersection of forest automata is not done by each microcode instruction but only in the cases described in the previous section. The method *reverseAndIsect* returns a new symbolic state that is used as the next symbolic state in the backward run.

The method *isSpuriousCE* returns `true` whenever a symbolic state returned by the method *reverseAndIsect* contains a forest automaton with an empty language. In such a case, it also creates the new predicates. When the method reaches the beginning of the forward trace, then the found error is real and the `false` is returned.

6.5.2 Intersection of Forest Automata

The method *reverseAndIsect* of class *FixpointBase* performs the intersection of forest automata. This class is inherited by the classes implementing microcode instructions `FI_abs, FI_fix` which both compute the fixpoint. The intersection of forest automata is implemented by class *SymState* representing a symbolic state. Moreover, it contains information about the variables in the current symbolic states, which is needed by the implementation of the algorithm for forest automata intersection.

The main method for symbolic state intersection is called *Intersect*. It computes the intersection between the symbolic states. The parameters of the intersection are the symbolic state, that calls the method *Intersect*, and the symbolic state given as a parameter. The result of the intersection replaces the content of the calling object.

Another operation implemented for the purposes of the backward run is the method *SubstituteRefs*. It performs substitution of the root references, which is needed for reversing the instruction `FI_free`. The root supposed to be substituted and reference for substitution are given in the parameters of this method. The method is again member of class *SymState*.

6.5.3 Predicate Abstraction

The predicate abstraction is implemented by a method *predicateAbstraction*, a member of class *Abstraction*. This method takes a set of tree automata representing the set of predicates as its parameter. The abstraction is performed over a forest automaton which is a data member of class *Abstraction*. The abstraction is chosen by setting value of macro `FA_USE_PREDICATE_ABSTRACTION` to 1. When predicate abstraction is chosen, the method *predicateAbstraction* is called on `FA_ABS` instruction execution.

The new predicates used by predicate abstraction are created during the testing whether a found error is spurious or real. Particularly, the method *isSpuriousCE* creates the predicates when the spurious error is detected using the method described in Section 6.4. It is necessary to perform the intersection between TA representing predicates and TA of FA from symbolic state of backward run. The bottom-up intersection algorithm of tree automata was implemented in VATA for this purpose because VATA contained only a top-down algorithm for computing intersection which is not suitable for our purposes.

The first run of predicate abstraction should be done with an empty set of predicates. This can lead to a very strong abstraction where the most TA states would be merged. This would lead to a trivial spurious counterexamples found in the first run of the analysis. Therefore we use height abstraction with height 1 (what is also the original configuration of abstraction used in Forester) for the first run of symbolic execution. When a spurious error is detected in this run then the new predicates are created and the next runs of symbolic execution use predicate abstraction. Moreover, the behavior of the new implementation is different only after the detection of a spurious error.

Chapter 7

Experimental Evaluation

This chapter summarizes the experimental evaluation of the results of this work. First, we compare a performance of the versions of Forester without and with the VATA library. Then we evaluate the backward run on the SV-COMP benchmark and finally the version of Forester using predicate abstraction is compared to the original one with height abstraction. We also discuss the programs that were newly verified by Forester because it was not possible to analyze them without the backward run and predicate abstraction.

All of the experiments have been evaluated on the computer with CPU Intel Core 2 Duo (2.13 GHz) and 4 GiB memory, using Debian Linux, testing version. We used g++, version 4.9, for compiling Forester and gcc, version 4.8, for compiling the analysed programs. The measured time is the CPU time. The presented times are the averages of the five measurings. The evaluations over SV-COMP benchmark are done following the rules of SV-COMP competition [24].

7.1 The Evaluation of Forester with VATA

This section provides a comparison of the early version of Forester that uses its own tree automata library and the version using VATA. Since the version without VATA is no longer compatible with the new version of Forester, we can only evaluate the performance on the programs that are possible to analyze also with the older version of the tool without backward run and predicate abstraction. Therefore we use height abstraction of the height 1, which was the original configuration of Forester before this work started.

The first performance tests were performed over the regression test suite of Forester. The results of these tests are summarized in Table 7.1. There are two cases that were measured. The first one is the case when Forester is compiled with the third level of optimization of the compiler. The original implementation outperformed us by the factor of two. Since we expected that VATA as an optimized library should be more efficient then Forester implementation of tree automata, we tried to verify these results by compiling Forester and VATA without optimization and running the tests again. We found that with the optimization turned off, Forester with VATA is more efficient. The are two possible explanations of the the worse performance of Forester with VATA in the case when optimizations of compiler are used. The first one is that the original implementation of tree automata in Forester uses simulation to reduce the size of tree automata what is not done in version with

Table 7.1: Comparison of Forester with and without VATA on the regression tests of Forester.

Compiler optimization	without VATA [s]	with VATA [s]
-O3	9.00	17.00
Default	65.00	43.00

Table 7.2: Comparison of Forester with and without VATA on the selected cases. Compiled with g++ default optimizations.

Version	without VATA [s]	with VATA [s]
SLL with CSLL	7.77	5.44
DLL with CDLL	13.23	10.70
Skiplist, the 3rd level	173.07	185.70
Skiplist, the 2nd level	3.86	1.90
Linux driver snippet	9.931	5.08

Table 7.3: Comparison of Forester with and without VATA on the selected cases. Compiled with g++ with level -O3 of optimizations.

Version	without VATA [s]	with VATA [s]
SLL with CSLL	0.85	2.07
DLL with CDLL	1.43	4.03
Skiplist, the 3rd level	15.00	46.00
Skiplist, the 2nd level	0.40	0.60
Linux driver snippet	1.07	1.92

VATA because the reduction based on simulation in VATA is not ready yet. However, this does not explain why the original implementation is more efficient only when the advanced compiler optimization are used. The second possible explanation of the slowdown is that VATA is linked to Forester as a static library so the compiler cannot perform the advanced optimization between Forester and VATA, as opposed to Forester using its own tree automata module.

We decided to further explore this problem by comparing the particular programs. These programs contain manipulation of singly-linked lists of circular singly-linked lists, doubly-linked lists of circular doubly-linked lists, the skip-lists of the 2nd and the 3rd level and also a snippet from the Linux drivers. Not all of these programs are included in the Forester regression test set.

The results comparing Forester with and without the VATA library compiled without the compiler optimization are in Table 7.2. The version of Forester with VATA is faster in this case. The only exception is a verification of the skip-list of the third level. We suppose the original implementation takes the advantage of reduction of tree automata because tree automata with the big number of states are created during it verification. The big number of the generated states is related to the fact that the program manipulating the skip-list of the third level is the probably the most difficult case from the whole Forester testing benchmark.

The comparison of the two versions of Forester compiled with the compiler optimization on the same programs are in Table 7.3. Here, the original implementation of Forester is faster. The biggest difference is in the time needed for the verification of the skip-list of the third level. It has probably the same cause as in the version without optimization. Putting the results together, we assume that the main performance loss for Forester using VATA comes from the inability of compiler to perform the advanced optimization. The fact that VATA does not reduce tree automata based on simulation relation has the impact probably mainly in the demanding cases such as verification of skip-list. However, employing the algorithm for reduction of tree automata in VATA would be needed in future to prevent inefficiency in verification of the programs where FA with many states are generated.

Although Forester using VATA is slower in some cases, the slowdown is not as significant to prevail the advantages of the better modularity and maintainability coming from having the separate library for tree automata, especially with regarded to the further development. On the other hand, one of the goals of the future work should be the deeper investigation of the causes of the slowdown.

Table 7.4: Evaluation of the backward run on SV-COMP benchmark and on the test set of Forester.

Category	Real	Spurious
HeapManipulation	8	3
MemorySafety	6	0
Forester test set	–	7

7.2 Backward Run Evaluation

The implementation of backward run was evaluated on SV-COMP benchmark and also on the test suite distributed with Forester. The evaluation was done by analysing the programs that was correct but Forester found a (spurious) error in them. We also checked whether the backward run correctly confirms the real errors are real.

Note there is a difference in reporting error between Forester test suite and SV-COMP benchmark. All found errors in the Forester test suite, including a memory garbage detection, are reported and the analysis is stopped after the detection. In the SV-COMP benchmark, only the errors breaking a given specification cause stop of the analysis. For instance, the analysis should not stop when a garbage on the heap is detected but the absence of the other errors such as an invalid pointer dereference is verified. We respect the specific rules for the both test sets in our evaluation. It is possible to configure Forester to stop after a garbage detection by the macro `FA_CONTINUE_WITH_GARBAGE` in file *config.h*.

The implementation of backward run confirmed 6 real errors in the programs in the memory safety category of SV-COMP benchmark and 8 real errors in the heap manipulation category. Moreover, the 3 spurious counterexamples were detected in the heap manipulation category. The results are summarized in Table 7.4. The analyzed programs in SV-COMP benchmark include e.g., programs from LDV (Linux Drivers Verification) project containing implementation of alternating singly-linked list or manipulation with mutex locks, or the programs implementing bubble sort over a list implementation from the Linux kernel. Forester does not successfully process all the programs in the test set because it does not currently support all C language constructions. It causes that the number of the found errors, spurious or real, is not as high as it would be if Forester could analyze all the programs in the benchmark.

The results of evaluation at the Forester test set are the following. There are 8 cases where the spuriousness of the found error was proved. A confirmation of the real errors does not make sense because these errors were supposed to be found by Forester also before the backward run implementation as a part of its testing.

The important contribution of backward run is a guarantee of correctness of answer when an error in a program is reported to the user. Unfortunately, so far we can apply the backward run only to the programs that use the supported C language constructions.

7.3 Predicate Abstraction

This section provides an evaluation of the Forester with predicate abstraction. The predicate abstraction is used in the way described in Section 6.5. The first run is performed using height abstraction with height 1 and when a spurious counterexample is found, the analysis creates the predicates and continues with predicate abstraction. This method ensures that Forester successfully passes all the cases that it passed before because the first run remains the same. First, the number of the new cases that Forester can analyse due to predicate abstraction is evaluated. Then the newly analysed cases are studied in a more depth and the performance of Forester is compared to the Predator tool, the winner of HeapManipulation category and the best sound tool in MemorySafety category of SV-COMP'15.

Table 7.5: Evaluation of predicate abstraction in the SV-COMP benchmark and the Forester test suite. Table shows the number of the verified programs which correctness was not verified by Forester before this work and the number of the newly found errors that Forester was not able to detect or confirm before.

Category	New errors found	Newly proved programs
HeapManipulation	9	2
MemorySafety	2	1
Forester test set	0	4

Table 7.5 summarizes the results of the evaluation of predicate abstraction on SV-COMP'15 benchmark and Forester test suite. We distinguish between the programs with a new detected error that Forester was not able to find before and the programs proved to be correct. The most of the new errors are detected in the heap manipulation category of SV-COMP'15 benchmark, where 9 errors have been detected that Forester has not been able to detect before. We also proved 2 programs from the LDV project to be correct. In the memory safety category, there is less found errors and verified programs but it is partially caused by the fact that this category contains less test cases then the heap manipulation category. Finally, we prove correctness of 4 programs from Forester test set which could not be proved without the use of predicate abstraction. These programs manipulate the data structures like doubly-linked list or trees. They do not need creating boxes during the verification procedure although some other use of these data structures can lead to an application of boxes.

The main issue complicating the successful analysis of more programs is the immaturity of Forester. It does not support all C language constructions and it fails due to an internal error because some unexpected state occurs during the analysis. This often causes that many test cases remain not analysed. The symbolic execution able to remove a detected garbage and continue would also improve the results as well. This is problematic mainly because the programs from SV-COMP benchmark create a garbage in the tests cases where a different safety property is verified.

Last, a comparison of the performance between Forester and Predator is given. The programs described here can be analysed only due to the backward run and predicate analysis implementation. Table 7.6 summarizes the results. Forester outperforms Predator in verification of red-black lists because the abstraction in Predator is not able to create the shape invariant. To our best knowledge, it is the only sound tool able to verify these kind of data structures. The other test case is a singly-linked list which has an integer value as the data. The items of the analysed singly-linked list are ordered by the value of their data. So for example, a distance between the head of a list and a node having integer 4 as a data member is smaller than the distance between a node having integer 6 as a data member. In the case of verification of correct program, Predator outperforms Forester because Forester found a few spurious counterexamples before the shape invariant was computed. In the case when the program contains an error, Forester is slightly faster.

Another case when Predator ran out of time is a tree (without the parent pointers) ensuring that each left successor node have allocated it right successor node. Forester is able to verify the correctness of such a structure but it is the most time demanding task of this set. Finally, the last tested program is the one traversing a singly-linked list of even length and then freeing the list by two nodes at time (which is safe because the number of nodes is even). Predator in this case reported a spurious error. Forester detects that the error is spurious but is unable to learn a set of predicates avoiding this error. After the first run, it learns that the list should not be 3 elements long. When the analysis is restarted then a spurious error is detected for the list with 5 elements. The predicates preventing the error are learnt but the analysis will find a new spurious error for the list with 7 elements and so on. Forester is unable to generalize the predicates to exclude all list of odd length from the set of all reachable heap configurations and so the analysis does not terminate.

It is necessary to admit that predicate abstraction is not fully complete in Forester. It needs to be implemented more efficiently to get the better performance on the hard cases. The new predicates

Table 7.6: Evaluation of predicate abstraction on SV-COMP benchmark and the Forester test set. Timeout is set to 1000 second.

Category	Forester [s]	Predator [s]
Red black list, correct	1.10	*timeout*
Red black list, error	0.12	*timeout*
SLL with integers, correct	0.12	0.03
SLL with integers, error	0.02	0.03
Tree, correct	251.03	*timeout*
SLL of even length, correct	*timeout*	*false positive*

are not also created in the most precise way so some states may not be abstracted although it would be theoretically possible. Predicate abstraction also often causes an internal Forester error when a garbage is created but the analysis continues because an absence of another kind of error should be proved.

Chapter 8

Conclusion

The main goals of this thesis were (a) to implement version of Forester tool that uses the VATA library for tree automata representation and manipulation and (b) to extend the verification procedure based on forest automata with backward run for detection of the spurious errors found in the analysed program. The theory of forest automata and the related theory of tree automata have been studied and described in the beginning of the thesis. The verification procedure based on forest automata has been also explored to fulfill the thesis goals.

The connection of Forester and the VATA library was designed and implemented after the analysis of the both tools. Forester had to be refactored for this purpose. Then the adapter design pattern was used to create an interface between Forester and VATA. The version of Forester using the VATA library successfully participated in the competition SV-COMP 2015 [24] and a paper introducing Forester was published in proceedings of the conference TACAS 2015 [14].

The backward run over symbolic states was designed for forest automata based shape analysis and implemented in the Forester tool. The intersection of non-hierarchical forest automata was implemented for the purposes of the backward run. The intersection of forest automata is not only a necessary part of backward run but it is a nice theoretical contribution of this work too. Predicate abstraction over forest automata was also designed including learning the new predicates by the backward run. Predicate abstraction uses bottom-up tree automata intersection in VATA which was also implemented (but not designed) as the part of this work.

The backward run and predicate abstraction were evaluated on SV-COMP benchmark and Forester test suite. These new techniques make it possible to successfully analyze some programs on which the earlier version of Forester fails. Moreover, the extension of Forester with predicate abstraction and backward run is to our best knowledge the only one sound tool able to verify the red-black lists (and similar data structures) fully automatically.

This work participated in the student conference *Excel@FIT 2015* [22]. A paper about this work was accepted to a local proceedings of the conference and the work was awarded with the second prize in the category „Scientific contribution“.

There is still many possibilities for the future work. Forester is not able to analyse many programs because it does not support all C language constructions or the analysis of a program ends in an internal error. Resolving these issues is one of the future goals. Predicate abstraction is not also yet fine tuned and the further improvements would bring the better performance. There is also needed a modification that would allow Forester to continue also when a garbage is created but a different property is verified. The other future directions involve also conceptual and theoretical extensions of backward run, including intersection of forest automata, and predicate analysis such as a generalization to the hierarchical forest automata.

Bibliography

[1] Parosh Aziz Abdulla, Yu-Fang Chen, Lukáš Holík, Richard Mayr, and Tomáš Vojnar. When Simulation Meets Antichains: On Checking Language Inclusion of Nondeterministic Finite (Tree) Automata. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 6015, pages 158–174. Springer-Verlag, 2010.

[2] Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, Peter W. O'Hearn, Thomas Wies, and Hongseok Yang. Shape Analysis for Composite Data Structures. In *Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 178–192. Springer Berlin Heidelberg, 2007.

[3] Ahmed Bouajjani, Peter Habermehl, Adam Rogalewicz, and Tomáš Vojnar. Abstract Regular (Tree) Model Checking. *International Journal on Software Tools for Technology Transfer*, 14(2):167–191, 2012.

[4] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer Berlin Heidelberg, 2000.

[5] Hubert Comon, Max Dauchet, Remi Gilleron, Christof Löding, Florent Jacquemard, Danis Lugiez, Sophie Tison, and Marc Tommasi. Tree automata techniques and applications. Available on: `http://www.grappa.univ-lille3.fr/tata`, 2007. release October, 12th 2007.

[6] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpointsa. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM.

[7] Edsger W. Dijkstra. Structured Programming. In *Software Engineering Techniques.* NATO Science Committee, 1970.

[8] Kamil Dudka, Petr Peringer, and Tomáš Vojnar. An Easy to Use Infrastructure for Building Static Analysis Tools. In *Proceedings of the 13th International Conference on Computer Aided Systems Theory*, pages 328–329. The Universidad de Las Palmas de Gran Canaria, 2011.

[9] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[10] Peter Habermehl, Lukáš Holík, Adam Rogalewicz, Jiří Šimáček, and Tomáš Vojnar. Forest Automata for Verification of Heap Manipulation. In *Computer Aided Verification*, volume 6806 of *Lecture Notes in Computer Science*, pages 424–440. Springer Berlin Heidelberg, 2011.

[11] Peter Habermehl, Lukáš Holík, Adam Rogalewicz, Jiří Šimáček, and Tomáš Vojnar. Forest Automata for Verification of Heap Manipulation. Technical Report FIT-TR-2011-001, FIT BUT, 2011.

[12] Peter Habermehl, Lukáš Holík, Adam Rogalewicz, Jiří Šimáček, and Tomáš Vojnar. Forest Automata for Verification of Heap Manipulation. *Formal Methods in System Design*, 41(1):83–106, 2012.

[13] Lukáš Holík, Ondřej Lengál, Adam Rogalewicz, Jiří Šimáček, and Tomáš Vojnar. Fully Automated Shape Analysis Based on Forest Automata. Technical Report FIT-TR-2013-01, FIT BUT, 2013.

[14] Lukáš Holík, Martin Hruška, Ondřej Lengál, Adam Rogalewicz, Jiří Šimáček, and Tomáš Vojnar. Forester: Shape analysis using tree automata. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 9035 of *Lecture Notes in Computer Science*, pages 432–435. Springer Berlin Heidelberg, 2015.

[15] Lukáš Holík, Ondřej Lengál, Adam Rogalewicz, Jiří Šimáček, and Tomáš Vojnar. Fully Automated Shape Analysis Based on Forest Automata. In *Computer Aided Verification*, volume 8044 of *Lecture Notes in Computer Science*, pages 740–755. Springer Berlin Heidelberg, 2013.

[16] Martin Hruška. Efficient Algorithms For Finite Automata, 2013. FIT BUT.

[17] Donald E. Knuth. *The Art of Computer Programming, Vol.1.* Addison-Wesley, 3rd edition, 1997. ISBN 0-201-89683-4.

[18] Quang Loc Le, Cristian Gherghina, Shengchao Qin, and Wei-Ngan Chin. Shape analysis via second-order bi-abduction. In *Computer Aided Verification*, volume 8559 of *Lecture Notes in Computer Science*, pages 52–68. Springer International Publishing, 2014.

[19] Ondřej Lengál. Efficient Library For Tree Automata, 2010. FIT BUT.

[20] Ondřej Lengál, Jiří Šimáček, and Tomáš Vojnar. VATA: A Library for Efficient Manipulation of Non-deterministic Tree Automata. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 7214, pages 79–94, 2012.

[21] Karl J. Lieberherr and Ian M. Holland. Assuring Good Style for Object-Oriented Programs. *IEEE Softw.*, 6(5):38–48, 9 1989.

[22] Web pages of Excel@FIT conference. Excel@FIT. `http://excel.fit.vutbr.cz/`, 2015 [cit. 2015-05-14].

[23] Web pages of Forester. Forester. `http://www.fit.vutbr.cz/research/groups/verifit/tools/forester/`, 2012 [cit. 2014-12-31].

[24] Web pages of SV-COMP 2015. SV-COMP 2015. `http://sv-comp.sosy-lab.org/2015/`, 2014 [cit. 2014-12-31].

[25] Web pages of the VATA library. The VATA library. `http://www.fit.vutbr.cz/research/groups/verifit/tools/libvata/`, 2012 [cit. 2014-12-31].

[26] Web pages of Timbuk. Timbuk. `http://www.irisa.fr/celtique/genet/timbuk/`, 2012 [cit. 2014-12-31].

[27] John C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, LICS '02, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society.

[28] Alexander Stepanov and Paul McJones. *Elements of Programming.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2009.

[29] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, 4th edition, 2013.

[30] Tomáš Vojnar. *Lecture Notes from Formal Analysis and Verification Course.* Brno, CZ.

[31] Wikipedia. Adapter pattern — Wikipedia, The Free Encyclopedia, 2014 [cit. 2014-12-31].

Appendix A

Storage Medium

The storage medium contains the source code of the Forester tool (that is distributed together with the Predator tool and Code Listener) and the source code of the VATA library. The tests used for the evaluation in this thesis are another content of the storage medium. It also contains an electronic version of this technical report and its LaTeX sources. The file *README* describes the structure of the storage medium and the instructions for compiling and running Forester.