BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FROM AUTOMATA TO TESTING AND FAR BEYOND

OD AUTOMATŮ K TESTOVANÍ A JEŠTĚ DÁL

PHD THESIS
DISERTAČNÍ PRÁCE

AUTHOR **MARTIN HRUŠKA**
AUTOR PRÁCE

SUPERVISOR **prof. Ing. TOMÁŠ VOJNAR, Ph.D.**
ŠKOLITEL

COSUPERVISOR **doc. Mgr. LUKÁŠ HOLÍK, Ph.D.**
ŠKOLITEL

BRNO 2022

Abstract

This thesis focuses on applications of automata theory to software quality. In the first part, we focus on shape analysis which can be used for formal verification of programs manipulating dynamic data structures. Particularly, we develop an approach of backward program execution along possible counterexamples traces and counterexample-guided refinement for shape analysis based on forest automata. We also introduce a new approach based on automata over graphs with a bounded tree width which is more general than forest automata but still has feasible computation properties.

In the second part, we introduce a method for automated testing of manufacturing execution systems (MES) in digital twin. We are able to orchestrate a digital twin to reproduce behaviour of a real-world setting in which MES is deployed and so provide a safe environment for testing. Moreover, we can generate new test cases by applying automata and abstraction over them in this context.

Abstrakt

Tato práce se zabývá aplikacemi teorie automatů v zajištění kvality software. V první části se zabývá aplikací automatů v tzv. analýze tvaru, kterou lze využít pro formální verifikaci programů pracujících s dynamickými datovými strukturami. Konkrétně představuje rozšíření analýzy tvaru založené na lesních automatech o zpětný běh analýzy přes řádky programu, které se objeví v potenciálním protipříkladu a zjemnění abstrakce založené protipříkladech. Dále je v práci představena nová doména pro analýzu tvaru a to automaty nad grafy s omezenou stromovou šířkou. Ty jsou obecnější než lesní automaty, ale zároveň výpočetní složitost algoritmů s nimi pracujících je použitelná.

V druhé části se zabýváme automatizovaným testováním výrobních informačních systémů v prostředí digitálního dvojčete. Představujeme metodu, která dokáže orchestrovat digitální dvojče tak, aby reprodukovalo reálné prostředí, v němž bývají zmíněné systémy nasazeny. To poskytuje bezpečné prostředí testování výrobních informačních systémů. Navíc jsme metodu rozšířili o možnost tvorby nových testovacích scénářů nad rámec pouhé reprodukce již pozorovaného chování reálného prostředí, a tak zvýšili kvalitu testovacího procesu. Tato metoda je založena na použití teorie automatů, konkrétně na použití abstrakce nad konečnými automaty.

Keywords

static analysis, formal verification, shape analysis, testing, automata

Klíčová slova

statická analýza, formální verifikace, analýza tvaru, testování, automaty

Reference

HRUŠKA, Martin. *From Automata to Testing and Far Beyond.* Brno, 2022. PhD thesis. Brno University of Technology, Faculty of Information Technology. Supervisor prof. Ing. Tomáš Vojnar, Ph.D.

From Automata to Testing and Far Beyond

Declaration

I hereby declare that this PhD thesis was prepared as an original work by the author under the supervision of prof. Ing. Tomáš Vojnar, Ph.D., and doc. Mgr. Lukáš Holík, Ph.D. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Martin Hruška
May 7, 2023

Acknowledgements

V této sekci je možno uvést poděkování vedoucímu práce a těm, kteří poskytli odbornou pomoc (externí zadavatel, konzultant apod.).

Contents

Chapter 1

Introduction

Assuring software quality is a crucial part of development cycles for all kinds of software. It is important for low-level code often deployed in highly critical applications where one bug can have expensive consequences or even threat human lives. It is crucial also in high-level software where bugs can destroy user experience and ruin the whole process of development and selling a new software.

Approaches to Software Quality There are different approaches to software quality at different stages of development cycle with different quality warranties. The most rigorous one is based on a formal specification of requirements followed by deriving program from the specification or proving program against the specification manually or with help of automated proof assistant. This approach is present from the beginning of computer science and although it provides the highest level of guarantee it has not became mainstream. The main negatives are the demands on qualification of developers and severe difficulties to scale to larger software.

There are also more or even fully automated approaches to proving correctness of programs such as model checking which strive to verify that a program satisfies a given specification. These methods are generally called formal verification. They are easier to use than manual proving of program, however, they still have problems with scaling and applicability to real-world software. There are more specialized automated methods such as different types of sound static analysis based, e.g., on data flow analysis or abstract interpretation, which are designed to verify certain program properties (such as values of integer variables or analyses of termination properties). These methods scale better and can work out of the box. However, they are less precise and often tuned for a particular program property.

Another approach comes from programming language community claiming that programming language itself should guide a programmer to write software without bugs. This is mostly achieved by static strong type systems which facilitate encoding different program properties to the type system. The advantage of this approach is that one does not need another technology and forces programmers to write correct code. On the other hand, the programming languages of this kind have a slow learning curve and can be even unusable for certain programmers. Moreover, only a limited set of program properties is encodable to type systems.

The third approach to software quality are lightweight automated methods such as automated testing, unsound static analysis based, e.g., on error pattern matching, symbolic execution, or extrapolating dynamic analysis. This class of methods also contains the relaxed versions of methods from formal verifiction such as bounded model checking or

different approaches to abstract interpretation which do not overapproximate the original semantics. The methods try to hit a sweet spot between mathematical rigour and plain bug hunting. They are easy to use, straightforward, often scale well but may detect many false alarms which can cause that they are ignored by developers completely.

The last of the main approach is based on traditional testing by humans possible aided by automated test execution, continuous integration, and the like. It may be developers writing unit tests or the whole tester teams implementing the complex integration tests. It is mostly known and time-tested approach, but it costs a lot of human resources and does not give any guarantees about correctness.

The presented dichotomy does not pretend to be complete or exhaustive. It tries to summarise the main approaches and give an overview of them. In practice, the different approaches are combined, complement each other, and may fall to different categories at the same time. The proposed thesis focuses on automated formal methods (the first part of thesis) and lightweight automated testing methods (the second part), which we used in a domain where scalability was required.

Finite Automata The main uniting theme of the thesis is application of finite automata to the fields of formal verification and automated testing. Finite automata are a (syntactically) simple but powerful tool for representing regular languages. Since their introduction in the 50s by Rabin and Scott [101], they have been used in various domains such as compilers, modelling of various kinds of systems, design of digital circuits, natural language processing, speech recognition, as well as in verification or testing of software and hardware. Moreover, finite automata and their variations are widely studied in theoretical computer science either from perspective of the decidable properties or complexity of algorithms for their manipulation.

Finite automata are mainly used to represent the various structures with reccurent patterns of substructures (we deliberately do not use word regular to prevent confusion with meaning of the word in context of theory of formal languages). A classical application is language consisting of plain strings (used in pattern-matching applications such as lexical analysis in compilers or detection of malicious strings in security domains). A more sophisticated case arises when using regular language s containing sequences of messages or generally objects with an inner structure, which is used, e.g., for modelling communication protocols. In formal verification, there are also automata representing regular languages of infinite strings (so called Buchi automata), which are used for verification of some properties requiring one to reason about infinite behaviours such as termination of a program. Another kind of automata used in formal verification are tree automata that represent sets of trees. These automata have been applied, e.g., to represent tree data structures (and even more complex structures defined over their tree backbones) allocated on heap by a program.

In the applications of automata in this thesis, we need to perform operations such as testing language inclusion and equivalence of two finite automata, testing emptiness of language of automaton or complementation of automaton. All these properties are decidable but some of them have a high complexity, e.g., complement has exponential complexity w.r.t. number of states of automaton, language inclusion and equivalence are PSACE-complete (or even EXPTIME-c for nondeterministic tree automata). Fortunately, efficient heuristics for the mentioned operations were introduced making them computationally feasible in many practical cases TODO CITE.

As we mentioned above, we focus on applications of automata to two different fields. The first one is shape analysis, i.e., static analysis of programs manipulating dynamic data structures. In shape analysis, we are interested in representing data structures allocated on the heap, such as linked lists or trees, and verifying the operations properties such as absence of invalid memory dereference, invalid memory free, or memory leaks. Programs using dynamic data structures are a great fit for methods of formal verification since they are prone to the mentioned bugs and are written in programming languages with manual memory management as C or C++ where memory management is fully within user's control. At the same time, these programs are used in critical software such as kernels of operating systems. Any bugs in them may affect many users and lead to expensive losses. Therefore they need strong safety guarantees which can be provided by the methods with formal basis.

Our approach to shape analysis is based on forest automata and automata over graphs (both extensions of tree automata) with a bounded tree width. Both formalisms are able to represent a quite wide class of data structures that can be allocated on the heap. We employ a verification procedure to compute shape invariants for each program location. In our case, a shape invariant is an automaton representing all possible states of heap. Finally, we check that shape invariants imply that a given specification is fulfilled by the program (usually checking absence of invalid dereferences, invalid frees or memory leaks).

The second field of application of automata in this thesis is automated software testing. Particularly, we focus on testing of a manufacturing execution system (MES) in an environment of digital twins. MES is critical software in controlling a factory where bugs may stop the production or even damage manufacturing machines. Such bugs may be very expensive, and it is important to find as many of them as possible before MES is deployed to a real factory. Our approach analyses the behavior of the software used in a real factory, builds its models and then generates tests for a digital twin of the factory which are applied, e.g., when a new version of the software is deployed and the rest of factory emulated.

1.1 Goals of the Thesis

The first goal of this thesis is to improve the current tools and methods of shape analysis. We aim to further develop the approach to shape analysis based on forest automata proposed in [54, 66] and enhance it by 1. revisiting and improving its description, 2. enhancing it by a mechanism of checking possible counterexamples and automated abstraction refinement, 3. improving its tool support, particulary tool Forester, to make it able to compete on an international level. We also attempt to find an alternative automata-based representation suitable for shape analysis with the aim of cover the class of graphs with bounded tree width.

Futher we want to study possible alternatives to shape analysis based on forest automata. We will focuse mainly on the following topics: 1. an alternative automata-based representation with the aim of cover the class of graphs with bounded tree width (taking inspiration from Courcelle theorem [41]), 2. exploring possibilities of shape analysis based on predicate logic, templates-based invariants, abstract domains, and SMT solving.

The second goal of the thesis is to apply automata to the field of testing, particularly to testing manufacturing execution systems. We want to create a novel approach combining automated test generation from observed behaviour of a manufactory with testing in an environment of a digital twin which will be orchestrated according to the tests we create.

1.2 Overview of the Achieved Results

Shape analysis The first of our results is an extension of shape analysis on forest automata by spurious counterexample detection and predicate abstraction refinement published in [65]. Abstraction is generally used in formal verification to overapproximate represented state space of the program under verification, to reduce the size of representation of finite state space and to allow infinite state spaces to be represented finitely. On the other hand, a violation of a specification can be detected in the overapproximated part of the state space, which needs no to correspond to a real behaviour in a program under verification. We call such fake violations of specification spurious counterexamples. Forest-automata-based shape analysis as proposed in [54, 66] used an abstraction overapproximating the set of reachable states of the heap, i.e., abstract forest automata represent more possible shapes of the heap than those allocated by the program in reality. However, the method did not contain any algorithm to check whether a found possible counterexample to the given specification is real or spurious. We develop a detection of spurious counterexamples based on so called backward runs. It will be described in more a technical way in the rest of the thesis, but, intuitively, it reverts all actions done over forest automata in the forward symbolic execution, starting from the forest automaton representing an erroneous configuration. When forest automata in the backward run diverge from the ones in the forward run, it means that the found possible counterexample is spurious.

Once we are able to detect spurious counterexamples, we need an abstraction which is refinable. In other words, we want to make the abstraction more precise once a spurious counterexample is found so that when we restart the analysis, we will not reach the same counterexample again. Therefore we developed a version of predicate abstraction for forest automata. We are able to derive new predicates from a spurious counterexample which guarantee that after a restart of the verification method, we exclude the spurious counterexample from the represented state space.

We also revisited the formalisation and description of shape analysis based on forest automata. As a part of our work we introduced refined presentation of the framework in more comprehensible way. A slow learning curve and hard to understand formalism were one limiting factor of development of shape analysis based on forest automata. The refined description is presented in this thesis and is accepted for publication in book about state-of-the-art methods in software verification.

We have also significantly improved implementation of the approach in the Forester tool. Consequently, Forester was able to seriously compete in international level. Particulary, it participated in software verification competition SV-COMP [6] in the editions of years 2015, 2016, 2017, and 2018. Although Forester did not win a medal in any major category, we managed to support a non trivial subset of the C language needed for a reasonable participation in the competition and made the tool mature enough to be competitive with other tools, often developed by large teams. The tool also verified soundly some test cases such as programs manipulating skip-lists of level 3 or variants of trees which were not analysable by any other tool.[1] Although Forester stayed at the state of a prototype and a lot of engineering work would be needed to make it a mature tool, we were able to show

[1] to the best of our knowledge, even today there is no tool in SV-COMP that would be capable of handling these programs in sound way. Some of the programs cound be handled by the S2 tool[78] whose abstraction is, however, rather fragile, allowing it to handle complex programs on one hand, but failing on simple ones on the other hand according to our experiments with the tool. See also related work section.

that a shape analysis based on automata can compete with other approaches, and it is a meaningful path of research.

Beyond the work done on forest-automata-based shape analysis, we introduced automata over graphs with a bounded tree width. The automata are directly influenced by the Courcelle's theorem [42] stating that any graph property definable in the monadic second-ordered logic over graphs can be decided in a linear time on graphs with a bounded tree width. We sketch automata using principles from this theorem together with an algorithm for entailment over these automata which has a singly-exponential complexity. Automata over graphs with a bounded tree width are able to represent a more general class of graphs than forest automata but still have feasible computational properties, making them a suitable domain for shape analysis.

Finally, our last result in shape analysis was participation on approach to shape analysis based on predicate logic, templates-based invariants, abstract domains, and SMT solving implemented in the 2LS framework [105]. In this case, the possible shapes of heap are described by a points-to relation between pointer variables and abstract memory objects. The program is converted to a SSA form over which a system of constraints modelling operations on the heap is put together using especially proposed templates of points-to relation. Then the invariants of the points-to relation are computed by an application of SMT solver to the system of constraints. This approach is more straightforward, more scalable but less general than the approaches based on automata.

Automated testing Our main result in automated testing is a system for orchestrating a digital twin to test a manufacturing execution system (MES). Our approach is based on learning a model of communication in a real manufactory between a MES, machines, and an enterprise resource planning (ERP) system. We also learn models of messages communicated in a monitored system. Once we learn a model of communication and models of sent messages, we are able to generate a testing scenario for a digital twin. A digital twin contains emulated parts of the factory such as machines, an ERP system, human workers, and a natively run MES. Such setup can be used, e.g., for testing a new version of the MES, when we derive the models from logs of communication in a real manufactory where an older version of the MES was deployed. Then the new version is run in the digital twin, and it is checked that the new version has the same semantics as the older version where it is expected.

Moreover, since we use finite automata to the model communication, we are able to make an abstraction over the learnt models and extrapolate the new test cases, which can be used for a more robust testing of the MES in the digital twin than just a reproduction of the previously seen runs.

The methods described above were implemented in the tool Tyrant [3]. The tool was developed for testing a particular MES used in industry, and was tested on communication logs from a real manufactory.

1.3 Plan of the Thesis

The plan of the rest of the thesis is the following. In Chapter 2 the state of the art in shape analysis is described. In Chapter 3, we provide a description of forest-automata-based shape analysis including a detailed description of the backward run and predicate abstraction, which were originally developed as a part of the thesis. Chapter 4 summarizes Forester

participation in the various editions of the software verification competition SV-COMP. Chapter 5 introduces automata over graphs with a bounded tree width together with their crucial properties. Chapter 6 discuss an approach to shape analysis based on predicate logic, templates-based invariants, abstract domains, and SMT solving implemented in 2LS framework [105]

Finally, Chapter 7 describes our methods for testing MES in the environment of digital twins.

Part I

Automata in Shape Analysis

Chapter 2

State of the Art

This section provides a summary of the state-of-the-art methods for shape analysis. First, we give an overview of the formalisms for representing data structures allocated on the heap. Then the refinement techniques for abstraction over various formalisms are summarized, and finally, existing works on automata for graph languages are presented.

2.1 Shape Analysis

As we mentioned, *shape analysis* deals with analysis of dynamically linked data structures allocated on the heap. Many different approaches to shape analysis have been proposed, using various underlying formalisms, such as logics [72, 92, 104, 103, 17, 53, 93, 111, 110, 35, 84, 18, 46, 79, 53, 85, 37, 30, 99], automata [28, 45, 57, 66, 29], graphs with summary nodes and edges [37, 48], graph grammars [59, 109], or upward closed sets [9].

In the following chapter, we abstract data structures allocated on the heap to graphs (so called *heap graphs* or *heap shapes*). An allocated memory heap cell corresponds to a node of the graph and a pointer pointing from one memory cell to another is represented by an edge between the nodes. The selectors are represented by the labels of the edges. Formally, a heap graph G is a tuple (V, E, α) where V is a set of nodes, $E \subseteq V \times V$ is a set of edges, $\alpha : E \to \mathit{Sel}$ is a labelling function, and Sel is a set of selector names.

Shape analysis heavily relies on the formal model used to represent the set of the heap shapes reachable in the program. The requirements on the model include: (a) genericity and expressivity — the more general classes of the heap grphs model can represent, more programs can be (in theory) analyzed, (b) automation — the model should be derivable from a program automatically and have decidable properties important for reasoning about the program, (c) efficiency and scalability — there should exist efficient algorithms for manipulation with the model even when it is applied to big systems.

2.1.1 Three-valued Predicate Logic with Transitive Closure

The approach of [104] uses predicates to express relations between nodes of heap graphs. Moreover, it introduces the third logical value (*unknown*) to the standard boolean values *true, false*. The *unknown* value is used when more items from a given universe may but need not to be in a relation. The *unknown* value may be needed for an abstraction merging some nodes. E.g., consider an abstraction $\mathcal{A}_\alpha : V \to V$ where V is a set of nodes of heap graphs, and the predicate NEXT : $V \times V \to \{\mathit{true}, \mathit{false}, \mathit{unknown}\}$ saying that we can go from the heap node u to the node v using the NEXT selector, formally $(u, v) \in$

$E \wedge \alpha(u,v) = NEXT$. Then when $\mathcal{A}_\alpha(u) = s$ and $\mathcal{A}_\alpha(v) = s$, where s is a so-called summary node, and there exists $w \in V$ such that $\text{NEXT}(u,w) = true \wedge \text{NEXT}(v,w) = false$, then $\text{NEXT}(s,w) = unknown$.

The work of [104] that build, the TVLA shape analyser on top of 3-valued predicate logic with transitive closure was one of the first works on shape analysis. The framework is general, it can find some basic predicates describing shapes in the analysed program automatically, but it needs to be parametrized manually by predicates to represent more complex data structures. The precision of the used abstraction by refinement was addressed in [91, 23]. Scalability of the method has not been systematically studied.

2.1.2 Separation Logic

Separation logic was first introduced by Reynolds [103] as an extension of *Hoare logic* for reasoning about programs manipulating dynamic data structures. Hoare logic builds on so-called *Hoare triples*. A triple has a form $\{P\}C\{Q\}$ where P, Q are assertions in predicate logic and C is a command in an imperative programming language. When P is satisfied before an execution of C, then the Q assertion should hold when C terminates. Another possible formulation is that predicates from the set P are transformed to the set Q with respect to the semantics of C.

While the assertions in Hoare logic consist of standard connectives and symbols of predicate logic, separation logic extends the framework with several new ones for heap description: *emp* (representing the empty heap), $P * Q$ (saying that the heap can be split to two separated parts, i.e., parts not allocating same nodes, but allocating disjoint sets of nodes, where one satisfies P and the second Q), $e \mapsto e'$ (the address defined by the expression e is mapped to the value defined by the expression e', and the rest of the heap is empty, i.e., $*emp$ is implicit), and $P \mathrel{-\!\!*} Q$ (which describes h such that the union of h with heap h' that is disjoint from h and that satisfies P will satisfy Q).

The interest in separation logic has started rising with the introduction of the automated, abstract-interpretation-based approaches associated with Space Invader [110] and SLAyer [18]. These approaches assume pre-defined templates of shape predicates and are not very general since they are restricited to programs over certain classes of linked lists (and cannot handle even structures such as linked lists with data pointers pointing either inside the list nodes or optionally outside of them, which we can easily handle, as discussed later on). E.g., the separation logic-based approach of [53] suffer from the need to manually provide inductive predicates describing the heaps shapes (even for some list structures). The work [79] concerning overlaid data structures mentions an extension of Space Invader to trees, but this extension is also of a limited generality and requires some manual help.

Separation logic has also been extended to a concurrent version, see, e.g., [34].

The scalability of the approach based on separation logic was significantly improved by bi-abduction [35]. Further, the second-order bi-abduction provided ability to learn some predicates automatically and so made possible analysis of more complex data structures such as skiplists [78]. The principles similar to second-ordered bi-abduction enabling analysis of complex data structures were used in [53].

The recent work [63] further extended separation logic by the low-level features of data structures manipulation such as pointer arithmetic, bit-masking on pointers, block operations with blocks of variable size, their splitting to fields of in-advance-not-fixed size, merging such fields back, and reinterpreting them differently, etc..

A main downside of the bi-abduction-based methods is their inability to provide a counterexample breaking the given specification. Moreover, these approaches are rather fragile. Especially, the [53] is quite dependent on the fact that the encountered data structures are built in a "nice" way conforming to the structure of the predicate to be learnt (meaning, e.g., that lists are built by adding elements at the end only), which is close to providing an inductive definition of the data structure. Similarly, the approach of [78] fails on examples that might seem easy (such as certain rather straightforward variants of creations and deletions of a doubly-linked list).

Separation logic has found its way to practice, mainly its implementation in the Facebook Infer tool widely used in Facebook [108] and other companies.

Separation logic was further modified to *incorrectness separation logic* [100] which combines local reasoning of separation logic with the newly introduced *incorrectness logic* [100]. An interprocedural analysis based on incorrectness separation logic has no false positives by construction and is aimed to rigorous bug hunting. Incorrectness logic enables reasoning about errors in programs based on an underapproximating analogue of Hoare triples. The authors also newly introduce the operator $x \not\mapsto$ saying that the pointer x has been deallocated (and not reallocated). The operator was needed to distinguish between a dangling pointer and pointer which we do not know anything about and it also makes possible to introduce a frame rule for underapproximating reasoning in separation logic. The methods were implemented in the plugin called Pulse inside Facebook Infer.

We note that there are other works on separation logic, e.g. [93], that consider tree manipulation, but these are usually semi-automated only. The work [85] proposes an approach that uses separation logic for generating numerical abstractions of heap manipulating programs allowing for checking both their safety as well as termination. The described experiments include even verification of programs with 2 level skip-lists. The work, however, still expects the user to manually provide an inductive definition of skip lists in advance. Likewise, the work [37] based on the so-called separating shape graphs reports on verification of programs with 2 level skip-lists, but it also requires the user to come up with summary edges to be used for summarizing skip list segments, hence basically with an inductive definition of skip lists.

For seperation logic to be applied in verification procedure, one needs to solve problems of satisfiability and entailment on separation logic. Many abstract interpretation tools solve the problem rather in ad hoc ways. On the other hand, there have also appeared many results on *automated decision procedures* for various fragments of separation logic addressing these problems [52, 70, 75, 77, 51]. Unfortunately, the cited works cannot be used to the bi-abduction problem described in the above section, which is crucial for a compositional (scalable) program analysis. In bi-abduction, one needs to solve the abduction problem $P * [?] \models Q$, where $*$ is the separating conjunction. The best solution (i.e., the logically weakest) is given by the formula $P \mathrel{-\!\!*} Q$ where one needs to use the magic wand operator $\mathrel{-\!\!*}$. The cited works do not support magic wand since it has been shown that adding even the singly-linked list-segment predicate to a propositional separation logic that includes magic wand causes undecidability of the satisfiability problem [44], [15]. The mentioned problem was recently addressed by [96] which defined new semantics for separation logic. The new semantics make possible to use magic wand and the singly-linked list-segment predicate. The work also discuss their potenial application to the abduction problem. However, to make this a reality, more research is needed.

2.1.3 Symbolic Memory Graphs

Symbolic Memory Graphs (SMG) [48] were designed as an abstract domain for the framework of abstract interpretation [43]. They can model states of the heap precisely, but one can also perform *widening* over SMGs by introducing summary nodes in the form of singly/doubly-linked lists nodes to accelerate the analysis and enable representation of infinite state spaces. Since the representation is quite straightforward, it is easy to model the semantics of the concrete program operations in the abstract domain. These practical advantages are confirmed by the repeated wins of the tool in the competition on software verification SV-COMP [6].

The formalism was designed mainly for list structures aiming at verification of system-level programs (e.g. Linux kernel) using such structures. Therefore it can also handle low-level memory operations with byte precision [48]. However, the domain has not been yet generalized to the tree structures. The scalability of the method has not been systematically studied yet, but the principles of bi-abduction should be applicable to the domain just as for the separation logic-based methods.

Many of the principles used in [48] to deal with low-level memory operations have been applied in the recent work [63] combining them with sepration logic. The scalability of the approach is, however, still limited despite the potential stemming from the modularity of the approach.

2.1.4 Abstract Regular Model Checking

Abstract regular model checking (ARMC) [32] is an automata-based method for verification of parametrized and infinite-state systems. It employs automata over words and trees to represent reachable configurations of the system being verified and a regular transition relation (represented by transducers or speciliazed operations over automata) to model the semantics of the analysed system. ARMC was applied on various kinds of parametric and infinite systems such push-down systems, systems with queues, parametrised networks of process, petri nets. It was also used for verification of programs with dynamic data structures where words and trees are used to represent reachable shapes of heap[31].

Abstraction over automata is used to overapproximate the set of reachable configurations, e.g., by collapsing automata states whose languages are same up to some bound or satisfy, i.e., intersect, the same set of predicate languages. Counterexample-guided refinement is applied to refine the abstraction and to validate the found potential counterexamples when needed.

When applied in forward manner, the verification procedure starts with an automaton representing initial configurations of an analysed system. An abstraction is applied to the automaton followed by an application of a transition in each step of the verification procedure. An intersection of the automaton representing so-far computed reachable configurations with an automaton representing the bad configurations of the system is also done in each step. When the intersection is empty, the verification procedure continues until a fixpoint is reached (the set of system configurations represented by an automaton is not enlarged after the application of a transition relation). When the intersection is not empty, a backward run is started to validate the counterexample. If the counterexample is not real, then the abstraction is refined to avoid reaching the counterexample again, and the verification procedure is restarted. Otherwise the real counterexample is reported to a user.

As inidicated above, abstract regular model checking is a general verification method that was applied to various kinds of systems including pointer programs [28, 54, 66] where automata are used as a domain for representation of the allocated data structures on the heap. In [28], the whole heap is encoded in one tree automaton, and the semantics of programs is represented by a tree transducer. The approach is able to verify complex tree structures but since it encodes the whole heap in one automaton, even a small change in one part of the heap may be propagated over the whole model, which negatively effects the scalability of the method. The scalability problem was improved in [54] where the heap is represented by a tuple of tree automata, so called forest automata, which localize a change in one part of heap to a change in a particular tree automaton. The approach needed forest automata representing the repeating sub-graphs of the heap to be provided manually. This was solved by [66] where the shape analysis based on forest automata became fully automated and was able to verify structures such as skiplists.

The work [11] added ordering relations into forest automata to allow verification of programs whose safety depends on relations among data values from an unbounded domain. The FORESTER tool implementing the method is able to verify complex data structures such as various tree and skiplists of the second and third level [66] which are not analysable by any other tool fully automatically. This work is starting point of some results of this thesis which extended [66] by potential counterexamples analysis (to distinguish spurious and real ones). and abstraction refinement.

2.1.5 Symbolic Execution

TODO:

2.1.6 Bounded Model Checking

So far, we introduced domains for representation of shapes allocated on the heap and assumed that the verification procedure used will be sound. However, when the requirement on soundness is relaxed, we will obtain a bug hunting method still able to detect many errors in program. One of such methods is *bounded model checking (BMC)* whose applications achieved many achievements in the various editions of the SV-COMP competition.

An exploration of the state space is limited up to some bound in BMC. The bound can be given, e.g., on the number of interleavings in a parallel program, on the number of the loop unfoldings, or on the size of the memory.

BMC systematically explores the state space of the analysed systems up to the given bound and checks whether the given specification holds in each possible state. From the perspective of shape analysis it means that the methods explores which shapes may be possibly allocated on heap in the given state of the program and checks whether it does not lead to a memory manipulation related error.

A model of the analysed system can be automatically derived from an analysed program. Therefore the approach may be fully automated. It can be also easily combined with other domains of interest in program analysis, e.g., interval analysis of numerical values. It is also general and scales to large software systems. On the other hand, because the approach is unsound it can only find bugs, but it cannot prove a program to be correct. There are attempts to get over this bottleneck by *k-induction* but they are still not as mature as the original bounded approach.

2.2 Counterexample Validation and Automatic Refinement of Abstraction for Shape Analysis

A common weakness of the current approaches to shape analysis is a lack of a proper support for checking spuriousness of potenial counterexample traces, possibly followed by automated refinement of the employed abstraction. This is one of the problems we tackle in this thesis. Below, we characterize previous attempts on the problem and preparing grounds for its application to forest-automata-based shape analysis described in the Section **??**.

A well-known approach to refinement of abstraction is the *counterexample-guided refinement (CEGAR)* principle [38] which works as follows:

1. Set a default (overapproximating) abstraction.
2. Perform an analysis of the given program.
3. If a counterexample is found:
   (a) start a validation of the counterexample,
   (b) if it is valid, report an error,
   (c) otherwise refine the abstraction and go to Point 2.
4. If no potential counterexample is found, report the program as correct.

CEGAR has been quite well elaborated in the context of software verification. In the following text, we discuss the application of CEGAR in the different approaches to shape analysis.

As we briefly mentioned in Section 2.1.1, the work [23] adds a CEGAR loop on top of the TVLA analyzer [104], which is based on *3-valued predicate logic with transitive closure*. The refinement is, however, restricted to adding more pointer variables and/or data fields of allocated memory cells to be tracked only (together with combining the analysis with classic predicate analysis on data values). The analysis assumes the other necessary heap predicates (i.e., the so-called core and instrumentation relations in terms of [104]) to be fixed in advance and not refined. The work [82] also builds on TVLA but goes further by learning more complex instrumentation relations using inductive logic programming. The core relations are still fixed in advance though. Moreover, the approach of [82] is not CEGAR-based—it refines the abstraction whenever it hits a possible counterexample in which some loss of precision happened, regardless of whether the counterexample is real or not.

In [98], a CEGAR-based approach was proposed for automated refinement of the so-called *Boolean heap abstraction* using disjunctions of universally quantified Boolean combinations of first-order predicates with free variables and transitive closure. The approach assumes the analyzed programs to be annotated by procedure contracts and representation invariants of data structures. New predicates are inferred using finite-trace weakest preconditions on the annotations, and hence new predicates with reachability constraints can only be inferred via additional heuristic widening on the inferred predicates. Moreover, the approach is not appropriate for handling nested data structures, such as lists of lists, requiring nested reachability predicates.

In the context of approaches based on *separation logic*, several attempts to provide counterexample validation and automated abstraction refinement have appeared. In [16], the SLAyer analyzer was extended by a method to check spuriousness of counterexample

traces via bounded model checking and SMT. The approach may, however, fail in recognizing that a given trace represents a real counterexample. Moreover, the associated refinement can only add more predicates to be tracked from a pre-defined set of such predicates.

In [13], another counterexample analysis for the context of separation logic was proposed within a computation loop based on the Impact algorithm [73]. The approach uses bounded backwards abduction to derive so-called spatial interpolants and to distinguish between real and spurious counterexample traces. It allows for refinement of the predicates used but only by extending them by data-related properties. The basic predicates describing heap shapes are provided in advance and fixed.

Another work based on backwards abduction is [26]. The work assumes working with a parametrized family of predicates, and the refinement is based on refining the parameter. Three concrete families of this kind are provided, namely, singly-linked lists in which one can remember bigger and bigger multisets of chosen data values, remember nodes with certain addresses, or track ordering properties. The basic heap predicates are again fixed. The approach does not guarantee recognition of spurious and real counterexamples nor progress of the refinement.

None of the so-far presented works is based on automata, and all of the works require some fixed set of shape predicates to be provided in advance. Among *automata-based approaches*, counterexample analysis and refinement was used in [28] (and also in some related, less general approaches like [27]). In that case, however, a single tree automaton was used to encode sets of memory configurations, which allowed standard abstraction refinement from abstract regular (tree) model checking [32] to be used. On the other hand, due to using a single automaton, the approach did not scale well and had problems with some heap transformations.

The first approach based on forest automata used a fixed abstraction [57]. However, in [57, 66], it was conjectured that counterexample validation and abstraction refinement should be possible in the context of forest automata too. This thesis will show that this is indeed the case, but that much more involved methods than those of [32] are needed.

2.3 Work on Graph Automata

The generality of a verification method for shape analysis crucially depends on the choice of the underlying formalism enabling representation of various heap graphs. Moreover, some data structures are defined with special relations between nodes or sets of nodes (e.g., red-black trees have red and black nodes with the special rules how they can alternate). Therefore the formalism should make it possible to represent also relations over graphs.

In the field of formal logic, a natural choice is monadic second order logic on graphs (MSO). It allows one to quantify over sets of nodes of graphs. Unfortunately, the logic is undecidable. There are different versions of graph automata with expressive power equal to MSO, i.e., for each MSO formula there is a graph automaton whose language (which is a set of graphs) is equivalent to the set of models of the formula. However, the undecidability of the logic implies that automata accepting such graphs have undecidable crucial properties, e.g., emptiness of the automata language. This is exactly what happens in the works [106, 102] where the designed automata suffer from the undecidability of some properties.

The undecidability does not necessarily imply that a formalism is not usable, but it is better to target decidability or if that is not possible, at least a formalism allowing for a design of efficient algorithmic heuristics.

When accepting a graph, an automaton needs to remember which nodes have already been processed and which will be processed further. This is a difference compared to the classical word or tree automata where an automaton runs over a given input in a particular direction. Remembering the so-far processed parts of the graphs is not easy task. In [106], the path through a processed graph is encoded in symbols. In [102], there is a special automaton for each node of the graph, the automata communicate in rounds, and the computation continues until all automata reach their stable state.

However, when the domain is restricted to a special class of graphs, it is possible to design automata with much more useful properties. It has been shown by Courcelle [42] that when we restrict MSO to graphs with a bounded tree width, it is possible to decide satisfiability of MSO formulas. A decision procedure is then implemented by encoding the formula to a tree automaton and checking its emptiness. Unfortunately, the tree automaton has an exponential number of states compared to the size of the formula, which makes the manipulation with it inefficient.

Another approach to the definition of graph automata is from an algebraic point of view. In the work [25], the automata were defined using concepts of *cospans* from category theory. Intuitively, the automata have as symbols some basic graph operations which can create an arbitrary graph with a bounded tree width. Such a definition implies that word and graph automata are basically the same because the graph operations can be viewed just as standard letters from an alphabet of word automata. Then a word automaton accepts a word consisting of letters mapped to the graph operations. When the operations are concatenated, we obtain the encoded graph. However, the authors have not found an efficient automatic derivation of their model from a system description.

Grammars are a standard counterpart to automata in the formal language theory. For instance, context-free grammars characterize the class of context-free languages just as pushdown automata. The paper [33] explores the relation between automata working over graphs and graph grammars and shows that they describe the same class of graph languages as the NCE graph grammars.

Graph grammars can also be naturally compared with the separation logic. The relation has been shown in Jansen et. all [88]. They show that so-called *tree-like separation logic* can describe the same graphs as those that can be generated by the so-called *tree-like grammars (TLG)*. Both formalisms are meant to describe tree-like data structures (i.e., data structures decomposable to trees). Further, a TLG formula can be transformed to the MSO_2 logic which is a variant of MSO on graphs where the quantification is allowed not only over sets of nodes but also over edges.

Both of the works comparing graph automata to grammars and separation logic explore the expressive power of the automata and their relation to other formalisms but do not concern efficient algorithms and their properties. Hence, they are not really suitable for applications in shape analysis.

Graph automata on graphs with bounded tree width can be applied for shape analysis indirectly as a decision procedure for separation logic. This is illustrated by [71] where a variation of automata accepting graphs with bounded tree width are used to encode graphs represented by a fragment of separation logic.

The notion of graph automata on graphs with bounded tree width has also been used for proving properties of other abstract machines. An example is emptiness of automata with auxiliary storage in [83].

In this work (Chapter 5) we present graph automata over graphs with bounded tree wdith that, unlike presented works, aims to efficient manipulation within automata-based shape analysis.

Chapter 3

Shape Analysis based on Forest Automata

3.1 Introduction

This chapters presents some of the main contributions of this the thesis, namely, the proposed way of checking spuriousness of potential counterexample traces within forest-automata-based shape analysis, the proposed refinement of the abstraction used, as well as the improvements done to the implementation of forest-automata-based shape analysis in the Forester tool.

In order for these approaches to be presented, the basic shape analysis based on forest automata is introduced first. We note here that while basic shape analysis based on forest automata were introduced in [54, 66, 57, **?**, 60], the conference publications [54, 66] were missing quite some details, the technical reports [**?**, 60] were not fully consistent and easy to read. Therefore they were revisted as part of our work and this revised version was accepted as a chapter of a book about tools participating in software verification competition SV-COMP. Further we modified basic approach using forest automata with respect to [65] to allow the analysis of potenatial counterexamples traces and abstraction refinement to be added in a smooth way. The revised description of the method is also presented in this chapter.

Forest-automata-based shape analysis is an approach to verification of sequential C-like programs with complex *dynamic linked data structures* such as various forms of singly and doubly-linked lists (SLLs/DLLs), possibly cyclic, shared, hierarchical, and/or having different additional (head, tail, data, and similar) pointers, as well as various forms of trees, based on the so-called *forest automata*. The approach is suitable for verification of generic *safety properties* like absence of null dereferences, double free operations, dealing with dangling pointers, or memory leakage. Moreover, Verification of special shape properties of the involved data structures is allowed via *testers*, i.e., additional parts of the code that, in the case some desired property is broken, lead the control flow to a designated error location.

Forest automata represent sets of heaps via *tree automata* (TAs). A heap is split in a canonical way into several *tree components* whose roots are the so-called *cut-points*. Cut-points are nodes pointed to by program variables or having several incoming edges. The tree components can refer to the roots of each other, and hence they are "separated" much like heaps described by formulae joint by the separating conjunction in sep-

aration logic [103]. Using this decomposition, sets of heaps with a bounded number of cut-points can be represented by *forest automata* (FAs) that are basically tuples of TAs accepting tuples of trees whose leaves can refer back to the roots of the trees. Moreover, alphabets of FAs may contain *nested FAs*, leading to a *hierarchical encoding of heaps*, so that FAs can represent even sets of heaps with an unbounded number of cut-points (e.g., sets of DLLs). Intuitively, a nested FA can describe a part of a heap with a bounded number of cut-points (e.g., a DLL segment), and by using such an automaton as an alphabet symbol an unbounded number of times, heaps with an unbounded number of cut-points are described.

In [54], it was shown that *entailment* of non-nested FAs (i.e., having a bounded number of cut-points) is decidable. This covers sets of complex structures like SLLs with head/tail pointers. It was also showed that entailment can be decided or quite precisely approximated for a large class of nested FAs. Further, C program statements manipulating pointers can be encoded as operations modifying FAs. This made it possible to generalise the essential parts (mainly the forward state space exploration) of the framework of *abstract regular tree model checking (ARTMC)* [31, 28] to forest automata and implement a shape analyser based on it. Finally, our work [65] (which is one of main contributions of this thesis) shows that it is possible to compute forest automata intersection (under-approximated or even precise for a large class of nested FAs) and implement a generalisation of the counterexample-based abstraction refinement of [31, 28] based on it.

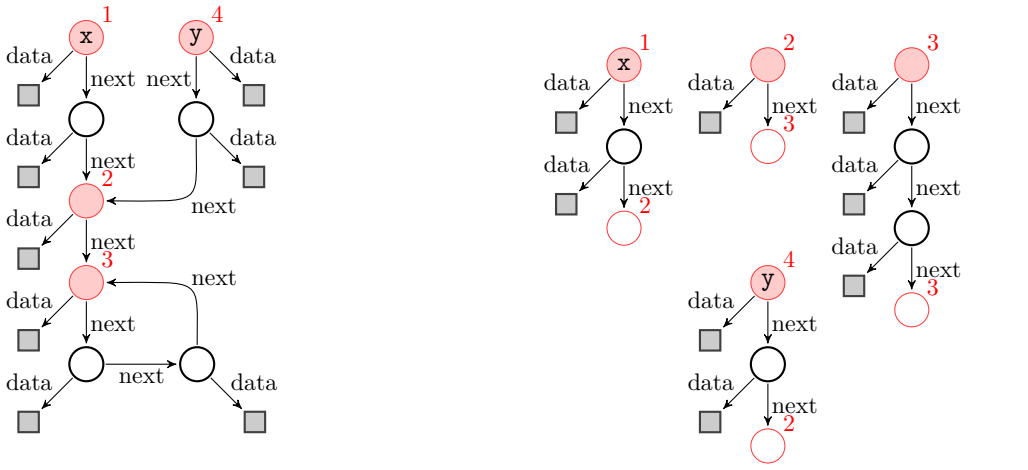
The proposed approach brings the principle of *local heap manipulation* (i.e., dealing with separated parts of heaps) from separation logic into the world of automata. It combines some advantages of using automata and separation logic. Automata provide higher generality and flexibility of the abstraction and allow one to leverage the recent advances of efficient use of non-deterministic automata [8, 10]. As further discussed below, the use of separation allows for a further increase in efficiency compared to a monolithic automata-based encoding proposed in [28].

Forest-automata-based symbolic execution and abstraction refinement was implemented in a prototype tool called Forester. The tool is able to fully automatically verify complex properties of programs with complex data structures such as various flavours of (nested and/or circular) lists, trees, or skip lists. In SV-COMP, it was able to handle a number of benchmarks with complex data structures that no other tool could successfully verify.

3.2 From Heaps to Forests

In this section, the concept of hierarchical forest automata as a symbolic representation of heaps in an informal way is outlined. For the purpose of the explanation, *heaps* may be viewed as directed graphs whose nodes correspond to allocated memory cells and edges to pointer links between these cells. The nodes may be labelled by non-pointer data stored in them (assumed to be from a finite data domain) and by program variables pointing to the nodes. Edges may be labelled by the corresponding selectors of data structures.

In what follows, the description is restricted to *garbage-free heaps*, in which all memory cells are reachable from pointer variables by following pointer links. This is, however, not a restriction in practice since the emergence of garbage can be checked for each executed program statement. If some garbage arises, an error message can be issued and the symbolic computation stopped. Alternatively, the garbage can be removed and the computation resumed.



(a) A heap graph with cut-points highlighted in red

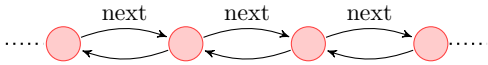
(b) The canonical tree decomposition of the heap with `x` ordered before `y`

Figure 3.1: A heap graph and its tree decomposition

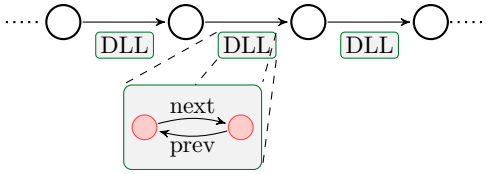
It is easy to see that each heap graph can be *decomposed* into a set of *tree components* when the leaves of the tree components are allowed to reference back to the roots of these components. Moreover, given a total ordering on program variables and selectors, each heap graph may be decomposed into a tuple of tree components in a *canonical way* as illustrated in Figs. 3.1a and 3.1b. In particular, one can first identify the so-called *cut-points*, i.e., nodes that are either pointed to by a program variable or that have several incoming edges. Next, the cut-points can be canonically numbered using a depth-first traversal of the heap graph starting from nodes pointed to by program variables in the order derived from the order of the program variables and respecting the order of selectors. Subsequently, one can split the heap graph into tree components rooted at particular cut-points. These components should contain all the nodes reachable from their root while not passing through any cut-point, plus a copy of each reachable cut-point, labelled by its number.

The notion of forest automata builds upon the described decomposition of heaps into tree components. In particular, a *forest automaton* (FA) is basically a tuple of tree automata (TAs). Each of the TAs accepts trees whose leaves may refer back to the roots of any of these trees. An FA then represents exactly the set of heaps that may be obtained by taking a single tree from the language of each of the component TAs and by glueing the roots of the trees with the leaves referring to them.

The described encoding allows one to represent sets of heaps with a bounded number of cut-points. Many common dynamic data structures, however, have an *unbounded number of cut-points*. Indeed, for instance, in doubly-linked lists (DLLs), every node is a cut-point. The problem of an unbounded number of cut-points is solved by representing heaps in a *hierarchical way*. In particular, one collects sets of repeated subgraphs (called *components*) containing cut-points into the so-called *boxes*. Every occurrence of such components can then be replaced by a single edge labelled by the appropriate box. To specify how a subgraph enclosed within a box is connected to the rest of the graph, the subgraph is equipped with the so-called input and output ports. The source vertex of a box then matches the input

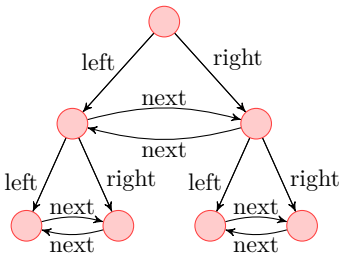


(a) A part of a DLL

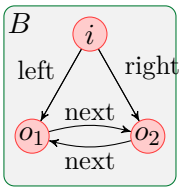


(b) A hierarchical encoding of the DLL

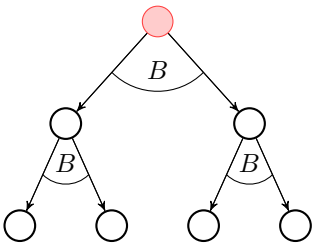
Figure 3.2: Encoding of a DLL using boxes



(a) A tree with linked sibling nodes



(b) A pattern that repeats in the structure and that is linked in such a way that all nodes in the structure are cut-points



(c) The tree with linked sibling nodes represented using hyperedges labelled by the box B

Figure 3.3: An example of a data structure where boxes with multiple output ports are necessary

port of the subgraph, and the target vertex of the edge matches the output port.[1] In this way, a set of heap graphs with an unbounded number of cut-points can be transformed into a set of *hierarchical heap graphs* with a bounded number of cut-points at each level of the hierarchy. Figs. 3.2a and 3.2b illustrate how this approach can reduce the representation of DLLs into singly-linked lists (with a DLL segment used as a kind of a meta-selector).

In general, a box is allowed to have more than one output port. Boxes with multiple output ports, however, reduce heap graphs not to graphs but rather *hypergraphs* with *hyperedges* having a single source node, but multiple target nodes. This situation is illustrated on a simple example shown in Fig. 3.3. The tree with linked siblings from Fig. 3.3a is turned into a hypergraph with binary hyperedges shown in Fig. 3.3c using the box B from Fig. 3.3b. The subgraph encoded by the box B can be connected to its surroundings via its input port i and *two* output ports o_1 and o_2. Therefore, the hypergraph from Fig. 3.3c encodes one level of the tree by a hyperedge with one source and *two* target nodes. A formalisation using hypergraphs and hyperedges was used in [57]. Here, in order to simplify

[1] Later on, the term *input port* will be used to refer to the nodes pointed to by program variables too since these nodes play a similar role as the inputs of components.

the formal development, ordinary graphs are used, in which a hyperedge is represented by a set of ordinary edges (which always need to occur together) from a single source to several targets.

Sets of heap graphs corresponding either to the top level of the representation or to boxes of different levels can then be decomposed into tree components and represented using *hierarchical FAs*, whose alphabets can contain nested FAs. Intuitively, FAs appearing in the alphabet of some higher-level FA play a role in some sense similar to that of inductive predicates in separation logic.[2] The approach is restricted to automata whose nesting forms a finite and strict hierarchy (i.e., there is no circular use of the automata in their alphabets). This is obviously different from separation logic, where recursive inductive predicates are standard. This kind of recursion is in forest automata represented through cycles of tree automata transitions.

3.3 Forest Automata and Heaps

We consider sequential non-recursive C programs operating on a set of pointer variables and the heap using standard statements and control flow constructs. Heap cells contain zero or several pointer or data fields.

Configurations of the considered programs consist of memory-allocated data and an assignment of variables. *Heap memory* can be viewed as a (directed) graph whose nodes correspond to allocated memory cells. Every node contains a set of named pointer and data fields. Each pointer field points to another node (we model the `NULL` address and undefined locations as special memory nodes pointed by variables `NULL` and `undef`, respectively), and the same holds for pointer variables of the program. Data fields of memory nodes hold a data value. Term *selector* is used to talk both about pointer and data fields. For simplification, data variables are modelled as pointer variables pointing to allocated nodes that contain a single data field with the value of the variable, and therefore consider only pointer variables hereafter.

Heap memory is represented by partitioning it into a tuple of trees, the so-called *forest*. The leaves of the trees contain information about roots of which trees they should be merged with to recover the original heap. The symbolic representation of a set of heaps by forest automata is based on representing the obtained sets of forests using tuples of tree automata.

Graphs and Heaps

Let Γ be a finite set of *selectors* and Ω be a finite set of *references* such that $\Omega \cap \mathbb{D} = \emptyset$. A *graph* g over $\langle \Gamma, \Omega \rangle$ is a tuple $\langle V_g, next_g \rangle$ where V_g is a finite set of *nodes* and $next_g : \Gamma \to (V_g \rightharpoonup (V_g \cup \Omega \cup \mathbb{D}))$ maps each selector $a \in \Gamma$ to a partial mapping $next_g(a)$ from nodes to nodes, references, or data values. References and data values are treated as special terminal nodes that are not in the set of regular nodes, i.e., $V_g \cap (\Omega \cup \mathbb{D}) = \emptyset$. For a graph g, we use V_g to denote the nodes of g, and for a selector $a \in \Gamma$, we use a_g to denote the mapping $next_g(a)$. The triple (v, a, u) is an *edge* of g if $a_g(v) = u$. Given a finite set of variables $\mathbb{X}$, a *heap* h over $\langle \Gamma, \mathbb{X} \rangle$ is a tuple $\langle V_h, next_h, \sigma_h \rangle$ where $\langle V_h, next_h \rangle$ is a graph over $\langle \Gamma, \emptyset \rangle$ and $\sigma_h : \mathbb{X} \to V_h$ is a (total) map of variables to nodes.

[2] For instance, a nested FA is used to encode a DLL segment of length 1. In separation logic, the corresponding inductive predicate would represent segments of length 1 or more. In forest-automata-based approach, the repetition of the segment is encoded in the structure of the top-level FA.

Forest Representation of Heaps

A graph t is a *tree* if its nodes and pointers form a tree with a unique root node, denoted $root(t)$ (references and data fields may have multiple incoming edges). A *forest* over $\langle \Gamma, \mathbb{X} \rangle$ is a pair $\langle t_1 \cdots t_n, \sigma_f \rangle$ where $t_1 \cdots t_n$ is a sequence of trees over $\langle \Gamma, \{\overline{1}, \ldots, \overline{n}\} \rangle$ and σ_f is a (total) mapping $\sigma_f : \mathbb{X} \to \{\overline{1}, \ldots, \overline{n}\}$. The elements in $\{\overline{1}, \ldots, \overline{n}\}$ are called *root references* (note that n must be the number of trees in the forest). A forest $\langle t_1 \cdots t_n, \sigma_f \rangle$ over $\langle \Gamma, \mathbb{X} \rangle$ represents a heap over $\langle \Gamma, \mathbb{X} \rangle$, denoted $\otimes\langle t_1 \cdots t_n, \sigma_f \rangle$, obtained by taking the union of the trees of $t_1 \cdots t_n$ (assuming w.l.o.g. that the sets of nodes of the trees are disjoint), connecting root references with the corresponding roots, and mapping every defined variable x to the root of the tree indexed by x. Formally, $\otimes\langle t_1 \cdots t_n, \sigma_f \rangle$ is the heap $h = \langle V_h, next_h, \sigma_h \rangle$ defined by (i) $V_h = \bigcup_{i=1}^{n} V_{t_i}$, (ii) for $a \in \Gamma$ and $v \in V_{t_k}$, if $a_{t_k}(v) \in \{\overline{1}, \ldots, \overline{n}\}$ then $a_h(v) = root(t_{a_{t_k}(v)})$ else $a_h(v) = a_{t_k}(v)$, and, finally, (iii) for every $x \in \mathbb{X}$ it holds that $\sigma_h(x) = root(t_{\sigma_f(x)})$.

3.3.1 Forest Automata

A forest automaton is essentially a tuple of tree automata accepting a set of tuples, each tuple being a forest decomposition of a graph, associated with a mapping of variables to root references.

Tree Automata

To simplify the formal development, we use a definition of tree automata specialised to the above sketched application in shape analysis.

A (finite, non-deterministic) *tree automaton* (TA) over $\langle \Gamma, \Omega \rangle$ is a triple $A = (Q, q_0, \Delta)$ where Q is a finite set of *states* (we assume $Q \cap (\mathbb{D} \cup \Omega) = \emptyset$), $q_0 \in Q$ is the *root state* (or initial state), denoted $root(A)$, and Δ is a set of *transitions*. Each transition τ is of the form $q \to \overline{a}(q_1, \ldots, q_m)$ where $m \geq 0$, $q \in Q$, $q_1, \ldots, q_m \in (Q \cup \Omega \cup \mathbb{D})$[3], and $\overline{a} = a^1 \cdots a^m$ is a sequence of different symbols from Γ. We call q the *parent state* of τ and $q_1, \ldots, q_m$ *child states*. Additionally, we assume the sequence of symbols $\overline{a} = a_1 \cdots a_m$ is always ordered according to some total ordering on Γ (this is important when checking entailment of forest automata, cf. Section 3.3.3, or constructing their intersection, cf. Section 3.5).

Let t be a tree over $\langle \Gamma, \Omega \rangle$, and let $A = (Q, q_0, \Delta)$ be a TA over $\langle \Gamma, \Omega \rangle$. A *run* of A over t is a total map $\rho : V_t \to Q$ where $\rho(root(t)) = q_0$ and, for each node $v \in V_t$, there is a transition $q \to \overline{a}(q_1, \ldots, q_m)$ in Δ with $\overline{a} = a^1 \cdots a^m$ such that $\rho(v) = q$ and for all $1 \leq i \leq m$, we have (i) if $q_i \in Q$, then $a_t^i(v) \in V_t$ and $\rho(a_t^i(v)) = q_i$, and (ii) if $q_i \in \Omega \cup \mathbb{D}$, then $a_t^i(v) = q_i$. The *language* of A is defined as $L(A) = \{t \mid \text{there is a run of } A \text{ over } t\}$, and the language of a state $q \in Q$ as $L(A, q) = L((Q, q, \Delta))$.

Forest Automata

A *forest automaton* (FA) over $\langle \Gamma, \mathbb{X} \rangle$ is a tuple of the form $F = \langle A_1 \cdots A_n, \sigma \rangle$ where $A_1 \cdots A_n$, with $n \geq 0$, is a sequence of TAs over $\langle \Gamma, \{\overline{1}, \ldots, \overline{n}\} \rangle$ whose sets of states $Q_1, \ldots, Q_n$ are mutually disjoint, and $\sigma : \mathbb{X} \to \{\overline{1}, \ldots, \overline{n}\}$ is a mapping of variables to root references. A forest $\langle t_1 \cdots t_n, \sigma_f \rangle$ over $\langle \Gamma, \mathbb{X} \rangle$ is *accepted* by F iff $\sigma_f = \sigma$ and $\forall 1 \leq i \leq n : t_i \in L(A_i)$.

[3] For simplicity, data values and references are used as special leaf states accepting the data values and references they represent, instead of having additional leaf transitions to accept them.

The *language* of F, denoted as $L(F)$, is the set of heaps over $\langle \Gamma, \mathbb{X} \rangle$ obtained by applying $\otimes$ on forests accepted by F.

3.3.2 Boxes and Hierarchical Forest Automata

Forest automata, as defined in Section 3.3.1, can represent heaps with cut-points of an unbounded in-degree as, e.g., in singly-linked lists (SLLs) with head/tail pointers (indeed, there can be any number of references from leaf nodes to a certain root). The basic definition of FAs cannot, however, deal with heaps with an unbounded number of cut-points, since this would require an unbounded number of TAs within the FAs. An example of such a set of heaps is the set of all doubly-linked lists (DLLs) of an arbitrary length, where each internal node is a cut-point. The solution provided in [54] is to allow FAs to use other nested FAs, called *boxes*, as symbols to "hide" recurring subheaps and in this way eliminate cut-points. The alphabet of a box itself may also include boxes, though it is required that they them to form a finite hierarchy—they cannot be recursively nested.[4]The language of a box is a set of heaps over $k+1$ special variables, $\mathtt{in}$ and $\mathtt{out}_1, \ldots, \mathtt{out}_k$, which correspond to the one input and k output ports of the box respectively.

An FA with the references $\{\mathtt{in}, \mathtt{out}_1, \ldots, \mathtt{out}_k\}$, for $k \geq 1$, is called a k-ary *box*; the arity of B is denoted as $\sharp B = k$. An FA over $\langle \Gamma, \mathbb{X} \rangle$ is a *nested/hierarchical* FA over $\langle \Gamma, \mathbb{X} \rangle$ of level 1, and a nested FA over $\langle \Gamma, \mathbb{X} \rangle$ of level $\ell > 1$ is an FA over $\langle \Gamma \cup \mathcal{B}, \mathbb{X} \rangle$ where $\mathcal{B}$ may contain selectors of the form $B_{(i)}$, where B is a nested box over $\langle \Gamma, \mathbb{X} \rangle$ of a level smaller than ℓ and $1 \leq i \leq \sharp B$. Finally, given a k-ary box B, a k-tuple of graph edges $(v, B_{(1)}, u_1), \ldots, (v, B_{(k)}, u_k)$ is called a *hyperedge* of B from the source node v to the target nodes $u_1, \ldots, u_k$.

In the case of a nested FA F, one needs to distinguish between its language $L(F)$, which is a set of heaps over $\langle \Gamma \cup \mathcal{B}, \mathbb{X} \rangle$, and its *semantics* $[\![F]\!]$, which is a set of heaps over $\langle \Gamma, \mathbb{X} \rangle$ that emerges when all boxes in the heaps of the language are recursively *unfolded* in all possible ways.

A heap h' is called an unfolding of a heap h if h has edges $(u, B_{(1)}, v_1), \ldots, (u, B_{(k)}, v_k)$ labeled with indexed variants of a box $B = \langle A_1 \cdots A_n, \sigma_{\mathcal{B}} \rangle$ of the rank k, and h' is obtained from h by substituting these edges by a graph from $L(B)$, connecting the input and output ports of B at u and $v_1, \ldots, v_k$, respectively. That is, all edges $(u, B_{(1)}, v_1), \ldots, (u, B_{(k)}, v_k)$ are removed, and the remainder of h is united with a heap $h_B \in L(B)$, with the variable map σ_B, in which $\sigma_B(\mathtt{in}) = u, \sigma_B(\mathtt{out}_1) = v_1, \ldots, \sigma_B(\mathtt{out}_k) = v_k$, and all its remaining nodes do not appear in h. We then write $h \rightsquigarrow_{B, u/h_B} h'$ or simply $h \rightsquigarrow h'$. Then $\rightsquigarrow^*$ is used to denote the reflexive transitive closure of $\rightsquigarrow$. The *semantics* of F, written as $[\![F]\!]$, is the set of all heaps h' over $\langle \Gamma, \mathbb{X} \rangle$ for which there is a heap h in $L(F)$ such that $h \rightsquigarrow^* h'$.

3.3.3 Entailment of Forest Automata

The *entailment*, or semantic inclusion, test of forest automata is needed when testing convergence of our program analysis. In this section, an overview of the techniques described in detail in [54, 107] is given.

The starting point is a component-wise language inclusion test. That is, for a pair of FAs $F = \langle A_1 \cdots A_n, \sigma \rangle$ and $F' = \langle A'_1, \ldots, A'_m, \sigma' \rangle$, the *component-wise test* $F \sqsubseteq F'$ returns true iff $m = n$, $\sigma = \sigma'$, and $L(A_i) \subseteq L(A'_i)$ for every $1 \leq i \leq n$. The test safely approximates

[4]Recursive boxes would introduce complications in symbolic execution and checking entailment. Instead, recursively repeating structure of heap graphs is with FAs expressed using cycles of transitions in TAs.

the language inclusion $L(F) \subseteq L(F')$ and hence also semantic entailment $[\![F]\!] \subseteq [\![F']\!]$. It is efficient, allowing us to use fast tree automata language checking algorithms such as [10]. It is incomplete though—if it returns *false*, the language inclusion and entailment may still hold. In the following, it is discussed how the test can be made more precise by means of converting the FAs into special normal forms.

Dense and Canonic Form of Non-hierarchical FAs

A *cut-point* of a heap h was defined as a node that is either pointed by some variable or is a target of more than one selector edge. The roots of trees of a forest f that are not cut-points in the heap $\otimes f$ represented by f are called *false roots*. A forest is *dense* if it does not have false roots, and a dense FA accepts only dense forests. Each FA can be transformed into a set of dense FAs that together have the same language as the original. Density will be used in abstraction refinement and construction of intersections of FAs discussed in Section 3.5.

Density is a part of *canonicity*. Canonicity extends density with the requirements that (1) every node of the corresponding graph is reachable from a node assigned to a variable and (2) the trees of the forest are ordered by the discovery time of their roots in the depth-first traversal starting from the nodes marked by the variables. The discovery times of nodes in the traversal depend on a predetermined ordering of the variables and selectors, according to which the traversal is steered. With the ordering fixed, the discovery times are completely deterministic (recall that every node in a heap is reachable from some variable and that a node has at most one successor for every selector). Hence, there is indeed exactly one canonic forest representation for every heap. An FA is *canonicity-respecting* if it accepts only canonical forests. The transformation into the canonicity-respecting form is discussed in detail in [54, 107]. It may transform a single FA into a set of canonicity-respecting FAs such that the semantics of FAs in the set form a partition of the semantics of the original FA. The component-wise test over canonicity-respecting automata is then a precise test of their language, so it is also a precise semantic entailment test of non-hierarchical forest automata (since the language of a non-hierarchical FA coincides with its semantics).

To test language inclusion of a pair of non-canonicity-respecting FAs F and F' in a sound and complete way, one first converts the two FAs into sets S and S' of canonicity-respecting FAs, respectively. The second step is testing language inclusion of the two sets, that is, testing whether $\bigcup_{F_a \in S} L(F_a) \subseteq \bigcup_{F_b \in S'} L(F_b)$.

The language inclusion of any two sets S and S' of canonicity respecting FA can be decided in a sound and complete way by converting each set into a single tree automaton and testing language inclusion of the two tree automata. The set S is converted into the tree automaton A^S, created as follows: Every forest automaton $F = (A_1 \cdots A_n, \sigma) \in S$ is converted into the tree automaton A^F that accepts exactly trees of the form $\sigma(t_1, \ldots, t_n)$ where $t_i \in L(A_i)$, for $1 \leq i \leq n$. That is, A^F accepts trees that arise by taking a tree from each A_i, and connecting them below a new common root, labeled by σ as a new symbol. This tree automaton is created from the tree automata union of $A_1, \ldots, A_n$ by introducing a new root state q and transitions $q \to (q_1, \ldots, q_n)$ where, for $1 \leq i \leq n$, it holds that q_i is the root state of A_i (one assumes w.l.o.g. that $A_1, \ldots, A_n$ have disjoint sets of states and, as mentioned later, the FAs are kept in the form where each TA has one root transition). A^S is then obtained as the tree automata union of all A^F for $F \in S$. The tree automaton $A^{S'}$ is constructed analogously from S'. It then holds that $\bigcup_{F_a \in S} L(F_a) \subseteq \bigcup_{F_b \in S'} L(F_b) \iff L(A^S) \subseteq L(A^{S'})$, which is equivalent to $\bigcup_{F_a \in S} [\![F_a]\!] \subseteq \bigcup_{F_b \in S'} [\![F_b]\!]$ for non-hierarchical FA.

Entailment and Canonicity-respecting Form of Hierarchical FAs

Entailment of hierarchical FAs is substantially more difficult since they can hide parts of heaps into boxes in an arbitrary way. We believe (though that has not been proved yet) that the problem is decidable, considering that the expressive power of hierarchical FAs is close to the decidable fragments of separation logic with inductive predicates studied within [71, 74]. Here is presented a solution that is (theoretically speaking) incomplete, but practical in the context of shape analysis with forest automata as the abstract domain—it is fast, relatively simple, and precise enough. To describe this solution, we again start from the component-wise test discussed above, and elaborate on how and when it can be made more precise by transformations of FAs to normal forms.

The first complication compared to the case of non-nested FAs appears already when converting to the canonicity-respecting form (even when the semantics of boxes is not taken into account yet). Namely, the conversion requires that all nodes of the heaps in the semantics of an FA are reachable from variables. With nested FAs, even if the nodes of heaps of $[\![F]\!]$ are reachable from variables, some nodes might, however, be unreachable in the corresponding heaps from $L(F)$. For instance, a path in a graph of $[\![F]\!]$ from a variable `x` reaching some node u might lead through boxes *against* the direction of the box-labelled edges (the path uses a sub-path through a graph g in $[\![B]\!]$ that leads from an output port to the input port of g). Such a path from `x` to u would then on the top level—in the corresponding graph of $L(F)$—appear as leading from the node u to a leaf labeled by the variable `x`. Hence, when boxes are viewed as plain selectors, u is not reachable but only *backward*-reachable from a variable. The notion of canonicity and the canonicity-respecting form for hierarchical FAs is hence in [57, 107] modified to take into account such paths as well. Essentially, one pre-computes the so-called *port interconnection* of boxes, i.e., how the graphs encoded by boxes interconnect their ports, and takes this into account in the depth-first traversal, which determines the canonic order of the tree components. This way, one can still have a precise language inclusion test of canonicity-respecting hierarchical FA, but since the full semantics of boxes is not taken into account, it is only a sound approximation of the semantic entailment.

The works [54, 107] go one step further and define a sub-class of nested FAs for which the generalised canonicity with the component-wise test is a complete semantic entailment test. In short, for the test of the entailment of FAs F and F' to be complete, it must hold that (1) the heaps of $L(F)$ and $L(F')$ are "maximally boxed", meaning that if a sub-graph is not hidden in a box, then there is no box of $L(F)$ or $L(F')$ that would contain it, (2) the sets of graphs hidden in different boxes are disjoint, and (3) different graphs hidden in the boxes are not "overlapping" (meaning, e.g., that one box would hide a DLL segment of length one and another box would hide a DLL segment of length two). We note that although these restrictions may seem strong, The program analysis using forest automata is designed in such a way that they are usually satisfied or can easily be enforced. The fact that the entailment test is oblivious to the full semantics of boxes (apart from the pre-computed information of port interconnection, it treats boxes as ordinary selectors) makes it also highly efficient.

Root Interconnection Graph

In the practice of shape analysis based on forest automata, an additional heuristic is used to speed up entailment testing. In particular, every FA is kept paired with the so-called *root interconnection graph* [107], which contains information about reachability between its

cut-points. This allows a very fast refutation of their entailment in case the reachability relation is incompatible.

The *root interconnection graph* of a forest $f = \langle t_1 \cdots t_n, \sigma \rangle$ is a (directed) graph $G = (V, E)$ in which the nodes $V = \{t_1, \ldots, t_n\}$ represent the roots of the trees $t_1, \ldots, t_n$, and the edges $E \subseteq V \times (\mathbb{N} \times \{1, 2\}) \times V$ represent the interconnection of the roots through the paths in f. In particular, an edge labeled by (k, ℓ) appears between t_i and t_j in G if and only if the depth-first traversal (DFT) started in the root of t_i visits a reference to t_j after visiting $k - 1$ other root references (when multiple occurrences of the same references are counted as one). If t_j is not visited anymore in the rest of the DFT, then $\ell = 1$, otherwise $\ell = 2$.

Forest automata are kept in a form in which they accept forests with the same root interconnection graph (this is actually a stronger property than the dense form). One can then talk about the root interconnection graph of a forest automaton. Such FAs can be transformed to the canonicity-respecting form by permuting the TA components based on the information on the edges of the root interconnection graph. Languages of two forest automata in the canonicity-respecting form and with defined root interconnection graphs may then intersect (or be included in one another) only if their root interconnection graphs are the same. Root interconnection graphs are thus used to speed up testing language inclusion of forest automata, computation of their intersection, and also to improve precision of their abstraction (discussed in detail in [107]).

3.4 Verification of Pointer Programs with Forest Automata

In this section, we provide an overview of the main components of the approach to verification of pointer programs using forest automata as implemented in the tool Forester. The detailed discussion of some of the components will be the subject of the further sections.

We start by discussing symbolic execution of C programs in the abstract domain of forest automata.[5] We will work on the level of the program's *control flow graph*, which is a mapping $p : \mathbb{T} \to (\mathbb{L} \times \mathbb{L})$ where $\mathbb{T}$ is a set of program statements and $\mathbb{L}$ is a set of program locations. Statements are partial functions $\tau : \mathbb{H} \rightharpoonup \mathbb{H}$ where $\mathbb{H}$ is the set of heaps over the selectors Γ and variables $\mathbb{X}$ occurring in the program. The heaps from $\mathbb{H}$ are used as representations of program configurations. The initial configuration is the heap $h_{\mathsf{init}} = \langle \emptyset, \emptyset, \emptyset \rangle$ (i.e., the empty graph and variable assignment). We assume that statements are indexed by their line of code, so that no two statements of a program are identical. If $p(\tau) = (\ell, \ell')$, then the program p can move from ℓ to ℓ' while modifying the heap h at the location ℓ to $\tau(h)$. We assume that $\mathbb{X}$ contains a special variable pc that always evaluates to a location from $\mathbb{L}$, and that every statement updates its value according to the target location. Note that a single program location can have multiple successors (corresponding, e.g., to conditional statements), or no successor (corresponding to exit points of the program). We use $src(\tau)$ to denote ℓ and $tgt(\tau)$ to denote ℓ' in the pair above. Every program p has a designated location ℓ_{init} called its *entry point* and $\ell_{\mathsf{err}} \in \mathbb{L}$ called the error location[6].

[5] By *symbolic execution*, we mean an execution of the program in the given abstract domain using abstract transformers (there are also other different notions of symbolic execution).

[6] For simplification, we assume checking the error line (un-)reachability property only, which is, anyway, sufficient in most practical cases. For detection of garbage (which is not directly expressible as line reachability), we can extend the formalism and check for garbage after every command, and if a garbage is found, we jump to ℓ_{err}.

A *program path* π in p is a sequence of statements $\pi = \tau_1 \cdots \tau_n \in \mathbb{T}^*$ such that $src(\tau_1) = \ell_{\mathsf{init}}$, and, for all $1 < i \leq n$, it holds that $src(\tau_i) = tgt(\tau_{i-1})$. We say that π is *feasible* iff $\tau_n \circ \cdots \circ \tau_1(h_{\mathsf{init}})$ is defined. The program p is safe if it contains no feasible program path with $tgt(\tau_n) = \ell_{\mathsf{err}}$. In the following, we fix a program p with locations $\mathbb{L}$, variables $\mathbb{X}$, and selectors Γ.

3.4.1 Symbolic Execution with Forest Automata

Safety of the program p is verified using symbolic execution in the domain $\mathbb{F}$ of forest automata over $\langle \Gamma, \mathbb{X} \rangle$. The program is executed symbolically by iterating abstract execution of program statements and a generalization step. These high-level operations are implemented as sequences of *atomic operations* and *splitting*. Atomic operations are partial functions of the type $o : \mathbb{F} \rightharpoonup \mathbb{F}$. Splitting splits an FA F into a set $\mathcal{S}$ of forest automata such that $[\![F]\!] = \bigcup_{F' \in \mathcal{S}} [\![F']\!]$. Splitting is necessary for some operations since forest automata are not closed under union, i.e., some sets of heaps expressible by a finite union of semantics of forest automata are not expressible by a single forest automaton.[7]

The symbolic execution builds the *abstract reachability tree* (ART) of the program, with branches being symbolic executions. Nodes of the ART are forest automata corresponding to sets of reachable configurations at particular program locations. The tree is rooted by the forest automaton F_{init} s.t. $[\![F_{\mathsf{init}}]\!] = \{h_{\mathsf{init}}\}$. Every other node is a result of an application of an atomic operation or a split on its parent, and the applied operation is recorded on the tree edge between the two. The atomic operation corresponds to one of the following: symbolic execution of an effect of a program statement, generalization, or an auxiliary meta-operation that modifies the FA while keeping its semantics (e.g., connects or cuts its components). Splitting appears in the tree as a node labelled by a special operation *split* with several children connected via edges labelled by a special operation *split*. The said operations are described in more detail in Section 3.6.

The tree is expanded starting from the root as follows: First, a symbolic configuration, represented using an FA, in the parent node is generalized by iterating the following three operations until fixpoint: (i) transformation of the FA into the dense form (see Section 3.3.3), (ii) application of regular abstraction over-approximating sets of sub-graphs between cut-points of the heaps represented by the FA (described in more detail in Section 3.7.2), and (iii) automatic discovery and folding of boxes to decrease the number of cut-points in the represented heaps (described in more detail in Section 3.8). The transformation into the dense form is performed in order to obtain the most general abstraction in the subsequent step. A configuration where one more loop of the transformation-abstraction-folding sequence has no further effect is called *stable*. Operations implementing effects of statements are then applied on stable configurations. Exploration of a branch is terminated if its last configuration is entailed (cf. Section 3.3.3) by a symbolic configuration with the same program location reached previously elsewhere in the tree (not necessarily on the given branch).

[7] To show an example of how a symbolic execution may generate a set of configurations not expressible using a single FA, assume that the statement `x = y->sel` is executed on a forest automaton that encodes cyclic singly-linked lists of an arbitrary length where `y` points to the head of the list. If the list is of length 1, then `x` will, after execution of the statement, point to the same location as `y`. If the list is longer, `x` and `y` will point to different locations. In the former case, the configuration has a single tree component, with both variables pointing to it. In the latter case, the two variables point to two different components. These two configurations cannot be represented using a single forest automaton.

A *symbolic path* is a path between a node and one of its descendants in the ART, i.e., a sequence of FAs and operations $\omega = F_0 o_1 F_1 \dots o_n F_n$ such that $F_i = o_i(F_{i-1})$. A *forward run* is a symbolic path where $F_0 = F_{\mathsf{init}}$. We write ω_i to denote the prefix of ω ending by F_i and ${}_i\omega$ to denote its suffix starting from F_i. A forward run that reaches ℓ_{err} is called an *abstract counterexample*. We associate every operation o with its *exact semantics* $\hat{o}$, defined as $\hat{o}(H) = \bigcup_{h \in H} \{\tau(h)\}$ if o implements the program statement τ, and as the identity for all other operations (operations implementing generalization, splitting, etc.), for a set of heaps H. The *exact execution* of ω is a sequence $h_0 \cdots h_n$ such that $h_0 \in [\![F_0]\!]$ and $h_i \in \hat{o}(\{h_{i-1}\}) \cap [\![F_i]\!]$ for $0 < i \leq n$. We say that ω is *feasible* if it has an exact execution, otherwise it is *infeasible/spurious*. Atomic operations in a symbolic path are either semantically precise, or over-approximate their exact semantics, i.e., it always holds that $\hat{o}([\![F]\!]) \subseteq [\![o(F)]\!]$. Therefore, if the exploration of the program's ART finds no abstract counterexample, there is no exact counterexample, and the program is safe.

3.4.2 Backward Run and Counterexample Analysis

Assume that the forward run $\omega = F_0 o_1 F_1 \cdots o_n F_n$ is spurious. Then there must be an index $i > 0$ such that the symbolic path ${}_i\omega$ is feasible but ${}_{i-1}\omega$ is not. This means that the operation o_i over-approximated the semantics of ω and introduced into $[\![F_i]\!]$ some heaps that are not in $\hat{o}_i([\![F_{i-1}]\!])$ and that are *bad* in the sense that they make ${}_i\omega$ feasible. An *interpolant for* ω is then a forest automaton I_i representing the bad heaps of $[\![F_i]\!]$ that were introduced into $[\![F_i]\!]$ by the over-approximation in o_i and are disjoint from $\hat{o}_i([\![F_{i-1}]\!])$. Formally,

1. $[\![I_i]\!] \cap \hat{o}_i([\![F_{i-1}]\!]) = \emptyset$ and
2. ω_i is infeasible from all $h \in [\![F_i]\!] \setminus [\![I_i]\!]$.

In the following, we describe how to use a backward run, which reverts operations of the forward run on the semantic level, to check spuriousness of an abstract counterexample. Moreover, we show how to derive interpolants from backward runs reporting spurious counterexamples, and how to use those interpolants to refine the operation of abstraction so that it will not introduce the bad configurations in the same way again.

A *backward run* for ω is the sequence $\overline{\omega} = \overline{F}_0 \cdots \overline{F}_n$ such that

1. $\overline{F}_n = F_n$ and
2. $[\![\overline{F}_{i-1}]\!] = \hat{o}_i^{-1}([\![\overline{F}_i]\!]) \cap [\![F_{i-1}]\!]$, that is, $\overline{F}_{i-1}$ represents the *weakest precondition* of $[\![\overline{F}_i]\!]$ w.r.t. $\hat{o}_i$ that is *localized* to $[\![F_{i-1}]\!]$.

If it happens that there is an FA $\overline{F}_i$ such that $[\![\overline{F}_i]\!] = \emptyset$ (and, consequently, $[\![\overline{F}_0]\!] = \emptyset, \dots, [\![\overline{F}_{i-1}]\!] = \emptyset$), the forward run is spurious. In such a case, an interpolant I_i for ω can be obtained as $\overline{F}_{i+1}$ where $i+1$ is the smallest index such that $[\![\overline{F}_{i+1}]\!] \neq \emptyset$. We elaborate on the implementation of the backward run in Section 3.7.1.

We note that our use of interpolants differs from that of McMillan [90] in two aspects. First, due to the nature of our backward run, we compute an interpolant over-approximating the source of the suffix of a spurious run, not the effect of its prefix. Second, for simplicity of implementation in our prototype, we do not compute a sequence of localized interpolants but use solely the interpolant obtained from the beginning of the longest feasible suffix of the counterexample for a global refinement. It would also, however, be possible to use the sequence $\overline{F}_i, \dots, \overline{F}_n$ as localized interpolants.

In Section 3.7.2, we show that using the interpolant I_i, it is possible to refine regular abstraction o_i (the only over-approximating operation) to exclude the spurious run. We can formulate *progress guarantees* for the next iterations of the CEGAR loop. The formulation will refer to the notion of *compatibility* of two FAs, which intuitively means that the two FAs represent the heaps from the intersection of their semantics in the same way: their boxes are folding the same sub-heaps of the represented heaps, and their TA components are partitioning the represented heaps into the same tree components. We will define compatibility formally in Section 3.5.2. The progress guarantees are that

1. for any FA F such that $[\![F]\!] \subseteq [\![F_{i-1}]\!]$ that is compatible with F_{i-1} it holds that $[\![o_i(F)]\!] \cap [\![I_i]\!] = \emptyset$, and
2. forward runs $\omega' = F'_0 o_1 F'_1 \cdots o_n F'_n$ such that for all $1 \leq j \leq n$, $[\![F'_i]\!] \subseteq [\![F_i]\!]$ and F'_i is compatible with F_i are excluded from the ART.

In the following sections, we elaborate in a greater detail on the components of the verification procedure outlined above. We discuss the construction of forest automata *intersection* in Section 3.5, then we describe how the *forward run* is implemented in Section 3.6, and how operations are inverted while checking spuriousness of a counterexample in a *backward run* in Section 3.7.1. We then discuss the *regular abstraction* and its *refinement* using the interpolants obtained from the backward run in Section 3.7.2. Lastly, we give more detail on how *boxes* to be folded are automatically identified within the transformation-abstraction-folding loop in Section 3.8.

3.5 Intersection of Forest Automata

The above presented inference of interpolants used the intersection of the semantics of forest automata to detect spuriousness of a counterexample. In this section, we give an algorithm that computes an under-approximation of the intersection of the semantics of a pair of FAs, and later give conditions (which are, in fact, met by the pairs of FAs in our backward run analysis) on the intersected FAs to guarantee that the computed intersection is precise.

3.5.1 Intersection Construction

A simple way to compute the intersection of the semantics of two FAs, denoted as $\cap$, is component-wise, that is, for two hierarchical FAs $F = \langle A_1 \cdots A_n, \sigma \rangle$ and $F' = \langle A'_1 \cdots A'_n, \sigma' \rangle$ over $\langle \Gamma, \mathbb{X} \rangle$ with $\sigma = \sigma'$, we compute the FA $F \cap F' = \langle (A_1 \cap A'_1) \cdots (A_n \cap A'_n), \sigma \rangle$. Note that intersection of forest automata with different numbers of components or different assignments of references is always empty. The tree automata product construction for our special kind of tree automata synchronizes on data values and on references. That is, a pair (a, b) that would be computed by a classical product construction where a or b is a reference or a data value is replaced by a if $a = b$, and removed otherwise (cf. [39] for the standard construction). This algorithm only under-approximates the actual semantic intersection, i.e., it is only guaranteed that $[\![F \cap F']\!] \subseteq [\![F]\!] \cap [\![F']\!]$.

To increase its precision, we replace the operator $\cap$ by the operator $\sqcap$ that takes into account the semantics of the boxes. Namely, we compute the FA $F \sqcap F'$ in a similar way as $F \cap F'$ but replace the tree automata product $A \cap A'$ by the construction of $A \sqcap A'$, which recursively calls $\sqcap$ on boxes. For TAs $A = (Q, q_0, \Delta)$ and $A' = (Q', q'_0, \Delta')$, it computes the

TA $A \sqcap A' = (Q \times Q', (q_0, q'_0), \Delta \sqcap \Delta')$ where $\Delta \sqcap \Delta'$ is built as follows:

$$\Delta \sqcap \Delta' = \big\{(q, q') \to a_1 \sqcap a'_1 \cdots a_n \sqcap a'_n((q_1, q'_1), \ldots, (q_m, q'_m)) \mid \\ q \to a_1 \cdots a_n(q_1, \ldots, q_m) \in \Delta, q' \to a'_1 \cdots a'_n(q'_1, \ldots, q'_m) \in \Delta'\big\}.$$

. Here, the symbol $a_i \sqcap a'_i$ is computed as follows:

1. If $a_i = a'_i$, then $a_i \sqcap a'_i = a_i$;
2. else if $a_i = B_{(i)}$ and $a'_i = B'_{(i)}$ where B and B' are boxes over $\langle \Gamma, \mathbb{X} \rangle$, then $a_i \sqcap a'_i = (B \sqcap B')_{(i)}$;
3. otherwise $a_i \sqcap a'_i$ as well as the whole transition is undefined (and no transition is added to $\Delta \sqcap \Delta'$).

3.5.2 Compatibility for Precise Intersection

In this section, we formally define the notion of *compatibility* of FAs. Compatibility is required in order to guarantee precision of the intersection of FAs from the forward and the backward runs. Intuitively, it means that the heaps represented by the FAs are folded into boxes in the same way. If this were not the case, due to the different positions (or types) of boxes occurring in the FAs, the intersection operation might yield an FA with the empty language although the intersection of the semantics of the FAs would be non-empty.

For a forest automaton $F = \langle A_1 \cdots A_n, \sigma \rangle$, its version with *marked components* is the FA $F^D = \langle A_1 \cdots A_n, \sigma \cup \sigma_{\mathsf{root}} \rangle$ where σ_{root} is the mapping $\{\mathsf{root}_1 \mapsto 1, \ldots, \mathsf{root}_n \mapsto n\}$. The *root variables* root_i are fresh variables that point to the roots of the tree components in $L(F)$. $[\![F^D]\!]$ then contains the same heaps as $[\![F]\!]$, but the roots of the components from $L(F)$ remain visible since they are explicitly marked by the root variables. In other words, the root variables track how the forest decomposition of the heaps in $L(F)$ partitions the heaps from $[\![F]\!]$. By removing the root variables of $h^D \in [\![F^D]\!]$, we get the original heap $h \in [\![F]\!]$. We call h^D the *component decomposition of h by F*.

Using the notion of component decomposition, we further introduce a notion of the *representation* of a heap by an FA. Namely, the *representation* of a box-free heap h by an FA F with $h \in [\![F]\!]$ records how F represents h, i.e., (i) how F decomposes h into components, and (ii) how its sub-graphs enclosed in boxes are represented by the boxes. Formally, the representation of h by F is a pair $\mathit{repre} = (h^D, \{\mathit{repre}_1, \ldots, \mathit{repre}_n\})$ such that h^D is the component decomposition of h by F, and $\mathit{repre}_1, \ldots, \mathit{repre}_n$ are obtained from the sequence of unfoldings

$$h_0 \rightsquigarrow_{B_1, u_1/g_1} h_1 \rightsquigarrow_{B_2, u_2/g_2} \cdots \rightsquigarrow_{B_n, u_n/g_n} h_n$$

with $h_0 = h^D$ and $h_n \in L(F^D)$, such that, for each $1 \leq i \leq n$, repre_i is (recursively) the representation of g_i in B_i.

We write $[\![\mathit{repre}]\!]$ to denote $\{h\}$, and, for a set of representations R, we let $[\![R]\!] = \bigcup_{\mathit{repre} \in R} [\![\mathit{repre}]\!]$. The set of *representations accepted by a forest automaton F* is the set $\mathit{Repre}(F)$ of all representations of heaps from $[\![F]\!]$ by F. We say that a pair of FAs F and F' is *(representation) compatible* iff $[\![F]\!] \cap [\![F']\!] = [\![\mathit{Repre}(F) \cap \mathit{Repre}(F')]\!]$. The compatibility of a pair of FAs intuitively means that for every heap from the semantic intersection of the two FAs, at least one of its representations is shared by them.

Lemma 1 *For a pair F and F' of compatible FAs, it holds that $[\![F \sqcap F']\!] = [\![F]\!] \cap [\![F']\!]$.*

3.6 Implementation of the Forward Run

This section describes the operations that are used to implement the forward symbolic execution over FAs. To be able to implement the backward run, we will need to maintain compatibility of the FAs (discussed in the previous section) between the forward run and the so-far constructed part of the backward run. Therefore, we will present the operations used in the forward run mainly from the point of view of their effect on the representation of heaps (in the sense of Section 3.5.2). Then, in Section 3.7.1, we will show how this effect is inverted in the backward run such that, when starting from compatible configurations, the inverted operations preserve compatibility of the configurations in the backward run with their forward run counterparts.

We omit most details of the way the operations are implemented on the level of manipulations with transitions and states of FAs. We refer the reader to [54, 107] for the details.

We note that when we talk about removing a component or inserting a component in an FA, this also includes renaming references and updating assignments of variables. When a component is inserted at a position i, all references to $\overline{j}$ with $j \geq i$ are replaced by $\overline{j+1}$, including the assignment σ of variables. When a component is removed from a position i, all references to $\overline{j}$ with $j > i$ are replaced by references to $\overline{j-1}$.

Splitting

Splitting has already been discussed in Section 3.4.1. It splits the symbolic execution into several branches such that the union of the FAs after the split is semantically equivalent to the original FA. The split is usually performed when transforming an FA into several FAs that have only one variant of a root transition of some of their components. From the point of view of a single branch of the ART, splitting is an operation, denoted further as *split*, that transforms an FA F into an FA F' s.t. $[\![F']\!] \subseteq [\![F]\!]$ and $\mathit{Repre}(F') \subseteq \mathit{Repre}(F)$. Therefore, F is compatible with F'.

Operations Modifying Component Decomposition

This class of operations is used to implement transformation of FAs to the dense form and as pre-processing steps before the operations of folding, unfolding, and symbolic implementation of program statements. They do not modify the semantics of forest automata, but change the component decomposition of the represented heaps.

- *Connecting of components.* When the j-th component A_j of a forest automaton F accepts trees with false roots, then A_j can be connected to the component that refers to it. Indeed, as such roots are not cut-points, a reference $\overline{j}$ to them can appear only in a single component, say A_k, and at most once in every tree from its language (because a false root can have at most one incoming edge). For simplicity, assume that A_j has only one root state q, which does not appear as a child state in transitions. The connection is done by adding the states and transitions of A_j to A_k, replacing the reference $\overline{j}$ in the transitions of A_k by q. The j-th component is then removed from F. The previous sequence of actions is below denoted as the operation $\mathit{connect}[j, k, q]$.

- *Cutting of a component.* Cutting splits a component with an index j into two. The part of the j-th component containing the root will accept tree prefixes of the original trees, and the new k-th component will accept their remaining sub-trees. The cutting

is done at a state q of A_j, which appears exactly once in each run (the FA is first transformed to a form that satisfies this). Occurrences of q as child states of transitions are replaced by the reference $\overline{k}$ to the new component, and q becomes the root state of the new component. We denote this operation by $cut[j, k, q]$.

- *Swapping of components.* The operation $swap[j, k]$ swaps the j-th and the k-th component (and renames references and assignments accordingly).

Folding of Boxes

In this section, we briefly describe the effect of folding FAs into boxes during our symbolic execution. More details and algorithms for selecting which part of which components of FAs are to be folded will be given in Section 3.8.

The folding operation assumes that the concerned FA is first transformed into the form $F = \langle A_{\mathtt{in}} A_2 \cdots A_{n-k} A_{\mathtt{out}_1} \cdots A_{\mathtt{out}_k} A'_1 \cdots A'_m, \sigma \rangle$ by a sequence of splitting, cutting, and swapping. The tuple of tree automata $A_{\mathtt{in}} A_2 \cdots A_{n-k} A_{\mathtt{out}_1} \cdots A_{\mathtt{out}_k}$ will then be folded into a new k-ary box B with $A_{\mathtt{in}}$ as its input component and $A_{\mathtt{out}_1}, \ldots, A_{\mathtt{out}_k}$ as its outputs. Moreover, the operation is given sets of selectors $S_{\mathtt{in}}, S_{\mathtt{out}_1}, \ldots, S_{\mathtt{out}_k}$ of roots of components in $A_{\mathtt{in}}$ and $A_{\mathtt{out}_1}, \ldots, A_{\mathtt{out}_k}$, respectively, that are to be folded into B. The box $B = \langle A^B_{\mathtt{in}} A_2 \cdots A_{n-k} A^B_{\mathtt{out}_1} \cdots A^B_{\mathtt{out}_k}, \{\mathtt{in} \mapsto 1, \mathtt{out}_1 \mapsto n-k+1, \ldots, \mathtt{out}_k \mapsto n\} \rangle$ arises from F by taking the tuple of tree automata $A_{\mathtt{in}} A_2 \cdots A_{n-k} A_{\mathtt{out}_1} \cdots A_{\mathtt{out}_k}$ and removing selectors that are not in $S_{\mathtt{in}}$ and $S_{\mathtt{out}_1}, \ldots, S_{\mathtt{out}_k}$ from the root transitions of $A_{\mathtt{in}}$ and $A_{\mathtt{out}_1}, \ldots, A_{\mathtt{out}_k}$ to obtain $A^B_{\mathtt{in}}$ and $A^B_{\mathtt{out}_1}, \ldots, A^B_{\mathtt{out}_k}$, respectively (we w.l.o.g. assume that the root states of the TAs do not appear as child states in transitions).

Folding returns the forest automaton $F' = \langle A'_{\mathtt{in}} A'_{\mathtt{out}_1} \cdots A'_{\mathtt{out}_k} A'_1 \cdots A'_m, \sigma' \rangle$ that arises from F as follows. All successors of the roots accepted in $A_{\mathtt{in}}$ and $A_{\mathtt{out}_1}, \ldots, A_{\mathtt{out}_k}$ reachable over selectors from $S_{\mathtt{in}}$ and $S_{\mathtt{out}_1}, \ldots, S_{\mathtt{out}_k}$ are removed in $A'_{\mathtt{in}}$ and $A'_{\mathtt{out}_1}, \ldots, A'_{\mathtt{out}_k}$, respectively (since they are enclosed into B). The root of the trees of $A'_{\mathtt{in}}$ gets an additional edge labelled by B, leading to the reference $\overline{n}$ (the output port), and the components $A_2, \ldots, A_{n-1}$ are removed (since they are also enclosed in B). This operation is denoted as $fold[n, S_{\mathtt{in}}, S_{\mathtt{out}_1}, \ldots, S_{\mathtt{out}_k}, B]$.

Unfolding of Boxes

Unfolding is called as a preprocessing step before operations that implement program statements in order to expose the selectors accessed by the statement. Before an unfolding is performed, the input FA with a k-ary box B is first transformed (using a sequence of cutting, splitting, and swapping) into the form $F' = \langle A'_{\mathtt{in}} A'_{\mathtt{out}_1} \cdots A'_{\mathtt{out}_k} A'_1 \cdots A'_m, \sigma' \rangle$ where the roots of the trees accepted by $A'_{\mathtt{in}}$ have an outgoing B-labeled hyperedge to references $\overline{2}, \ldots, \overline{k+1}$. Furthermore, assume that the box B is of the form $\langle A^B_{\mathtt{in}} A_2 \cdots A_{n-k} A^B_{\mathtt{out}_1} \cdots A^B_{\mathtt{out}_k}, \{\mathtt{in} \mapsto 1, \mathtt{out}_1 \mapsto n-k+1, \ldots, \mathtt{out}_k \mapsto n\} \rangle$ and the ports have outgoing edges with selectors from the sets $S_{\mathtt{in}}$ and $S_{\mathtt{out}_1}, \ldots, S_{\mathtt{out}_k}$ respectively. The operation returns the forest automaton F that arises from F' by inserting components $A^B_{\mathtt{in}} A_2 \cdots A_{n-k} A^B_{\mathtt{out}_1} \cdots A^B_{\mathtt{out}_k}$ in between $A'_{\mathtt{in}}$ and $A'_{\mathtt{out}_1}$, removing the B-transition including its targets, and merging $A^B_{\mathtt{in}}$ with $A'_{\mathtt{in}}$ and $A^B_{\mathtt{out}_1}, \ldots, A^B_{\mathtt{out}_k}$ with $A'_{\mathtt{out}_1}, \ldots, A'_{\mathtt{out}_k}$ respectively. The merging on the TA level consists of merging root transitions of the corresponding TAs. We denote this operation as $unfold[n, S_{\mathtt{in}}, S_{\mathtt{out}_1}, \ldots, S_{\mathtt{out}_k}, B]$.[8]

[8] The parameters $S_{\mathtt{in}}, S_{\mathtt{out}_1}, \ldots, S_{\mathtt{out}_k}$ are used in the backward run to easily invert the operation of unfolding by folding, cf. Section 3.7.1.

Symbolic Execution of Program Statements

We will now discuss our symbolic implementation of the most essential statements of a C-like programming language. We assume that the operations are applied on an FA $F = \langle A_1 \cdots A_n, \sigma \rangle$.

- $\texttt{x} := \texttt{malloc}()$: A new $(n+1)$-th component A_{new} is appended to F s.t. it contains one state and one transition with all selector values set to $\sigma(\texttt{undef})$. The assignment $\sigma(\texttt{x})$ is set to the root reference $\overline{n+1}$.
- $\texttt{x} := \texttt{y->sel}$ and $\texttt{y->sel} := \texttt{x}$: If $\sigma(\texttt{y}) = \sigma(\texttt{undef})$, the operation moves to the error location. Otherwise, by splitting, cutting, and unfolding, F is transformed into the form where $A_{\sigma(\texttt{y})}$ has only one root transition and the transition has a $\texttt{sel}$-successor that is a root reference $\overline{j}$. The statement $\texttt{x} := \texttt{y->sel}$ then changes $\sigma(\texttt{x})$ to $\overline{j}$, and $\texttt{y->sel} := \texttt{x}$ changes the reference $\overline{j}$ in $A_{\sigma(\texttt{y})}$ to $\sigma(\texttt{x})$.
- $\texttt{assume}(\texttt{x} \sim \texttt{y})$ where $\sim \in \{==, !=\}$: This statement tests the equality of $\sigma(\texttt{x})$ and $\sigma(\texttt{y})$ and stops the current branch of the forward run if the result does not match $\sim$.
- $\texttt{assume}(\texttt{x->data} \sim \texttt{y->data})$ where $\sim$ is some data comparison: We start by unfolding and splitting F into the form where $A_{\sigma(\texttt{x})}$ and $A_{\sigma(\texttt{y})}$ have only one root transition with exposed $\texttt{data}$ selector. The data values at the $\texttt{data}$ selectors are then compared and the current branch of the forward run is stopped if the values do not satisfy $\sim$. The operation moves to the error locations if $\sigma(\texttt{x})$ or $\sigma(\texttt{y})$ are equal to $\sigma(\texttt{undef})$.
- $\texttt{free}(\texttt{x})$: First, we cut $A_{\sigma(\texttt{x})}$ at all positions that appear in its root transition, then we remove $A_{\sigma(\texttt{x})}$ from F and set $\sigma(\texttt{x})$ to $\sigma(\texttt{undef})$.

The updates are followed by checking that all components are reachable from program variables in order to detect garbage. If some component is unreachable, the execution either moves to the error location, or—if the analysis is set to ignore memory leaks—removes the unreachable component and continues with the execution.

3.7 Abstraction and Counterexample-based Refinement

In this section, we will describe the complete counterexample-based refinement loop (CEGAR) for forest automata. It is a generalisation of the CEGAR for abstract regular tree model checking of [31, 28].

3.7.1 Backward Run for Counterexample Analysis

Our counterexample trace validation is based on *backward symbolic execution* of a candidate counterexample trace on the level of FAs (with no abstraction on the FAs) while checking *non-emptiness of its intersection* with the forward symbolic execution (which was abstracting the FAs). For that, we have to revert not only abstract transformers corresponding to program statements but also various meta-operations that are used in the forward symbolic execution and that significantly influence the way sets of heap configurations are represented by FAs. In particular, this concerns *folding* and *unfolding* of boxes as well as *splitting*, *merging*, and *reordering* of component TAs, which is used in the forward run for the following two reasons: to prevent the number of component TAs from growing and to obtain a canonic FA representation.

If the above meta-operations were not reverted, we would not only have problems in reverting some program statements but also in intersecting FAs obtained from the forward and backward run. Indeed, the general problem of checking emptiness of intersection of FAs that may use different boxes and different component TAs (i.e., intuitively, different decompositions of the represented heap graphs) is open. When we carefully revert the mentioned operations, it, however, turns out that the FAs obtained in the forward and backward run use *compatible* decompositions (cf. Section 3.5.2) and hierarchical structuring of heap graphs, and so checking emptiness of their intersection is possible. Even then, however, the intersection is not trivial as the boxes obtained in the backward run may represent smaller sets of sub-heaps, and hence we cannot use boxes as symbols and instead have to perform the intersection *recursively* on the boxes as well.

The analysis of spurious counterexamples is further used to refine the abstraction. Particularly, we use a modification of the so-called *predicate language abstraction* on TAs [32], which collapses those states of component TAs that have non-empty intersection with the same predicate languages, which are obtained from the backward run. In case the intersection of the set of configurations of the above described forward and backward symbolic runs is empty, we can derive from it an *automata interpolant* allowing us to get more predicate languages and to refine the abstraction such that progress of the CEGAR loop is guaranteed (in the sense that the same abstract forward run is not repeated).

Inverting Operations in the Backward Run

We now present a description of how we compute the weakest localized preconditions (*inversions* for short) of the operations from Section 3.6 in the backward run. As mentioned in Section 3.6, it is crucial that compatibility with the forward run is preserved. Let $F_i = o(F_{i-1})$ appear in the forward run and $\overline{F}_i$ be an already computed configuration in the backward run s.t. F_i and $\overline{F}_i$ are compatible. We will describe how to compute $\overline{F}_{i-1}$ such that it is also compatible with F_{i-1}.

Inverting most operations is straightforward. The operation $\mathit{cut}[j, k, q]$ is inverted by $\mathit{connect}[k, j, q_k]$ where q_k is the root state of A_k, $\mathit{swap}[j, k]$ is inverted by $\mathit{swap}[k, j]$, and *split* is not inverted, i.e., $\overline{F}_{i-1} = \overline{F}_i$.

One of the more difficult cases is $\mathit{connect}[j, k, q]$. Assume for simplicity that k is the index of the last component of F_{i-1}. Connecting can be inverted by cutting, but prior to that, we need to find *where* the k-th component of $\overline{F}_i$ should be cut. To find the right place for the cut, we will use the fact that the places of connection are marked by the state q in the FA F_i from the forward run. We use the tree automata product $\sqcap$ from Section 3.5, which propagates the information about occurrences of q to $\overline{F}_i$, to compute the product of the k-th component of F_i and the k-th component of $\overline{F}_i$. We replace the k-th component of $\overline{F}_i$ by the product, which results in an intermediate FA $\overline{F}'_i$. The product states with the first component q now mark the places where the forward run connected the components (they were leaves referring to the k-th component). This is where the backward run will cut the components to revert the connecting. Before that, though, we replace the mentioned product states with q by a new state q'. This replacement does not change the language because q was appearing exactly once in every run (because in the forward run, it is the root state of the connected component that does not appear as a child state of any transition), therefore, a product state with q can appear at most once in every run of the product too. Finally, we compute $\overline{F}_{i-1}$ as $\mathit{cut}[k, j, q'](\overline{F}'_i)$.

Folding is inverted by unfolding and *vice versa*. Namely, we invert the operation $\mathit{fold}[n, S_{\mathtt{in}}, S_{\mathtt{out}_1}, \ldots, S_{\mathtt{out}_k}, B]$ by $\mathit{unfold}[n, S_{\mathtt{in}}, S_{\mathtt{out}_1}, \ldots, S_{\mathtt{out}_k}, B]$ and $\mathit{unfold}[n, S_{\mathtt{in}}, S_{\mathtt{out}_1}, \ldots, S_{\mathtt{out}_k}, B]$ by $\mathit{fold}[n, S_{\mathtt{in}}, S_{\mathtt{out}_1}, \ldots, S_{\mathtt{out}_k}, B']$ where the box B' folded in the backward run might be semantically smaller than B (since the backward run is returning with a subset of configurations of the forward run).

Regular abstraction is inverted using the intersection construction from Section 3.5. That is, if o_i is a regular abstraction, then $\overline{F}_{i-1} = \overline{F}_i \sqcap F_{i-1}$.

Finally, inversions of abstract statements use $\overline{F}_i = \langle \bar{A}_1 \cdots \bar{A}_m, \bar{\sigma} \rangle$ and $F_{i-1} = \langle A_1 \cdots A_n, \sigma \rangle$ to compute the FA $\overline{F}_{i-1} = \langle \bar{A}'_1 \cdots \bar{A}'_n, \bar{\sigma}' \rangle$ as follows:

- $\mathtt{x} = \mathtt{malloc}()$: We obtain $\overline{F}_{i-1}$ from $\overline{F}_i$ by removing the j-th TA, for $\bar{\sigma}(\mathtt{x}) = \bar{j}$. The value of $\bar{\sigma}'(\mathtt{x})$ is set to $\sigma(\mathtt{x})$.
- $\mathtt{x} := \mathtt{y{-}{>}sel}$: Inversion is done by setting $\bar{\sigma}'(\mathtt{x})$ to the value of $\sigma(\mathtt{x})$ from F_{i-1}.
- $\mathtt{y{-}{>}sel} := \mathtt{x}$: The target of the $\mathtt{sel}$-labelled edge from the root of $A_{\bar{\sigma}'(\mathtt{y})}$ is set to its target in $A_{\sigma(\mathtt{y})}$.
- $\mathtt{assume}(...)$: Tests do not modify FAs and, since we are returning with a subset of configurations from the forward run, they do not need to be inverted, i.e., $\overline{F}_{i-1} = \overline{F}_i$.
- $\mathtt{free}(\mathtt{x})$: First, the component of F_{i-1} at the index $\sigma(\mathtt{x})$, which was removed in the forward run, is inserted at the same position in $\overline{F}_i$, and $\bar{\sigma}'(\mathtt{x})$ is set to that position. Then we must invert the rewriting of root references pointing to $\sigma(\mathtt{x})$ to $\sigma(\mathtt{undef})$ done by the forward run. For this, we compute the $\sqcap$ forest automata product from Section 3.5 with F_{i-1}, but modified so that instead of discarding reached pairs $(\sigma(\mathtt{undef}), \sigma(\mathtt{x}))$, it replaces them by $\sigma(\mathtt{x})$. Intuitively, the references to $\mathtt{x}$ are still present at F_{i-1}, so their occurrences in the product mark the occurrences of references to $\mathtt{undef}$ that were changed to point to $\mathtt{undef}$ by $\mathtt{free}(\mathtt{x})$. The modified product therefore redirects the marked root references to $\mathtt{undef}$ back to $\mathtt{x}$.

The Role of Compatibility in the Backward Run

Inversions of regular abstraction, component connection, and $\mathtt{free}(\mathtt{x})$, use the TA product construction $\sqcap$ from Section 3.5. The precision of all intersection and product computations in the backward run depends on the compatibility of the backward and forward run. Inverting the program statements also depends on the compatibility of the backward and forward run. Particularly, inversions of $\mathtt{x} := \mathtt{y{-}{>}sel}$ and $\mathtt{y{-}{>}sel} := \mathtt{x}$ use indices of components from F_{i-1}. They, therefore, depend on the property that heaps from $\overline{F}_i$ are decomposed into components in the same way. The compatibility is achieved by inverting every step of folding and unfolding, and every operation of connecting, cutting, and swapping of components.

3.7.2 Regular Abstractions over Forest Automata

Our abstraction over FAs is based on automata abstraction from the framework of *abstract regular tree model checking* (ARTMC) [32]. This framework comes with two abstractions for tree automata, *finite height abstraction* and *predicate language abstraction*. Both of them are based on merging states of a tree automaton that are in the same class of a given equivalence relation. Formally, consider a tree automaton $A = (Q, q_0, \Delta)$ and an equivalence relation $\sim \subseteq Q \times Q$, then an abstraction of A is the TA $\alpha(A) = (Q/{\sim}, [q_0]_\sim, \Delta_\sim)$, such that

$Q/_\sim$ is the set of $\sim$'s equivalence classes, i.e. $Q/_\sim = \{[q] \mid q \in Q\}$, where $[q_0]_\sim$ denotes the equivalence class of q_0, and $\Delta_\sim$ arises from Δ by replacing occurrences of states in transitions by their equivalence classes, i.e., $\Delta_\sim = \{[q] \to \overline{a}([q_1], \ldots, [q_m]) \mid q \to \overline{a}(q_1, \ldots, q_m) \in \Delta\}$. It holds that $|Q/_\sim| \leq |Q|$ and $L(A) \subseteq L(\alpha(A))$.

Finite height abstraction is a function α_h that merges states with languages equivalent up to a given tree height h. Formally, it merges states of A according to the equivalence relation $\sim^h$ defined as follows: $q_1 \sim^h q_2 \Leftrightarrow L^{\leq h}(A, q_1) = L^{\leq h}(A, q_2)$ where $L^{\leq h}(A, q)$ is the language of tree prefixes of trees from $L(A, q)$ up to the height h. *Predicate language abstraction* is a function $\alpha_{[\mathcal{P}]}$ parameterized by a set of predicate languages $\mathcal{P} = \{P_1, \ldots, P_n\}$ represented by tree automata. States are merged according to the equivalence $\sim_\mathcal{P} \subseteq Q \times Q$, such that $\sim_\mathcal{P} = \{(q, q') \in Q \times Q \mid \forall P \in \mathcal{P} : L(A, q) \cap L(P) = \emptyset \Leftrightarrow L(A, q') \cap L(P) = \emptyset\}$. Informally, q and q' are merged if their languages $L(A, q)$ and $L(A, q')$ intersect with the same subset of predicate languages from $\mathcal{P}$.

We extend the abstractions from ARTMC to FAs by applying the abstraction over TAs to the components of the FAs. Formally, let α be a tree automata abstraction. For an FA $F = \langle A_1 \cdots A_n, \sigma \rangle$, we define $\alpha(F) = \langle \alpha(A_1) \cdots \alpha(A_n), \sigma \rangle$. Additionally, in the case of predicate abstraction, which uses automata intersection to annotate states by predicate languages, we use the intersection operator $\sqcap$ from Section 3.5, which descends recursively into boxes, and it is thus more precise from the point of view of the semantics of FAs. More precisely, when a TA A is abstracted, we perform the intersection $\sqcap$ of A and all P from $\mathcal{P}$. An intersection TA $A \sqcap P$ has a state set $Q'_{A \sqcap P}$ consisting of pairs of states from A and P, i.e., $Q'_{A \sqcap P} \subseteq Q_A \times Q_P$. The states q and q' of A are merged iff $\{p \in P \mid P \in \mathcal{P} \wedge (q, p) \in Q_{A \sqcap P}\} = \{p \in P \mid P \in \mathcal{P} \wedge (q', p) \in Q_{A \sqcap P}\}$

Since the abstraction only over-approximates languages of the individual components, it holds that $[\![F]\!] \subseteq [\![\alpha(F)]\!]$ and $\mathit{Repre}(F) \subseteq \mathit{Repre}(\alpha(F))$; therefore, F and $\alpha(F)$ are compatible.

3.7.3 Abstraction Refinement

The finite height abstraction may be refined by simply increasing the height h. Advantages of finite height abstraction include its relative simplicity and the fact that the refinement does not require counterexample analysis. A disadvantage is that the refinement by increasing the height is quite rough. Moreover, the cost of computing in the abstract domain rises quickly with increasing the height of the abstraction, as exponentially more concrete configurations may be explored before the abstraction closes the analysis of a particular branch. The finite height abstraction was used—in a specifically fine-tuned version—in the first versions of Forester [**?**, 66], which successfully verified a number of benchmarks, but the refinement was not sufficiently flexible to prove some more challenging examples.

Predicate abstraction offers the needed additional flexibility. It can be refined by adding new predicates to $\mathcal{P}$ and it gives strong guarantees about excluding counterexamples. In ARTMC, interpolants in the form of tree automata I_i are extracted from spurious counterexamples in the way described in Section 3.7.1. The interpolant is then used to refine the abstraction so that the spurious run is excluded from the program's ART.

The guarantees shown to hold in [32] on the level of TAs are the following. Let A and $I = (Q, q_0, \Delta)$ be two TAs and let $\mathcal{P}(I) = \{L(I, q) \mid q \in Q\}$ denote the set of languages of states of I. Then, if $L(A) \cap L(I) = \emptyset$, it is guaranteed that $L(\alpha_{[\mathcal{P}(I)]}(A)) \cap L(I) = \emptyset$. That is, when the abstraction is refined with languages of all states of I, it will exclude $L(I)$—unless applied on a TA whose language is already intersecting $L(I)$.

We can generalize the result of [32] to forest automata in the following way, implying the progress guarantees of CEGAR described in Section 3.7.1. For a forest automaton $F = \langle A_1 \cdots A_n, \sigma \rangle$, let $\mathcal{P}(F) = \bigcup_{i=1}^{n} \mathcal{P}(A_i)$.

Lemma 2 *Let F and I be FAs s.t. I is compatible with $\alpha^{[\mathcal{P}]}(F)$ and $[\![F]\!] \cap [\![I]\!] = \emptyset$. Then $[\![\alpha^{[\mathcal{P} \cup \mathcal{P}(I)]}(F)]\!] \cap [\![I]\!] = \emptyset$.*

We note that the lemma still holds if $\mathcal{P}(I)$ is replaced by $\mathcal{P}(A_i)$ only where A_i is the i-th component of I and $L(A_i \sqcap A'_i) = \emptyset$ for the i-th component A'_i of $\alpha^{[\mathcal{P}]}(F)$.

3.8 Automatic Discovery of Boxes

In this section, we discuss in a more detail *box folding* from Section 3.6; mainly, we focus on how boxes are discovered (synthesised) automatically.

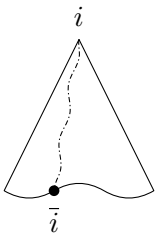
Originally, our verification approach, as published in [**?**], relied on the following two facts:

1. the user is able to provide a set of nested boxes that is sufficient for the verification of the given program and
2. it is enough to look for instances of the provided boxes to be folded at roots existing in the FA in which the folding should occur.

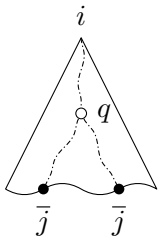
In our experiments, however, it turned out that constructing a set of boxes suitable for verification of a given program is quite inconvenient. In particular, constructing boxes requires a non-trivial insight into the program's semantics and a considerable amount of manual effort of an experienced user. It is not possible to reuse boxes even for quite common sub-heap patterns—such as doubly-linked list segments—in cases where the structures are connected via different selectors (because we are working at the level of `gcc` intermediate code, the issue is not with selector names but rather with their offsets within the data structures). Finally, in certain cases, the algorithm could at some point choose between two actions: either eliminate an existing root by concatenating two components together or perform a folding of some box at this root. If the concatenation is performed first (during normalization), it could happen that the box (whose folding is crucial for eliminating other cut-points that could not be eliminated in any other way) cannot be folded anymore because the required root disappeared, and, as a consequence, the analysis does not terminate. To address these issues, we have developed a fully automatic approach that is able to automatically find a suitable set of boxes and, moreover, does not rely on folding at existing roots only.

Recall that boxes were introduced in order to bring a possibility of hiding certain cut-points that appear repeatedly within the heap. The way boxes look like heavily depends on the shape of the data structures being handled, but the general principle is to hide into boxes some repeating (possibly hierarchical) heap patterns with cut-points.

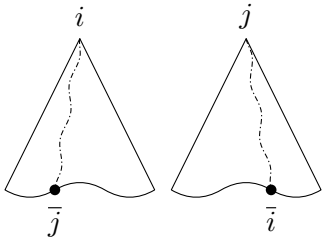
In this text, we follow the procedure described in [107], which is the one used in the SV-COMP version of Forester. The procedure can handle most usual data structures, such as (doubly) linked and nested or cyclic lists, trees with additional pointers, and even skip lists. We note that a more complete and complex solution (theoretically capable of handling a larger class of shape graphs) was presented in [66]. In [107], we concentrate directly on dealing with the growing number of cut-points. That is, we do not ask "Which repeated patterns does one need to fold?", but rather "Which cut-points need to be eliminated?".



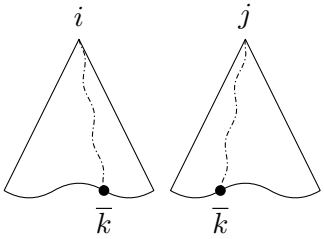
(a) Type 1 cut-point i (reference within the same component)



(b) Type 2 cut-point j (a single component contains multiple references to j)



(c) Type 3 cut-points i and j (several mutually linked components)



(d) Type 4 cut-point k (referenced from several other components)

Figure 3.4: Four possible types of cut-points.

In the rest of this section, we show how we can automatically eliminate certain kinds of cut-points using box folding.

3.8.1 Cut-point Types

We are now going to describe four types of cut-points (illustrated in Fig. 3.4), each of which is eliminated by a special algorithm. We distinguish whether a node is a cut-point due to a cycle in the heap or whether it is only referenced multiple times, and we also distinguish whether this can be seen locally within a single component or only across several different components. This gives us four different types of cut-points. One cut-point can be of multiple types.

In particular, Type 1 cut-points arise, for instance, when one works with cyclic lists. In such a case, a single TA component encodes a set of cyclic lists with the cycle encoded using a leaf that refers back to the root of the component. Type 2 cut-points are those cut-points that are referenced multiple times from within a single component. These typically arise when one deals with trees whose all leaves refer to some designated node (e.g., the root). Next, a set of two or more cut-points is said to consist of cut-points of Type 3 if the components rooted at them are linked in a cycle. A typical scenario where such cut-points appear is working with doubly-linked lists, where the cycle appears between a pair of successive list nodes. Observe that each inner element of the lists is pointed from its predecessor and from its successor at the same time, and so (without using a hierarchical encoding) every element has to reside in its own component (and the present loops cannot be reduced to loops within a single component with a cut-point of Type 1). Finally, Type 4 cut-points are the cut-points referred from two or more components at the same time.

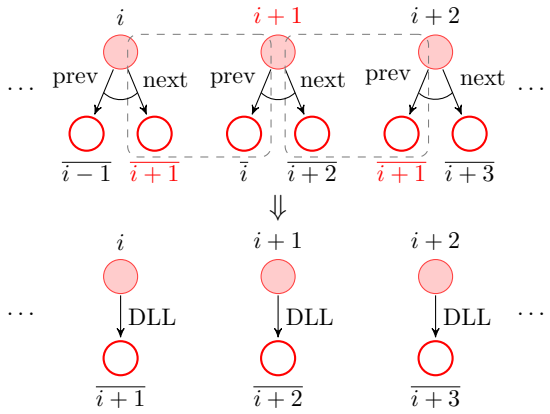


Figure 3.5: Folding of a doubly-linked list with a Type 4 cut-point $i + 1$

In the following section, we will show how to eliminate cut-points of Type 1 and 2 as well as some cut-points of Type 3. In the case of Type 3 cut-points, we only show how to deal with pairs of cut-points of Type 3 that are the roots of neighboring components linked in a cyclic way. We do not consider any other form of Type 3 cut-points since we have not encountered them in any of our case studies (in particular, we have not encountered loops going through more than two cut-points). We also do not have any direct elimination procedure for cut-points of Type 4. This was not needed in our case studies either, because no matter how ubiquitous cut-points of Type 4 were, they were always of other types too, and eliminating cut-points of Type 1, 2, and 3 either eliminated Type 4 too or turned Type 4 cut-points to one of the first three types. For instance, in the case of doubly-linked lists, each node in fact corresponds to a cut-point of Type 3 and Type 4 at the same time. Nevertheless, when we fold the parts of the heap that make the concerned nodes to be cut-points of Type 3, there will not remain any cut-points of Type 4 either (see Fig. 3.5 for an illustration).

3.8.2 Cut-point Elimination

In this section, we discuss the automatic elimination of cut-points of Type 1, 2, and 3, discussed in the previous section. As we have already mentioned, it might happen that some cut-point is of more than one type, which is solved by several applications of the folding algorithm. The heuristic, which currently seems to give the best results in our experiments, eliminates cut-points in the order 3, 2, and 1.

Before we explain how to deal with the particular types of cut-points, we introduce the so-called *component slicing*. On the level of a single forest $\langle t_1, \ldots, t_n, \sigma \rangle$, folding is an operation that (1) selects the part of the forest to be folded into the new box, in the form of one root component t_i, a set of selectors S, and possibly several additional components, and (2) *slices* the component t_i into two parts, the *kernel* and the *residue*. The kernel is the root and its sub-trees connected to it by the selectors of S. The residue is the root and its sub-trees connected to it by the remaining selectors outside S. The box will then hide the kernel and the eventual additional components. The original forest is then represented by its modification in which (i) t_i is replaced by a component created from the residue by adding an additional hyperedge leading from the root labeled by the new box, and (ii) the additional components are removed.

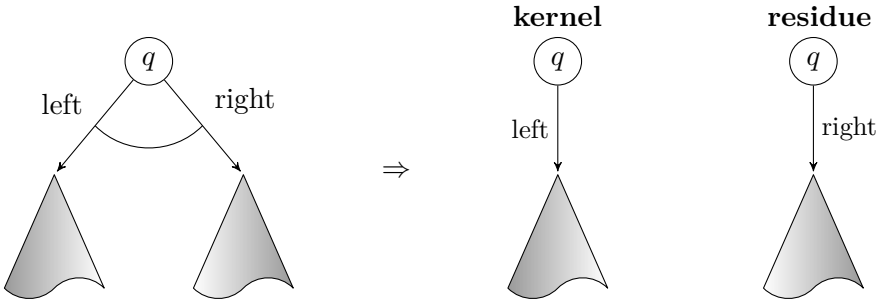


Figure 3.6: An example of component slicing. The left part shows the original automaton that is to be sliced at the state q according to the edge labeled by "left" (i.e., we perform $q \to \langle \mathit{left}, \mathit{right} \rangle(q_1, q_2) \rhd \{\mathit{left}\}$). The right part shows the result after slicing, in which the kernel contains the structure reachable via "left", and the residue contains the rest of the original automaton (in this case, the structure reachable via "right").

We will now discuss how slicing is implemented on the level of forest automata. Let us recall that an FA is a tuple $F = \langle A_i \cdots A_n, \sigma \rangle$ where the TAs A_i, for $1 \leq i \leq n$, have transitions of the form $q \to a_1 \cdots a_n(q_1, \ldots, q_n)$. Such transition generates a tree node v connected to nodes $v_1, \ldots, v_n$ via edges (n, a_i, n_i) for each $i : 1 \leq i \leq n$. The symbols a_i are either selectors or indexed boxes of the form $B_{(k)}$, where B is a box and $1 \leq k \leq \sharp B$, where the symbols $B_{(1)}, \ldots, B_{(\sharp B)}$ always appear together and represent a hyperedge labeled by B. We use additional technical assumptions on the form of FAs that simplify the operation of slicing; in the symbolic execution, we always keep FAs in the form. Namely, (1) root states of tree automata components never appear as child states of transitions, (2) for every state, all transitions with the state as the parent state have the same vector of symbols (both can be achieved by simple automata transformations). Slicing will be further specified using the operator $\rhd$ called *transition cut* that, when given a transition $t = q \to a_1 \cdots a_n(q_1, \ldots, q_n)$ and a set of selectors and indexed boxes E, produces a new transition $t \rhd E$ that is made from t by, for all $a_i \in E$, erasing a_i from the sequence of symbols $a_1 \cdots a_n$ and also erasing q_i from the sequence of states $q_1, \ldots, q_n$. Slicing of the i-th component A_i of F w.r.t. a set of selectors or indexed boxes E then produces two versions of A_i, again called the kernel and the residue. The residue has each accepting transition $t = q \to a_1 \cdots a_m(q_1, \ldots, q_m)$ of A_i replaced by $t \rhd E$, while the residue has the accepting transition replaced by $t \rhd \{a_1, \ldots, a_k\} \setminus E$ (see Fig. 3.6 for an illustration of slicing).

Type 1 Cut-point Elimination

A Type 1 cut-point (cf. Fig. 3.4a) corresponds to a component that contains references to its own root (these are called *self-references* in the following). To eliminate such a cut-point, we want to "hide" all the self-references into a box. Therefore, we identify a set E of all selectors (or boxes) within the accepting transitions that lie on some path going to some self-reference. Then, we perform slicing of the component w.r.t. the set E to obtain the kernel containing all self-references and a residue containing the rest. Suppose that the root references appearing in the kernel form the set $R \subseteq \{1, \ldots, n\}$; the rank of the new box will be $k = |R| - 1$.

Next, we perform a depth-first traversal on the kernel and we rename all references from R according to the order in which they are visited (such that the self-references are

always relabeled to 0) to obtain a mapping $f : R \to \{0, \ldots, k\}$. The relabeled kernel is transformed into a new box B with $\sharp B = k$. All accepting transitions $q \to \overline{a}(q_1, \ldots, q_n)$ of the residue are modified to

$$q \to \overline{a} B_{(1)} \cdots B_{(k)}(q_1, \ldots, q_n, r_1, \ldots, r_k)$$

where r_j is a state representing a reference to a root u such that $f(u) = j$ (i.e., the additional child states encode the mapping f and, as a result, also the correspondence between the root references appearing inside the box and the root references appearing on the level on which the box is used).[9] As the last step, we replace the original component by the modified residue.

Note that the process of folding can easily be reversed whenever needed. First, we extract the mapping f. Then, we relabel the root references inside the box using f^{-1}, and we replace the given box in some transition t by the relabeled component.

Type 2 Cut-point Elimination

A Type 2 cut-point arises when the reference to a node appears multiple times within a single component (cf. Fig. 3.4b). In this case, we will fold into a box the smallest sub-tree of the component that contains all references to the cut-point. The box will then allow us to reduce the number of references to the given cut-point to one. We first identify a state q that accepts sub-trees containing all root references to the given cut-point such that none of q's child states have this property (see Fig. 3.4b). If there are more such states, the folding is performed separately for each of them. We then cut the component into two at the state q (the operation of cutting of Sec. 3.6). Then, in the new component rooted by q (the tree suffix component created by the cut), we identify the set of selectors and indexed boxes E that start paths from the root (corresponding to state q) to references to the cut-point we wish to eliminate. Then, we perform slicing of the component parametrized by E. The kernel of the slicing is transformed into a box B, which is then appended to the residue in the same manner as in the case of cut-points of Type 1.

Type 3 Cut-point Elimination

To eliminate a pair of Type 3 cut-points i and j (i.e., the cut-points mutually referencing each other—see Fig. 3.4c), we need to use a box encoding a cycle starting at cut-point i going through cut-point j and, finally, back to i (this corresponds, e.g., to a pair of forward and backward links in a doubly-linked list). Intuitively, we create a box composed of two components (unlike cut-points of Type 1 and 2, which have only a single component). The first component of the box will be made from the part of the i-th component that contains all references to j. Similarly, the second component of the box will be made from the part of the j-th that contains all references to i. As a result, the box will represent a sub-graph containing the loop between the roots of the original i-th and j-th component.

Let us describe the elimination of Type 3 cut-point in a greater detail. First, we perform slicing of components i and j, which mutually refer each other, such that the kernels obtained in the slicing contain all paths from i to j (resp. from j to i). The two kernels are transformed into a box B. The box B is then appended to the accepting transitions of

[9] Here, for the sake of simplicity, we ignore the fact that the selectors and the boxes are ordered, hence B should not be put simply behind $\overline{a}$. The sequence of symbols on the rule should be then ordered, and the sequence of states on the right-hand-side should be permuted accordingly.

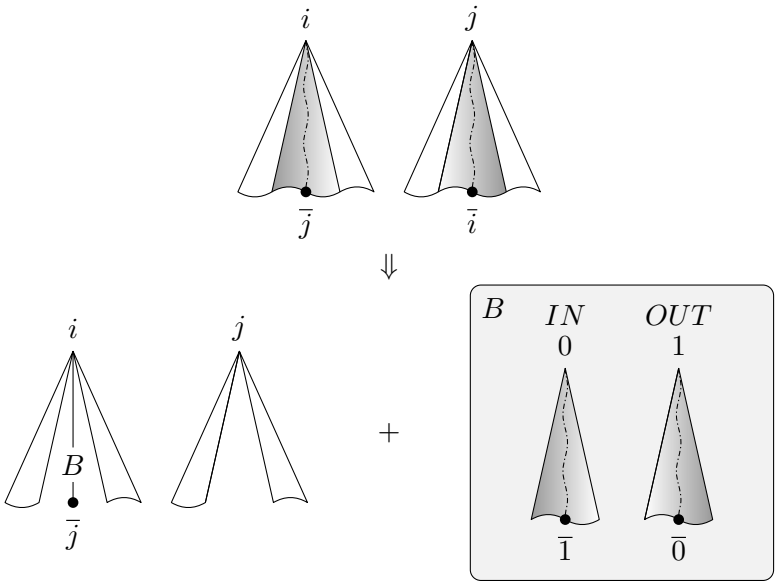


Figure 3.7: An illustration of elimination of Type 3 cut-points. First, the two components are sliced at their accepting states. Next, the kernels of the result containing references to i and j are transformed into a new box B, which is added to the remaining part of the component i. Finally, the modified component j contains no references to i.

the residue of the component i in the same way as in the previous cases. As the last step, we replace (i) the component i within the original FA by the modified residue and (ii) the component j by the residue of component j. As a result, the new component i contains a single reference to j, and the new component j contains no reference to i—see Fig. 3.7.

Note that we could also perform the symmetrical transformation in which the newly created box is added into the component j. This would yield a different structure with the same semantics. In order to decide which variant to choose in practice, we use an ordering on the set of components of the original FA, i.e., if $i < j$, we append the newly created box to the residue of i.

3.8.3 From Nested FAs to Alphabet Symbols

When performing certain automata operations, such as language inclusion, we treat all symbols as not having any underlying structure at all. To make this work, we have to ensure that the nested FAs (boxes) with the same semantics are represented using the same alphabet symbol. This is achieved by maintaining a database that maps boxes to alphabet symbols. Every time a new box is created, it is first compared to existing boxes having the same root interconnection graph (cf. Section 3.3.3) (which also implies the same number of components). The comparison of two boxes is done via checking language equality of their underlying FAs[10][11]. If the same box already exists, the symbol obtained from the database

[10] Here, we are not dealing with sets of FAs, hence the comparison of two FAs can be done component-wise.

[11] The language equality check is performed as two inclusion queries. The inclusion test itself is again only a safe approximation, since we ignore the structure of nested boxes that can appear within the box we have just created. Therefore, it can be the case that two boxes with the same semantics are represented by different alphabet symbols. According to our experiments, such approximation, however, works well in practice.

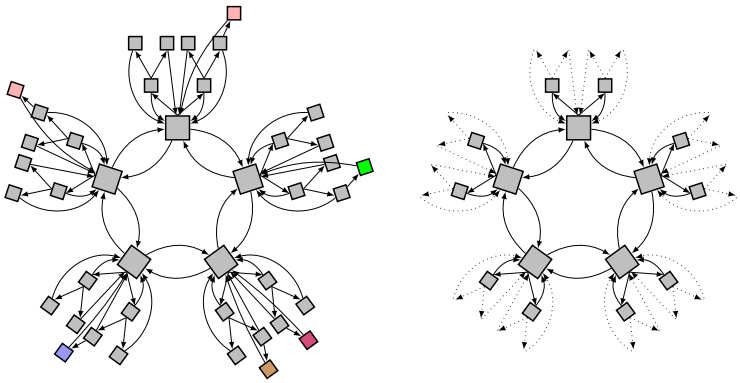


Figure 3.8: On the left-handed side is a data structure allocated on heap by a program. On the right-handed side is the same structure after the first abstraction.

is used for its representation. Otherwise, a new alphabet symbol is created and stored in the database.

Furthermore, we can also consider only language inclusion to relax the requirement of equality when testing whether the two boxes match. This means that during the folding the given box can be potentially replaced by a "bigger" one that already exists in the database. As discussed later, this can in some cases drastically decrease the number of boxes that are required, and thus substantially increase the performance.

3.9 Running Example

Since theory of forest automata and its application to shape analysis may be too abstract we provide with a running example in the following section. We are going to show computation of fixpoint in an abstraction loop where abstraction and folding are performed alternately until a fixpoint is reached. The presented example should illustrate the main principles and give you an intuition, therefore it is imprecise in some details. E.g., for a sake of clarity we show the main principle on a concrete heap graph instead of forest automata where one would need to visualise the sets of heap graphs represented by automaton.

Consider a data structure allocated on a heap shown on the left-handed side of Figure 3.8. It is a doubly-linked list which nodes are also the roots of trees with the root pointers. As the first step, all solo boxes are unfolded before the abstraction loop is executed. In the figure, this is illustrated by colored nodes in the figure which are unfolded from the solo boxes.

After unfolding the solo boxes, the first abstraction is performed which result is shown on the right-handed side of Figure 3.8. The dotted lines represent a loop in corresponding automaton created by abstraction. In our example, we use height abstraction with height two. It mergers all states with height higher than two to to one leave node with the root pointer. In the figure, we illustrate it be removing the abstracted nodes but preserving the root pointers to illustrate the created cycle while keeping the figure understandable. The resulting automaton now represents doubly-linked list which nodes are roots of trees of arbitrary depth. We note again that the example does not reproduce behaviour of algorithm

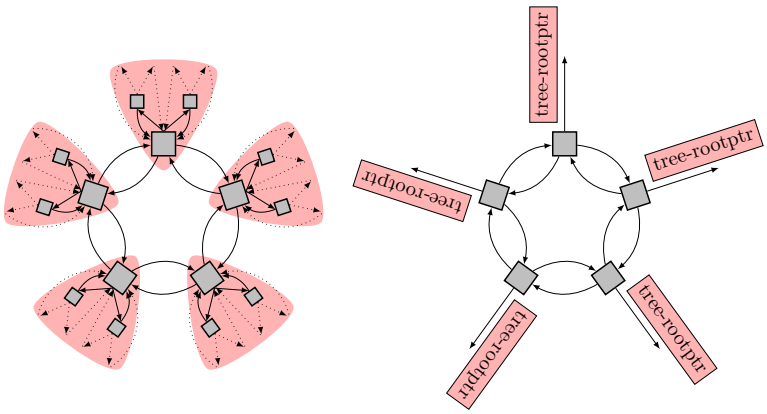


Figure 3.9: On the left-handed side is the studied data structure with the highlighted parts which are going to be folded to a box. On the right-handed side the highlighted parts of graph are folded and replaced by box *tree-rootptr*.

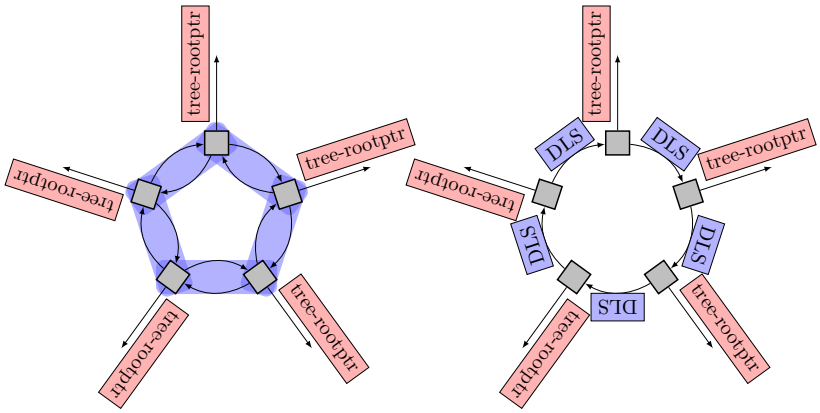


Figure 3.10: On the left-handed side is the data structure before another folding where blue areas are going to be folded. On the right-handed side is the resulting graph with the highlighted parts folded in the box *DLS*.

precisely but it should bring an intuition how abstraction works, i.e., creates a loop in the automaton.

Then the folding operation is performed once the first abstraction is finished. The folding algorithm finds suitable parts of heap graph to be folded. These parts of graph are trees pointed by the nodes of doubly-linked list as it is shown in Figure 3.9. They are folded to box called *tree-rootprt* which is added to edges leading from the nodes of doubly-linked list as shown on the right-handed side of Figure 3.9.

The second abstraction is performed, however the abstraction algorithm finds nothing what could be abstracted. The second folding follows. It finds the suitable sub-graphs for folding which are highlighted by blue color on the left-handed side of Figure 3.10. These sub-graphs are segments of doubly-linked list, i.e., part of graph between two DLL nodes. It folds the sub-graphs to the boxes called *DLS*. The resulting graph after folding is shown on the right-handed side of Figure 3.10.

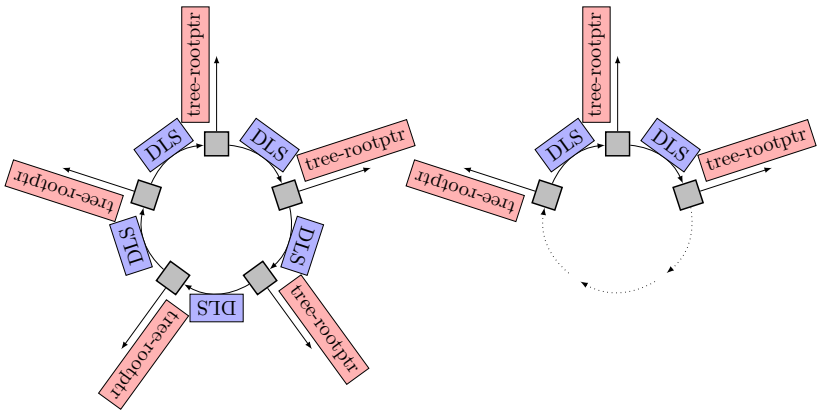


Figure 3.11: On the left-handed side is the data structure before the last abstraction. On the right-handed side is the graph after abstraction.

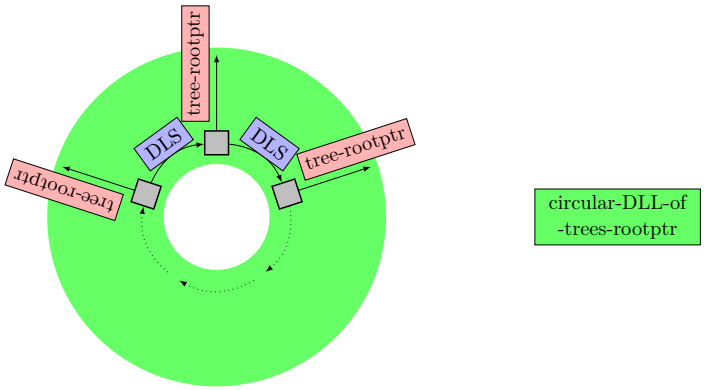


Figure 3.12: On the left-handed side there is the data structure before the last folding. The highlighted green part is going to be folded. On the right-handed side is the graph with the folded highlighted part to box *circular-DLL-of-trees-rootptr*. Now, the fixpoint is reached.

Now, another abstraction is performed and merges its nodes in such way that the automaton now represents arbitrary doubly-linked list of length at least two (which is a consequence of fact that we used again height abstraction with height 2). The abstraction is shown on Figure 3.11.

Finally, the last folding is performed. The whole heap graph could be folded to one box as shown by the left-handed side of Figure 3.12. As the result, we obtain the box *circular-DLL-of-trees-rootptr* shown on the right-handed side of Figure 3.12. The fixpoint is reached in this step and verification procedure proceed with an automaton containing just one transition with the mentioned box.

3.10 Architecture of Forester

Forester is an open source tool (written in C++) for verification of programs manipulating complex dynamic data structures. Currently, it supports analysis of programs written in

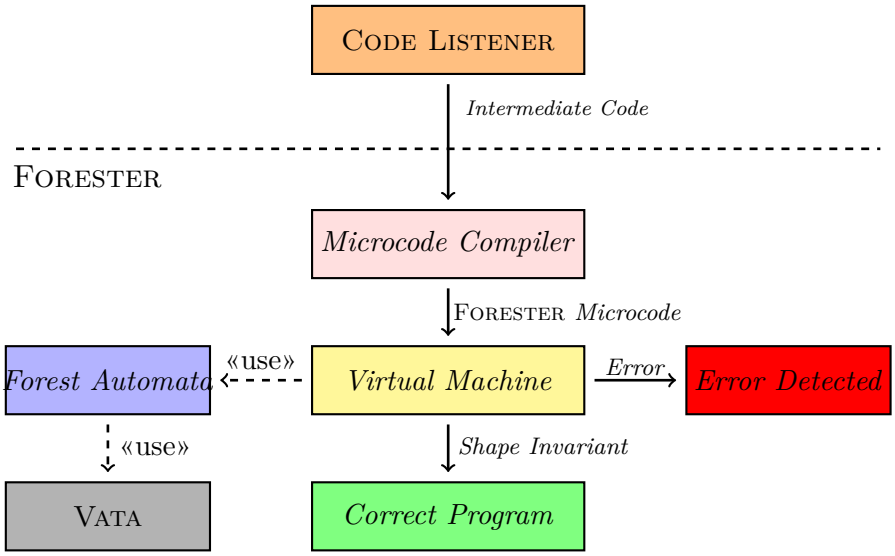


Figure 3.13: High-level overview of Forester

the C language. Forester was designed as a `gcc` plugin under the GPLv3 license and can be obtained from its website [55].

3.10.1 Design

Although Forester is implemented as a `gcc` plugin, it does not directly analyze `gcc`'s low-level intermediate code, called GIMPLE. Instead, it uses the Code Listener infrastructure [47], which provides a higher-level interface over GIMPLE.

Let us describe the architecture of Forester (see Fig. 3.13). Forester starts the analysis by using Code Listener (providing an interface to `gcc`) to obtain the intermediate code of the analysed program. This intermediate code is in the second step compiled, using Forester's *Compiler* module, into its representation in the Forester *Microcode* (representing abstract transformers), which can be interpreted by the Virtual Machine. We give more details about the microcode in Section 3.10.2.

The Virtual Machine is an abstract execution engine whose configuration contains a forest automaton and a register set. The registers are used for auxiliary operations, e.g., whenever a memory location is to be modified, it is first (after the necessary splitting and cutting performed on the FA) loaded into a register, changed there, and then stored back into the FA. The registers can contain an arbitrary symbolic value, which can be, e.g., a data value, a reference to a root in the FA (representing a symbolic pointer), or contents of a memory node (i.e., a `struct` data type). The Virtual Machine performs the symbolic execution over the FAs (represented by the module *Forest Automata*) as described in Section 3.4, including automatic abstraction refinement and box discovery. The analysis can conclude either by finding an error or by inferring safe shape invariants (represented by FAs) for each line of the program.

Efficient handling of FAs needs fast algorithms for performing operations and tests on tree automata, especially for operations with high worst-case complexity, such as the FA entailment test, which is reduced to the ExpTime-complete TA language inclusion check. Therefore, Forester uses the efficient TA algorithms from the Vata library [81].

3.10.2 Forester Microcode

In this section, we provide an example of Forester microcode. Consider the following data structure implementing a singly-linked list:

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

The syntax of a microinstruction is *op* r_0, r_1, r_2 where *op* is the name of an instruction and r_0, r_1, and r_2 are operands such that r_0 is the destination register and r_1 and r_2 are source registers or constants. A microinstruction can have zero to three operands. The notation $[r + c]$ is used to access the selector with displacement c in a label under the root referenced by register r.

Suppose that the verified program contains the following statement:

```
x = malloc(sizeof(struct T));
```

The statement is translated into the following sequence of microinstructions:

```
1: mov_reg      r0, (int)8
2: alloc        r1, r0
3: node_create  r2, r1, <next,data>[0:4:+0,4:4:+0]
4: mov_reg      r3, ABP + 0
5: mov_reg      [r3 + 12], r2
6: check
```

First, on line 1, the size of the newly allocated data is stored into register r_0 (it is the size of `struct T`, which is in this case 8). Second, on line 2, a new symbolic memory location of the size given by r_0 is created and its symbolic address stored into r_1. On line 3, we create a new TA t with a single transition representing a freshly created instance of the type `struct T` at the memory location give by r_1 and append it to the FA representing the symbolic state. The reference to the TA is stored into r_2. The type of the instance is given by `<next,data>[0:4:+0,4:4:+0]`, which gives names of the selectors and for each of them their offset (location in the memory cell), size (we assumed a 32-bit system, which uses 4 B for a pointer, and 4 B `int`), and displacement (it is valid only for pointers and denotes offset at which the pointer points into the referenced memory location[12]). On line 4, a reference to the TA representing the current function's stack frame (ABP) is loaded into register r_3. (Note that in Forester, the program stack is represented using memory locations on the heap.) Finally, on line 5, the root reference to the TA t stored in register r_2 is added to the selector with displacement 12 (corresponding to the displacement of the local variable `x` in the current stack frame) in the TA referenced by register r_3. The instruction `check` on line 6 checks that the heap is garbage free.

3.11 Tutorial

Forester is can be obtained from its webpage [55], and the easiest way to run it is from the virtual machine [56]. Once you boot the virtual machine (Ubuntu 16) you will find

[12] For instance, linked lists used within the Linux kernel do not point at the beginning of the linked data structure, but, instead, into its middle (the position of the *list header*).

the `forester` folder on the desktop. The folder contains compiled binaries (the `fa_build` folder) of Forester and also its source code (the `fa` folder). For running the tool please read `README`. The instruction for compilation of the tool are in the `INSTALL` file.

3.11.1 Running Forester with BenchExec

The BenchExec framework is a platform for evaluation of tools over a benchmark. Since it is used for evaluation of the SV-COMP competition it contains modules necessary for running Forester. If you want to use BenchExec, you still need to compile Forester and provide the `include` directory, the files `libfa.so`, and `sv_comp_run.py` from the directory `fa_build`. Please check the GitHub repository of BenchExec[13] for further information about how to run a tool using the framework.

3.12 Experiments

The following section summarizes different experiments performed to evaluate the abilities of Forester. The benchmark set consists of the tasks from [66] and [65]. We selected the programs in order to illustrate the two crucial features of Forester: the ability to learn boxes automatically and counterexample analysis with abstraction refinement. We also compare with Predator [46, 48], a many-times winner of heap-manipulation-related categories of SV-COMP.

The benchmarks are C programs manipulating singly and doubly-linked list, trees, skip lists, and their combinations. We were able to analyse all of them fully automatically without any need to supply manually crafted predicates or any other manual aid. The test cases are described in detail in Section 3.12.1.

We present our experimental results in Table 3.1. The table gives for each test case its name, the information whether the program is safe or contains an error, the number of lines of code, the time needed for the analysis, the number of refinements, the number of predicates learnt during the abstraction refinement, and, finally, the time that Predator needed to analyse it (*err* denotes a wrong answer and ∞ denotes timeout, which was set to 900 s, the same as in SV-COMP).

Some of the test cases consider dynamic data structures without any data stored in them, while some of them consider data structures storing finite-domain data. Such data can be a part of the data structure itself, as, e.g., in red-black trees; they can arise from some finite data abstraction; or they are also sometimes used to mark some selected nodes of the data structure when checking the way the data structure is changed by a given algorithm (e.g., one can check whether an arbitrarily chosen successive pair of nodes of a list marked red and green is swapped when the list is reversed—see e.g. [32]).

As the results show, some of our test cases do not need refinement. This is because the predicate abstraction (even with the empty set of predicate languages) is *a priori* restricted in order to preserve the forest automata root interconnection graph (cf. Section 3.3.3), which roughly corresponds to the reachability relation among variables and cut-points in the heaps represented by a forest automaton. This restriction was already used with the finite height abstraction in the versions of Forester from [**?**, 66].

Table 3.1 also provides a comparison of the version of Forester from [66] with the version from [65]. In particular, the highlighted cases are not manageable by the version

[13] https://github.com/sosy-lab/benchexec/

from [66]. These cases can be split into two classes. In the first class, there are safe programs where the initial abstraction is too coarse and introduces spurious counterexamples, and the abstraction thus needs to be refined. The other class consists of programs containing a real error (which could not be confirmed without the backward run). The times needed for analysis are comparable in both versions of Forester.

To illustrate a typical learnt predicate, let us consider the test case *GBSLL*. This program manipulates a list with nodes storing two data values, green and blue, for which it holds that a green node is always followed by a blue one (this data structure is a simplified version of a red-black tree). The program also contains a tester code to test this property. Forester first learns two predicate languages describing particular violations of the property: (1) "a green node is at the end of the list" and (2) "there are two green nodes in a row." After that, Forester derives a general predicate representing all lists with the needed invariant, i.e, every green node is followed by a blue one. The program is then successfully verified.

Another example comes from the analysis of the program *TreeOfCSLL*, which creates and deletes a tree where every tree node is also the head of a circular singly-linked list. The program contains an undefined pointer dereference error in the deletion of the circular lists. Forester first finds a spurious error (also an undefined pointer dereference) in the code that creates the circular lists. In particular, the abstraction introduces a case in which a tree node that is also the head of a list needs not be allocated, and an attempt of accessing its next selector causes an undefined pointer dereference error. This situation is excluded by the first refinement, after which the error within the list deletion is correctly reported. Notice that, in this case, the refinement learns a property of the shape, not a property over the stored data values. The ability to learn shape as well as data properties (as well as properties relating shape with data) using a uniform mechanism is one of the features of our method that distinguishes it from most of the related work.

3.12.1 Description of Benchmarks

In this section, we describe the test cases used above in our experimental evaluation. Note that we use a limited set of integer values since we do not support integer abstraction. We use "SLL" and "DLL" to denote singly and doubly-linked lists respectively.

The cases described in the following list satisfy some (regular-expressible) safe invariant (if their status in the table is "safe") or contain an error (if their status is "error"). As an error, we consider violation of memory safety properties, i.e., absence of `null`/undefined pointer dereference, invalid free, and presence of garbage. For some data structures, we have both a safe case and an error case.

- *(SLL/DLL) (operation)*: Performing *operation* on a SLL/DLL.
- *CDLL*: Construction of a circular DLL.
- *SLL+head*: Construction of a list where each element points to the head of the list.
- *SLL-Linux*: Implementation of lists used in the Linux kernel, which uses type casts and restricted pointer arithmetic.
- *DLL+subdata*: A DLL implementation that uses data pointers pointing either inside the list nodes or optionally outside of them.
- *SLLOf(CSLL/2CDLL)*: An SLL of circular SLLs or two circular DLLs respectively.

- *SkipList$_i$*: Construction and traversal of a skip list of level i.
- *(SLL/DLL)01*: The nodes of the list may or may not point to an external node, which, if present, is unique for each list item. We check the invariant that each node has a pointer set to `null` or to an address of an external node.
- *C(SLL/DLL)Mon*: A circular SLL/DLL consisting of nodes with integer values. The head of the list has the reserved value 0. The rest of the nodes with their integer values form a non-decreasing sequence. We verify that the successor of an arbitrary node can have a smaller value than the node only when the successor is the head of the list.
- *OptPtr(SLL/DLL)*: Each node of the list has an integer value, a pointer to the next node, and an optional pointer to an external node. When constructing the list, an integer value of every node is chosen nondeterministically. When the integer value 0 or 1 is chosen, the optional pointer points to the node itself. On the other hand, when 2 is chosen, a new external node is allocated and its address is assigned to the optional pointer. We verify the relation of integer values and optional pointers for all nodes of the list.
- *Queue(SLL/DLL)*: We create a list with nodes containing the integers 0, 1, 2, and 3. The list can form sequences 0, 01, 012, 0123*. A particular sequence is created during construction of the list nondeterministically. We remember which sequence was actually created by an auxiliary integer variable. Then we traverse the list and check that the sequence formed by the list corresponds to the value of the auxiliary variable.
- *GB(SLL/DLL)*: We create a list containing green and blue nodes. The colors arbitrarily alternate but it holds that a green node is always followed by a blue node.
- *GB(SLL/DLL)Sent*: This case is similar to the previous one but instead of terminating the list with the `null` value, the list is terminated using a dedicated sentinel node.
- *RG(SLL/DLL)*: The list contains an arbitrary prefix of white nodes, one red node followed by a green one, and an arbitrary suffix of white nodes. The list is reversed and it is checked whether the green node is followed by the red node.
- *WB(SLL/DLL)*: Exactly one blue node is inserted into a list of white nodes of an arbitrary length. Then the list is traversed and it is checked that the number of blue nodes is one.
- *Sorted(SLL/DLL)*: This test case contains a sorted list of nodes with integer values 0 and 1. A node with value 1 is added at an arbitrary position that keeps the order of values of nodes in the list. Finally, it is checked that the values of nodes in the list are still ordered.
- *End(SLL/DLL)*: The last element of the list has a special integer value.
- *TreeRB*: We construct a red black tree and then go through the tree checking (regular) invariants of this data structure. The created tree has an arbitrary height, and the nodes may have both, one, or no child allocated. A transposition of nodes needed

to preserve the data structure's invariants is done continuously during construction of the tree when a new node is added. This operation is complex since it requires relocation of nodes in several levels of trees. The nodes also need to have parent pointers to allow the transposition.

- *TreeWB*: We construct a tree that has all nodes white except exactly one blue node. The blue node is at an arbitrary position. We traverse the tree and check that there is a single blue node.

The following test cases from our benchmark are checked only for memory safety properties.

- *TreeCnstr*: The construction of an arbitrary binary tree.
- *TreeOfCSLL*: We construct a binary tree where each node points to a circular SLL of an arbitrary length. Then we traverse the whole tree and all nested lists.
- *TreeStack*: The construction of a binary tree, which is subsequently destroyed using stack implemented by an SLL.
- *TreeDsw*: We construct a binary tree and perform the Deutsch-Schorr-Waite traversal.
- *TreeRootPtr*: The construction and traversal of a binary tree with nodes containing root pointers.

Table 3.1: Results of experiments

Program	Status	LoC	Time [s]	Refnm	Preds	Predator [s]
SLL (reverse)	safe	32	0.03	0	0	0.03
SLL (delete)	safe	33	0.02	0	0	0.03
SLL (insertsort)	safe	36	0.04	0	0	0.03
SLL (bubblesort)	safe	42	0.02	0	0	0.2
SLL (mergesort)	safe	76	0.08	0	0	0.10
DLL (reverse)	safe	39	0.70	0	0	0.03
DLL (insert)	safe	39	0.56	0	0	0.05
DLL (insertsort1)	safe	47	0.40	0	0	0.11
DLL (insertsort2)	safe	48	0.12	0	0	0.05
CDLL	safe	32	0.02	0	0	0.03
SLL+head	safe	33	0.05	0	0	0.03
SLL-Linux	safe	60	0.03	0	0	0.03
DLL+subdata	safe	40	0.12	0	0	∞
SLLOfCSLL	safe	47	0.02	0	0	0.05
SLLOf2CDLL-Linux	safe	35	0.16	0	0	0.26
SkipList_2	safe	84	3.36	0	0	∞
SkipList_3	safe	92	20.05	0	0	∞
SkipList_2	error	84	0.08	1	1	0.10
SLL01	safe	70	0.90	1	1	*err*
DLL01	safe	73	0.54	2	2	2.2
CSLLMon	safe	49	2.1	3	3	2.3
CDLLMon	safe	52	31.0	18	24	2.3
OptPtrSLL	safe	59	0.94	3	3	∞
OptPtrDLL	safe	62	1.3	5	5	∞
QueueSLL	safe	71	17.00	10	10	2.2
QueueDLL	safe	74	1.5	14	14	2.1
GBSLL	safe	64	6.10	3	3	∞
GBDLL	safe	71	1.4	4	4	∞
GBSLLSent	safe	68	0.48	3	3	∞
GBDLLSent	safe	75	1.3	4	4	∞
RGSLL	safe	72	14.41	22	38	0.13
RGDLL	safe	76	78.76	26	26	0.13
WBSLL	safe	62	0.71	5	5	0.16
WBDLL	safe	71	1.60	7	7	0.14
SortedSLL	safe	190	227.12	15	15	2.2
SortedDLL	safe	82	120.00	11	11	2.20
EndSLL	safe	45	0.10	2	2	2.1
EndDLL	safe	49	0.14	3	3	1.8
TreeRB	error	130	0.08	0	0	∞
TreeWB	error	125	0.05	0	0	∞
TreeCnstr	safe	52	0.31	0	0	∞
TreeCnstr	error	52	0.03	0	0	∞
TreeOfCSLL	safe	109	0.57	0	0	∞
TreeOfCSLL	error	109	0.56	1	3	∞
TreeStack	safe	58	0.20	0	0	∞
TreeStack	error	58	0.01	0	0	∞
TreeDsw	safe	72	1.87	0	0	∞
TreeDsw	error	72	0.02	0	0	∞
TreeRootPtr	safe	62	1.43	0	0	∞
TreeRootPtr	error	62	0.17	2	6	∞

Chapter 4

Developing Shape Analyser for Software Verification Competition

4.1 Introduction

The following chapter describes development of the Forester tool [55] for Software verification competition [6] (SV-COMP) which is annually held within TACAS conference. The tools competing in SV-COMP vary from sound abstract interpreters or formal verifiers through static analysers to unsoud tools like bounded model checkers. The tools compete in verification of C programs from the competition benchmark. The length of programs are usually from tens to hundreds lines of code. The verification tasks are separated to different categories, e.g., reachability, termination, or memory safety category. Each category has its own specification of program invariants which are verified. E.g., in reachability category an invariant may be that a certain line of program is never reached.

A verification tool is supposed to answer for a given program and specification whether the program fulfill the specification (tool answers true), break specification (tool answers false), or tool cannot decide and gives an answer *unknown*. The tools are rewarded by 2 points for a correct answer true, 1 point for a correct answer false, or penalized by 32 points for a incorrect answer true, 16 points for an incorrect answer false. As one can see, since SV-COMP is a verification competition, not a testing competition, it favors sound tools over unsound ones by its scoring scheme.

Forester was participating in SV-COMP from 2015 to 2018. It was competing in the tasks related to memory safety and heap manipulation. These tasks were split to the two main categories: 1. a reachability subcategory containing programs working with dynamically allocated data structures, 2. memory safety categories. In memory safety categories, the checked invariants were absence of invalid memory dereferences, invalid memory deallocations, and memory leaks.

In particular, Forester competed in the categories *HeapManipulation, MemorySafety* in the year 2015, in the categories *HeapReach, HeapMemSafety* in 2016, and in the categories *ReachSafety-Heap, MemSafety-Heap, MemSafety-LinkedLists* in the years 2017 and 2018. It major success was the second places in *MemSafety-LinkedLists*, a sub-category of *MemSafety*, in the years 2017 and 2018.

Although Forester has never won any medal, but it was able to analyze some really tough test cases that have not been successfully verified by any other (sound) tool. Examples of these programs are programs with skip lists (of the second and the third level) and some

test cases from [65], which were part of the category *MemSafety-LinkedLists* in SV-COMP 2019. The former cases are hard to analyse because a tool needs to derive complicated shape invariants what Forester can do thanks to the automated box discovery. The latter cases are hard due to the need to derive invariants describing relations between nodes of data structures, which Forester can infer using the CEGAR loop.

We note that Forester was created as a proof of concept, to test the limits of forest automata in shape analysis, with the focus on difficult shape properties. It was not engineered to become a robust tool capable of handling a large class of practical C programs. It hence usually did not rank high in SV-COMP, since the programs targeted by Forester are only in a small minority in SV-COMP, which is aimed at evaluation of general software verifiers. Even the categories specialized to heap manipulation consist of programs that use a wide range of features of the C language, not only a small subset of syntax (focused on pointer manipulation) supported by Forester. Therefore, for many tasks, our tool did not even start verification, and even when the analysis successfully started, Forester often failed because of some unsupported feature of C (such as the standard C library functions `alloca` or `memset`, whose modelling is hard and needs to be done by hand, potentially making changes in the core engine of Forester). Still, Forester succeeded to show that the combination of forest automata with techniques from ARTMC are superior to number of other existing approaches in terms generality and flexibility, and that it is competitive with the best in terms of speed. Building a more robust tool on top of our experience with Forester is one of our future goals.

The main concepts of the verification approach used by Forester were described in detail in the previous parts of thesis therefore the rest of the chapter is going to mainly discuss technical improvements done during the preparations for different editions of competition.

4.2 Technical Preparation of Forester for SV-COMP

This section contains description of technical improvements of the tool we did for the different editions of the competition. The conceptual improvements of forest automata based shape analysis in different editions of the competition are described in the next section. For the first participation of Forester in SVCOMP (edition 2015) [64], no principle changes in Forester were done and we aimed to preparing the tool from technical perspective to fulfill the requirements of the competition. Forester did not have a counterexample analysis or an abstraction refinement in this first participation. The tool was submitted as a binary file in an archive together with the scripts wrapping execution for the competition. The scripts, e.g., translates outputs of Forester to predefined outputs by competition rules.

However, it was needed to add to Forester a support for printing out a path leading to a found counterexample violating the given specification, so called witness. The witness format is a dialect of GraphML (which is based on XML) defined by the competition rules and described in more detail way in [22]. We use the fact that Forester performs a symbolic execution and creates symbolic states containing link to a previous state. We travel a path of symbolic states from an erroneous state to the beginning of program and gradually generate needed witness in GraphML. The witnesses are later used by so called witness checkers which perform precise model checking (i.e., without abstraction) navigated by the input witness to check that a found error is really in program.

In the next edition of competition (year 2016), we adapt Forester to a new competition environment. Particularly, the competition started using Benchexec [24], a platform making

easy execution of the different tools over a given benchmark and reporting results in a structured way. We implemented a script called by Benchexec to execute Forester over a given verification task. This script calls another script directly encapsulating Forester and translating outputs of the tool the ones predefined one by competition. It was also needed to create a specification of participation in XML format and to upload it to SV-COMP repository. The specification defines in which categories Forester participates and which parameters should be used for a particular category.

Finally, since edition 2017 of SV-COMP Forester supports printing so called correctness witness. I.e., a description of state space explored during a verification of error-free task together with invariants implying correctness of program. A witness is again based on GraphML format defined by competition and described in [21]. In our case, the invariants are forest automata representing a shape invariant (i.e. all possible shapes of data structures allocated on the heap) for each line of program visited during verification procedure.

4.3 Conceptual Improvements over the Editions of Competition

For the first edition in which Forester participated (2015), we did mainly technical improvements of the tool. The main conceptual change was a replacement of ad-hoc implemented tree automata library by the VATA library [81]. It improved overall architecture of Forester (and therefore maintainability) and made possible to benefit from improvements in the VATA library which is specialized to an efficient implementation of algorithms for tree automata.

The main bottleneck of Forester in competition, an inability to distinguish between false and real bugs in program, was fixed for the next edition (2016) (a possibility of false bug is caused by using abstraction over forest automata as was described in the previous chapters). It was achieved by implementation of backward run from the point where a violation of specification was detected. The backward run starts with forest automaton representing data structures allocated on heap in a program state where the violation was found. It goes back through visited states during forward symbolic execution and reverts an effect of the operation done in the given state. Then it checks whether an intersection of forest automaton from forward run and backward run in the given state is empty or not. If it is empty the found violation is not real. When such a spurious counterexamples is detected, a refinement of abstraction is done. The intersection of forest automata needed for backward run was also newly implemented for this edition.

The backward run and abstraction refinement were implemented in the standard CEGAR loop. We used already existing forward run and added it to a loop where a (potentially) found counterexample is analysed by new backward run. If it is a spurious one we perform refinement abstraction, either by increasing height for height abstraction or by deriving new predicates for predicate abstraction which was newly implemented. Particularly, the new predicates forest automaton from backward. Once the abstraction is refined we can recover from detection of a spurious counterexample and restart the analysis.

Finally, for edition 2017 we extended backward run and abstraction refinement to hierarchical forest automata (forest automata with another forest automata, called boxes, used as symbols in their transitions). It complicates intersection of forest automata since the intersection needs to take semantics of boxes into account. One possible solution would be on fly unfolding of boxes which would make the intersection algorithm complicated.

Therefore we take another approach which keeps automata in forward and backward run in so called compatible form. That is for assuring that the same heaps are represented in the same way. From technical point of view we need to change implementation of the abstract operations performed in the forward run and their reversion in backward run. As it is was described in the previous chapter, the most challenging operations to revert in backward run w.r.t. compatible form are fold (creating box from forest automaton), unfold (reverting unfold), and normalization (i.e., operation of removing redundant cut-points and connecting associated tree automata).

4.4 Strengths and Weaknesses

The main strength of the tool in competition came from the generality of the underlying verification procedure. Forest automata as a domain are able to represent many different data structures, ranging from doubly linked list to different kinds of trees. E.g., Forester was the only tool able to soundly verify correctness of implementation of skip lists of the second and the third level. The newly implemented backward run and abstraction refinement moreover gave Forester an ability to learn more sophisticated invariants. An example of such invariant is alternation of two kinds of nodes in single or double linked list. This class of data structure were also tasks which only Forester verified soundly.

The weaknesses came from technical immaturity of the tool. It did not support the whole C syntax and therefore it reported unknown on many tasks from SV-COMP benchmark. The examples of features occurring often in memory safety related categories are function pointers, alloca function, or arrays. We lack manpower to add these features to Forester which was still prototype and would need a major refactoring to reach a state where an easy maintainability and extensibility would be possible.

Chapter 5

Towards Efficient Shape Analysis with Tree Automata

5.1 Introduction

The recent 20 years have seen a rise of many approaches to verification of pointer programs, aka shape analysis, up to their industrial deployment (e.g. the technique of [35] in Facebook's Infer). The existing approaches are mainly distinguished by the formalism used to describe sets of memory configurations (shape graphs), which are essentially graphs with nodes being memory locations and edges being pointers. The dominant position, previously held by frameworks such as [104, 92], is currently occupied by more automated and scalable approaches based on separation logic (SL) [103, 17, 37] such as symbolic memory graphs [48], on forest automata [57], and on graph grammars [58]. These approaches clearly identified the importance of local reasoning and modularity in reasoning about memory configurations as the key to scalability.

One of the major bottlenecks in the field is extending the techniques to more complex data structures: with anything beyond relatively simple variants of lists and trees, the existing approaches struggle with scalability and precision or require a non-trivial users assistance. None of the existing formalisms for describing shape graphs have all the following desirable properties. 1. *Expressiveness*: the ability to talk about variants variants lists, trees, structures such as skip-lists, threaded trees, their combinations and overlayer variants. 2. *Local reasoning*: running a program statement on the abstract domain should have only a local effect, it should be possible to reason locally about the affected parts. 3. *Effectiveness*: satisfiability and entailment should be efficiently decidable, as well as additional graph operations needed, e.g., in (higher-order) bi-abduction [35, 78]. 4. *Abstraction and generalization learning mechanisms*: the ability to learn abstractions of inductive invariants with controlled precision.

Separation logic approaches provides the first two qualities, expressiveness and local reasoning, but lacks in the other two. Earlier verification methods are not sufficiently general their decision procedures handle mostly just lists. The approaches such as [70, 52, 97] are rather restricted and incomplete, while the general approaches [71, 88] are theoretical and far from being efficiently implementable. The recent works [74, 95, 50, 49] finally came with an entailment for a large fragment of separation logic. These algorithms have not yet been tried within actual verification of pointer programs but are promising not only in the context of separation logic.

SL approaches so far lack ways to automatically learning shape invariants without a help of user-predefined patterns. The higher-order bi-abduction [78] is a notable exception: it is capable of learning extremely complex shape invariants such as B+ trees, skip-lists, or threaded trees. It is, however, very sensitive to how the code is written (since it is, in a sense, transforming the recursive code to inductive shape predicates) and hence quite fragile, easily failing on seemingly easy examples such as natural implementations of a doubly-linked list reversal.

Outside separation logic, especially the approach based on Forest automata [54, 57, 67, 65] has been shown viable [64, 61, 62]. It allows for some degree of local reasoning, it is efficient as it allows to utilise advanced algorithms for tree automata (such as simulation reduction or antichain language inclusion). The main distinguishing advantage is its compatibility with abstraction schemes from abstract regular model checking [64] with counterexample guided refinement loop. The formalism however suffers from some deficiencies, such as that expressible classes shape graphs are limited and that it is not closed under union.

We discuss here our ongoing work on developing a new graph formalism in the spirit of Forest automata that would remedy their weaknesses. We present main ideas on which such formalism can be built. First, we explain how graphs can be encoded into trees and tree automata (as a variation on tree decomposition of graphs [41] and also the formalism used in [71, 70]). We then discuss basic ideas for an entailment procedure for the formalism. We believe that this new formalism can eventually combine local reasoning of separation logic and forest automata, strong entailment procedures of [74, 95, 50, 49], efficiency of tree automata [10, 14, 8, 81], and powerful abstraction schemes of regular model checking.

5.2 Representing Graphs with Trees and Tree Automata

We will first discuss encoding of graphs as variations on tree decompositions, similar to that used in [71] and also [70].

A Σ-*labeled graph* is a pair $g = (V, E)$ where V is a finite set of nodes, $E \subseteq V \times \Sigma \times V$ is a set of Σ-labeled edges. A graph $g = (V, E)$ is *deterministic* if for every node $n \in V$ and every label $a \in \Sigma$, there is at most one node $n' \in V$ such that $(n, a, n') \in E$. Unless stated otherwise, we will assume all graphs deterministic.

A *tree decomposition* of a labeled graph g over a finite set of variables $\mathbb{X}$ and alphabet Σ is a tree $t = (B, E)$. Nodes B of t are Σ-labeled graphs called *bags*. Nodes of a bag are variables from $\mathbb{X}$. Edges of t are labelled by partial mappings $\rho : \mathbb{X} \to \mathbb{X}$ called *parameter assignments*. The *tree-width* of a decomposition, $tw(t)$, is the maximum cardinality of a parameter assignment in it. A *node occurrence* in t is a pair $(x, b) \in \mathbb{X} \times B$ where either x is a node of a bag $b \in B$ (i.e., $x \in b$) (so called an *active* occurrence) or x belongs to the image of the parameter assignment on the edge targeting b (then it is a *passive* occurrence), i.e., $x \in img(\rho)$ such that (b, ρ, b') is an edge of t and ρ is a label of the edge. The *alias relation* $\sim$ is the smallest equivalence of occurrences such that if $(b, \rho, b') \in E$ and $x' = \rho(x)$ then $(x, b) \sim (x', b')$. The Σ-labeled *graph represented by* t is the graph $g^t = (V^t, E^t)$ where the nodes V^t are the equivalence classes of $\sim$, called *pipes*, and $E^t = \{([(x, b)]_\sim, a, [(x', b)]_\sim) \mid (x, a, x')$ is an edge of the graph $b \in B\}$ (that is, every edge (x, a, x') of every graph $b \in B$ gives rise to an edge $([(x, b)]_\sim, a, [(x', b)]_\sim)$ of the represented graph g^t).

An example of a graph (representing circular doubly-linked list data structure) and its tree decomposition is shown in Figure 5.1 (a) and (b).

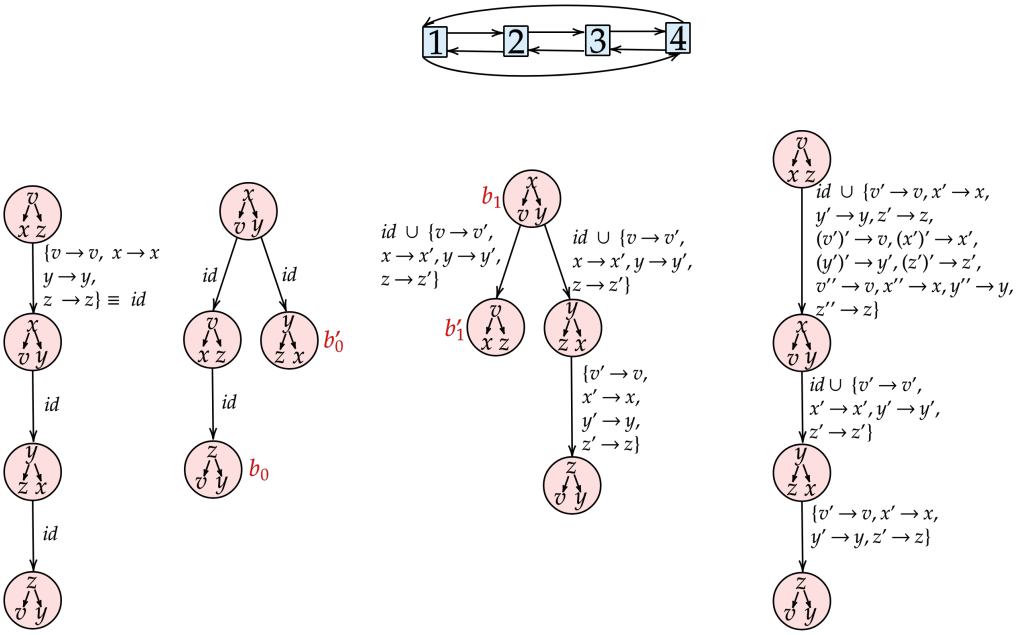


Figure 5.1: Figure shows circular doubly-linked list (a) and its tree decompositions. In the decompositions (b), the variables v, x, y, z represent the nodes $1, 2, 3, 4$ (in this order) of the list. The figure shows that it is possible to transform the right decomposition in (b) to the left one using transformations shown in (c) and (d). Figure (c) illustrates the reconnection operation. It shows the decomposition obtained by reconnection applied to the right-handed side decomposition of (b) where the bag b_0 is reconnected below the bag b'_0. The figure also shows the primed variables introduced by the reconnection to prevent inference of reconnected pipes with pipes along the reconnection path. Analogously, (d) show result of rotation applied to (c) where the bags b_1 and b'_1 were rotated. The operation changes orientation of edge between them and also introduces the new primed variables for each existing variable, e.g., $(x')'$ for x' and since x' already exists, x'' is created for x.

We will work with the following restrictions of graphs and tree decompositions. If a bag b has an edge originating at x, then the pipe $[(x, b)]_\sim$ is *allocated at* b. A *backbone decomposition* corresponds to an (unoriented) tree backbone of the graph. It has three defining properties: 1. Every bag b *allocates exactly one graph node*. 2. Every graph node is *allocated only once*. 3. The tree is *connected* in the sense that every tree edge corresponds to a graph edge (regardless the edge orientation). That is, for every two adjacent bags, one of them, say b, has an edge adjacent with x, and the other, b', has an active occurrence (b', z) with $(b, x) \sim (b', z)$.

Last, assuming a backbone decomposition, we define so called *pipe child* relation $\rhd$. Two pipes p and p' of a decomposition t are in the relation $p \rhd p'$ iff p is allocated in node b and p' in node b' such that $(b, a, b') \in E$ for some a.

A set of tree decompositions can be represented by a tree automaton. Intuitively, a node of a tree represents a node in the tree decomposition, and the label of each tree node records the bag and the labels on the decomposition edges leading from the node to its children.

5.3 Towards Entailment

The idea for deciding entailment is to use tree automata language inclusion algorithms (such as [**?**, 14, 8, 81]) over tree automata encodings of tree decompositions. The difficulty here is that a single graph has multiple decompositions and the tree automata may accept only some of them, hence simple language inclusion check may underapproximate the inclusion of sets of represented graphs (entailment). We therefore propose means of saturating the tree automata languages with all possible tree decompositions of the represented graphs. Conceptually, we will define a small set of operations which is *complete* for tree decompositions, that is, allows to transform a decomposition into any other decomposition of the same graph. The two tree automata under the entitlement accept decompositions with certain maximum tree width t, that can be easily determined. The entailment procedure will apply the decomposition operations symbolically over the tree automata until they are saturated with all tree decompositions of the represented graphs with the tree width up-to t. Computing the language inclusion of thus saturated automata is then a sound algorithm for entailment.

We will now outline the operations. These operations are essentially meant to complete the *rotation* operation of [**?**].

Reconnection. The operation of reconnection is parameterised by two *peer* bags b and b' of the decomposition that allocate pipes p and p', respectively (they are peer bags, i.e. not on the same branch of the tree). Its purpose is to create an equivalent decomposition with the child relation on graph nodes being the same up to that p becomes a child of p' in the new decomposition.

The operation is implemented as a function $\mathit{reconnect}(b, b', t)$ that transforms a tree decomposition $t = (B, E)$ into t' as follows. First, the pipes that are reaching b in t must be in t' sent to b' through the path between b and b', called the *reconnection path*. Let $\rho_1, \ldots, \rho_k$ be the sequence of edge labels appearing on the reconnection path. Take every label ρ_i of an edge on the reconnection path with $2 < i \leq k$, and replace it by $\rho'_i = \rho_i \cup \{x' \mapsto x' \mid x \in \mathit{img}(\rho_1)\}$. The primed variables must be such that they do not appear anywhere on the reconnection path, neither in the graph bags nor in the variable renamings (they may be fresh variables). This is to stretch the pipes reaching b through the reconnection path towards b'. Replace also ρ_2 by $\rho'_2 = \rho_2 \cup \{x' \mapsto y \mid \rho_1(y) = x\}$. This binds the new primed part of the pipes with the original pipes reaching n. Last, replace the edge leading to b by (b', ρ'_1, b) where $\rho'_1 = \{x' \mapsto x \mid x \in \mathit{img}(\rho_1)\}$. This makes b a child of b' and connects the new primed pipes to the corresponding original pipes of b.

An example of the operation is shown in Figure 5.1 (c) where the righ-handed side decomposition t_0 from Figure 5.1 (b) is rotated by $\mathit{reconnect}(b_0, b'_0, t_0)$.

Rotation. The rotation operation is parameterised by two bags b and b' of the decomposition. The operation inverts the edges in the path π between b and b'. Then it redirects the incoming edge of b to b'. Intuitively, it takes a subtree with the root b, changes the root to b' and inverts the edges in the subtree. It yields an equivalent tree decomposition with respect to child relation which is inverted between pipes allocated in the nodes of the path π.

The operation is implemented as a function $\mathit{rotation}(b, b', t)$. At the tree level, it works as follows. Consider the path $\pi : b = b_1, \ldots, b_n = b'$. Remove each edge (b_i, ρ, b_{i+1}), where $1 \leq i \leq n$, between nodes in π from E and add (b_{i+1}, ρ, b_i) to E. Moreover, we

replace the edge $(m, \rho, b) \in E$ by the edge (m, ρ, b') to E. The labels along the rotated path are changed in the following way. Replace the label ρ_1 of the edge (m, ρ_1, b') by $\rho_1 = \{x \mapsto x' \mid x \in dom(\rho_1)\}$. Replace each label ρ_i in the path π by $\rho_i = \rho_i \cup \{x' \mapsto x' \mid x' \in img(\rho'_1)\}$ for $2 \leq i \leq b$. Finally, replace the label ρ_b over the edge leading to b by $\rho'_b = \rho_b \cup \{x' \mapsto x \mid x' \in img(\rho'_1)\}$.

An example of the operation is shown in Figure 5.1 (d) where the decomposition t_1 from Figure 5.1 (c) is rotated by $rotation(b_1, b'_1, t_1)$.

Phase. The operations from one tree decomposition to an equivalent one may take unboundedly many operations. We will devide them into *phases*. One phase can still perform unboundedly many operations, but the set is restricted: the reconnection paths of all reconnections must be node disjoint.

Formally, a phase is characterised by a set of operation parameters, pairs of nodes $\{(b_1, b'_1), \ldots, (b_k, b'_k)\}$ of a decomposition t such the for any two $1 \leq i, j \leq k, i \neq j$, the operation paths between b_i and b'_i and between b_j and b'_j are node disjoint. A result of the phase is any decomposition which arises by performing the appropriate operation on (b_i, b'_i) for each i (in any order).

An example of two phases are show in Figures 5.1 (c) and (d) where the right-handed side decomposition from Figure 5.1 (b) is transformed in the left-handed side decomposition of the same figure in these two phases.

An important consequence of the disjointness of the reconnection paths is that all the operations can be implemented while only doubling the number of variables and the tree-width of the original decomposition. Particularly, all the operations can use the same set of fresh primed versions of the variables $\mathbb{X}$ on the operation path without fearing a conflict with the names of the existing pipes.

Lemma 3 *One phase at most doubles the number of variables.*

We conjecture that the number of phases needed depends on only on the tree-width:

Conjecture 1 *An equivalent decomposition t' can be obtained from t in a number of phases that depends only on* $\max(tw(t), tw(t'))$.

Next, we discuss implementation of a phase over tree automata (TA) representation. We design phase over TA in such way that its results is an automaton that encodes all possible results of phase applied at any possible decomposition represented by the original automaton. We will briefly sketch the basic idea of the operation.

Namely, saturation with reconnections can be implemented by a tree transducer. Seen as a top-down machine, it oscillates between two routines, *idle*, and *reconnecting*. In the idle state, it is just traversing the tree. At any node r, it may non-deterministically chose to start reconnecting. When reconnecting, it non-deterministically selects two peer descendants of r, nodes b and b', and performs the reconnection on them. The reconnection, roughly, involves adding of primed versions of existing pipes on the path from b to b' and reconnecting the subtree of b below b'. After that, the reconnecting phase stops and the transducer continues traversing the tree in the idle phase. The requirement in the definition of phase on the disjointness of the reconnection maths makes this doable—when reconnecting, the transducer needs to worry only about one reconnection path at a time. Saturation with rotations can be then implemented similarly as in [70].

We conjecture, based partially on Lemma 3, that such implementation of a phase over tree automata representation is cheap:

Conjecture 2 *The implementation of a tree automata phase at most doubles the number of variables and leads to an automaton that is of a polynomial size assuming a fixed tree-width of the original automaton.*

Based on Conjecture 1 and 2, the saturation of a tree automata representation with all decompositions until a fixed tree-width t can be done in a time that is polynomial (when the t is fixed). Recall that deciding entailment between two TA representations, we saturate both of them up to the maximum tree-width t (in fact, it is enough to use the maximum tree with of the automaton that is supposed to entail), and then we compute language inclusion between the saturated automata. Since language inclusion of tree automata is EXPTIME-complete, Conjectures 1 and 2 give an entailment algorithm singly exponential, assuming a fixed maximum tree-width t.

5.4 Conclusions and Future Work

We have presented basic outline of a formalism for representing shape graphs based on tree automata. The ideas should lead to an entailment algorithm, and conjectures that, if true, would imply that the algorithm is relatively fast assuming fixed maximum tree-width of the graph representations.

We plan to perfect these ideas and to prove Conjectures 1 and 2. The conjectures are somewhat optimistic, but not in a direct contradiction with the recent 2-EXPTIME-hardness result of [49]. If the conjectures turn out to be false, we wish to search for (1) restrictions under which they are true, and/or (2) to prove termination of the described entailment algorithm regardless its complexity. Our long term plan is to develop a shape analysis framework based on this formalism and entailment check in the spirit of [67] and also [78].

Chapter 6

Shape Analysis based on SMT Solving

Apart from automata based approach the author of this thesis participated in work on shape analysis based on SMT solving. Although we are not the main authors and have a rather minor share in this work (published in [86]) we add it here to give another perspective on shape analysis. We provide with an overview of the approach. For a full technical description please check [86].

The verification method is based on describing state space of a program w.r.t. to a property of interest using system of logic formulae and so called templates which describes state space of program using SMT solver. The templates are special logic formulae and are conceptually similar to abstract domain in abstract interpretation. They are manually crafted to capture only certain aspects of an analysed program what makes analyses computationally feasible. Therefore we call templates (abstract) domains in the rest of the chapter.

The main contributions of this work are:

1. We propose a novel abstract template domain for reasoning over heap-allocated data structures such as singly and doubly linked lists using a template-based parameter synthesis engine.
2. Since we designed our domain in existing framework for template-based verification we can build product and power domain combinations of our heap domain with structural domains (e.g. trace partitioning) and value domains such as template polyhedra that capture the content of data structures.
3. We implement our abstract heap domain in the 2LS verification tool for C programs. We demonstrate the power of the proposed domain on benchmarks, which require combined reasoning about the shape and content of data structures, showing that other tools, which performed well in SV-COMP, cannot handle these examples.

In this chapter, we firstly describe the verification method and then the domain for shape analysis which we designed.

6.1 Template-based Program Verification

In template-based program verification, a state of a program is a logical interpretation of logical variables corresponding to each program variable. A set of states can be described using a formula—states in the set are defined by models of the formula. Given a vector

of variables $\vec{x}$, the predicate $Init(\vec{x})$ describes the initial states. A transition relation is described as a formula $Trans(\vec{x}, \vec{x}')$.

From these, it is possible to determine the set of reachable states as the least fixed-point of the transition relation starting from the states described by $Init(\vec{x})$. This is, however, difficult to compute, so instead, we use an *inductive invariant*. A verification task requires showing that the set of reachable states does not intersect with the set of error states $Err(\vec{x})$. Using the concept of inductive invariants and existential second-order quantification ($\exists_2$), we can formalise it as:

$$\begin{aligned} \exists_2 Inv.\ \forall \vec{x}, \vec{x}'.\ & (Init(\vec{x}) \implies Inv(\vec{x})) \wedge \\ & (Inv(\vec{x}) \wedge Trans(\vec{x}, \vec{x}') \implies Inv(\vec{x}')) \wedge \\ & (Inv(\vec{x}) \Rightarrow \neg Err(\vec{x})) \end{aligned} \tag{6.1}$$

The above equitation would need a solver able to deal with second-order logic quantification, however, there are currently no general and efficient solvers able to do it. Therefore we reduce the problem to iterative application of a first-order solver by restriction of inductive invariant Inv to $\mathcal{T}(\vec{x}, \vec{\delta})$ where $\mathcal{T}$ is a fixed expression (a so-called *template*) over program variables $\vec{x}$ and template parameters $\vec{\delta}$. Template captures only properties of interest for a certain analysis (which corresponds to choice of an abstract domain in abstract interpretation). The resulting first order search is following:

$$\begin{aligned} \exists \vec{\delta}.\ \forall \vec{x}, \vec{x}'.\ & (Init(\vec{x}) \implies \mathcal{T}(\vec{x}, \vec{\delta})) \wedge \\ & (\mathcal{T}(\vec{x}, \vec{\delta}) \wedge Trans(\vec{x}, \vec{x}') \implies \mathcal{T}(\vec{x}', \vec{\delta})) \end{aligned} \tag{6.2}$$

Finally, since quantifier alternation is a problem for current SMT solvers we iteratively search for the negated formula:

$$\begin{aligned} \exists \vec{x}, \vec{x}'.\ & \neg(Init(\vec{x}) \implies \mathcal{T}(\vec{x}, \vec{d})) \vee \\ & \neg(\mathcal{T}(\vec{x}, \vec{d}) \wedge Trans(\vec{x}, \vec{x}') \implies \mathcal{T}(\vec{x}', \vec{d})) \end{aligned} \tag{6.3}$$

6.1.1 Program Encoding

We encode the program into a formula representing a specific *static single assignment form* (SSA). As usual, we use a fresh copy x_i of each variable x for each program location i where the value of x is modified. In this encoding, special variables called *guards* are used to track the control flow of the program in such way that for each program location i there is a Boolean variable g_i which value encodes whether the program location is reachable.

To achieve acyclicity of the SSA, we cut the path coming from the end of the loop. We then represent the value of each variable x at the loop head using a *phi variable* x^{phi} whose value is defined by a non-deterministic choice between the value coming from before the loop, say x_0, and the value coming from the end of the loop. The latter value is represented by a newly introduced *loop-back* variable x^{lb}. In particular, we let $x^{phi} = g^{ls}\ ?\ x^{lb} : x_0$ where g^{ls} is a so-called *loop-select* Boolean guard that is unconstrained in order to model the non-deterministic choice. The effect of the loop is overapproximated and the value of the loop-back variable x^{lb} is initially unconstrained and later constrained by the derived candidate loop invariants

6.2 Template domain for Shape Analysis

We design a template domain which are used in the described framework to derive *loop invariants* of the analysed programs. Templates are also used to constrain values of the *loop-back variables* that are used in the SSA-based program encoding to over-approximate values returning from the end of the loop to the loop head. Since we are interested in describing shapes of heaps, we limit our shape domain to the set Ptr^{lb} of all *loop-back pointers*. Let L be the set of all loops in the program. Since there is one loop-back pointer variable for each pointer variable and each loop, we define $Ptr^{lb} = Ptr \times L$. We denote elements $(p, l) \in Ptr^{lb}$ by p_i^{lb} where i is the program location of the end of the loop l. Intuitively, the value of p_i^{lb} is an abstraction of the value of the pointer p coming from the end of the body of the loop l. The property that our base shape domain describes is the *may-point-to* relation between the set Ptr^{lb} and the set $Addr$ of memory addresses.

The template of our base shape domain has the form of the formula:

$$\mathcal{T}^S \equiv \bigwedge_{p_i^{lb} \in Ptr^{lb}} \mathcal{T}^S_{p_i^{lb}}(d_{p_i^{lb}}). \tag{6.4}$$

It is a conjunction of so-called *template rows* $\mathcal{T}^S_{p_i^{lb}}$, each row corresponding to one loop-back pointer from the set Ptr^{lb}. A template row $\mathcal{T}^S_{p_i^{lb}}(d_{p_i^{lb}})$ describes the may-point-to relation for the loop-back pointer p_i^{lb}. The parameter $d_{p_i^{lb}} \subseteq Addr$ of the row (a so-called *abstract value of the row*) specifies the set of all addresses from the set $Addr$ that p may point to at the location i. The template row can thus be expressed as the quantifier-free formula:

$$\mathcal{T}^S_{p_i^{lb}}(d_{p_i^{lb}}) \equiv (\bigvee_{a \in d_{p_i^{lb}}} p_i^{lb} = a) \tag{6.5}$$

Abstract values of template rows corresponding to pointer fields of abstract dynamic objects allow the domain to describe unbounded linked paths in the heap, such as list segments.

In order to use the base shape domain in our approach, we have to augment it with information about the guard variables that encode the program's control flow in the SSA. The guards express when an appropriate loop-back control edge is executed and the loop-back pointer has a defined value[1]. A row of a *guarded shape template* is defined as a formula:

$$\mathcal{T}^G_{p_i^{lb}}(d_{p_i^{lb}}) \equiv G_{p_i^{lb}} \Rightarrow \mathcal{T}^S_{p_i^{lb}}(d_{p_i^{lb}}) \tag{6.6}$$

where $G_{p_i^{lb}}$ is a conjunction of SSA guards associated with the definition of the variable p_i^{lb} and $\mathcal{T}^S_{p_i^{lb}}$ is as in the base shape domain. If $G_{p_i^{lb}}$ is true for a program run, the definition of p_i^{lb} was reached in the run. A shape template $\mathcal{T}^G$ with guards is then a conjunction:

$$\mathcal{T}^G \equiv \bigwedge_{p_i^{lb} \in Ptr^{lb}} \mathcal{T}^G_{p_i^{lb}}(d_{p_i^{lb}}) \tag{6.7}$$

Let p_i^{lb} be a loop-back pointer abstracting the value of a pointer $p \in Ptr$ coming from the end of a loop $l \in L$. The row guard $G_{p_i^{lb}}$ is a conjunction of the following guards:

[1] Using the base domain without the guard variables would be sound. However, it would produce very imprecise results since the abstract value would need to cover even states in which the loop-back edge was not taken.

- The guard g_j^{lh} linked with the head of the loop l located at program location j, encoding that the loop l is reachable.
- The guard g_i^{ls} linked with the use of p_i^{lb}. The value of g_i^{ls} is true if p_i^{lb} is chosen as the value of the corresponding *phi* variable at the head of l.
- If p_i^{lb} describes a pointer field of some abstract dynamic object (i.e. it has the form $ao_j^k.f_i^{lb}$ for some $ao_j^k \in AO, f \in Fld$ where AO is a set of all abstract dynamic object and Fld a set of all fields of abstract objects), we also use the guard $g^{ao_j^k}$ linked with the allocation of ao_j^k at program location j. This guard conjoins the guard expressing reachability of program location j with the object-select guards $g_{j,l}^{os}$ and their negations denoting allocation of the k-th materialisation ao_j^k of the object allocated at j.

Once we assemble a set of templates describing possible we derive shape invariants using SMT solver (since shape invariants are models of the logic formulae put into SMT solver).

6.3 Conclusion

The previous sections provided a brief and dense introduction to the template-based approach to static analysis and it extension by shape analysis.

Beyond the described aspects we needed to design operations *dynamic memory allocation* (`malloc`), *reading through dereferenced pointers*, and *memory free* in our abstract template domain to be able to model semantics of programs. As we mentioned above, we also designed combination of the described abstract domain for shape analysis with others already existing domains.

The last part of the described method is checking that a program contains no *absence of null pointer dereferences*, *free safety*, and *absence of memory leaks*. E.g., to check absence of null pointer dereference, we verify the assertion $p_i \neq \mathsf{null}$, where p_i is the version of p valid at a program location i, for each expression $*p$ occurring in every i.

A detailed description of abstract memory operations or how all three memory safety properties are checked can be found in [86] as well as empirical evaluation of our method on benchmark of heap manipulating programs. Their full description is out of scope of the thesis and since we are not the main authors of the approach we limited ourselves to this brief overview.

Part II

Automata in Software Testing

Chapter 7

Orchestrating Digital Twins for Distributed Manufacturing Execution Systems

7.1 Introduction

One of the main challenges of the Industry 4.0 is to develop secure and bug-free components, especially in distributed, manufacturing execution systems. These systems usually include manufacturing machines paired with controlling terminals, industrial control systems (ICS), and/or system that controls the whole manufactory process (manufactory execution system). Development and testing of such systems is quite complex because their components (1) work in distributed environment, (2) use different communication protocols, (3) use different software ranging from low-level embedded software to complex information systems, (4) require interaction between humans and machines, and (5) often cannot be tested in a real-world environment during the common traffic.

Moreover, any bug or security issue may be quite costly which can be substantiated by the expected grow of the market of ICS security up to $22.2 billions by 2025 [1]. Quality assurance teams usually utilize some form of test automation while keeping effort spent on the testing itself. Unfortunately, test automation of distributed manufacturing systems is hard for two main reasons.

First, testing in a real-world environment (so called out-of-the-lab testing) is expensive. Hence, we usually construct the so called digital twin: a virtual environment where components (such as production machines) are emulated or simulated to replicate the digital copy of manufacturing process. Such a copy can then be used for testing in an environment as close as possible to a real system.

The other problem is how to model the communication among number of quite different components common in manufacturing process. The communication within the system is often purpose-specific and requires strong domain knowledge. Hence, creating the automated test suite is complicated as it requires effort spent on precise test environment setup and deterministic test case description.

In this work, we propose a generic framework for creating automated test suites for digital twins of manufacturing execution systems (MES). The framework analyses the communication captured from a run of the real system, learns a model of the communication protocol and models of data sent. Based on these models, we generate test scenarios that

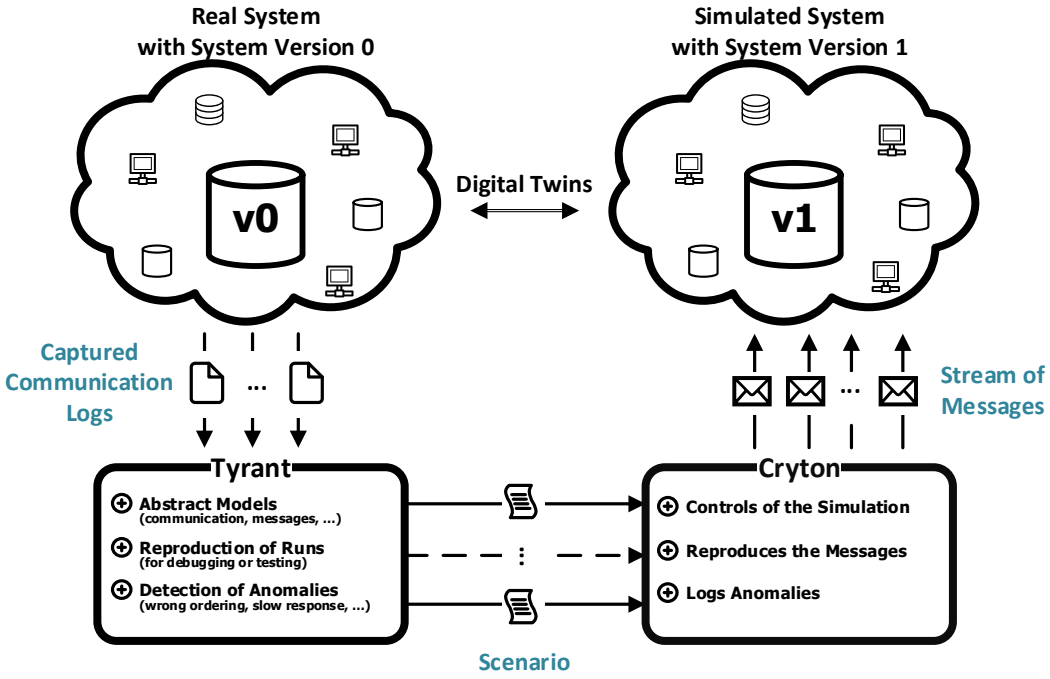


Figure 7.1: Scheme of our solution.

are used for orchestration of corresponding digital twin. In particular, we suggest to use such scenarios for automated testing of systems, e.g., when a new version of MES is being developed.

7.2 Framework for Generating Orchestration Scenarios

We propose a generic framework that can be applied to various settings of distributed MES systems. In this paper, however, we will demonstrate it on a particular use case consisting of various types of nodes communicating using various protocols. We assume the following infrastructure: the distributed system consists of an Enterprise Resource Planning (ERP) system, the MES system that controls the actual production, manufacturing machines and their corresponding terminals used by human operators. We expect that the communication between particular components uses different protocols and data structures, e.g.: (1) MES and ERP communicate using REST protocol with XML data, (2) MES and Terminal communicate using REST protocol with JSON data, and (3) MES communicates with machines using OPC-UA protocol, although some minor manual tweaks for understanding specific-purpose data might be necessary.

An overview of our framework is shown in Figure 7.1. Our framework requires logs of communication collected from a real-world system, e.g., with an older version of MES system under testing: the collected log usually represents either expected communication in the system, or a log of communication that led to some incident. The log is in form of a sequence of messages between pairs of communicating components logged with the timestamps of the communication and the data that were transferred. We derive two kinds

of model based on this log: (1) model of the data transmitted in the messages, and (2) the model of the whole communication in the system. To model communication, we convert the log to a so called event calendar which provides efficient and direct manipulation with seen messages. Then, we eventually convert event the calendar to a finite automaton (where every event is a symbol) which is more abstract representation but provides options for postprocessing (e.g., by applying length abstraction to generate new test cases) or analyses (e.g., by searching for particular string representing an error behaviour). From the derived (abstract) models, we generate a so called scenario: a sequence of concrete messages that will be sent in a real-world or simulated environment. A scenario is later used by a digital twin orchestrator to perform simulation of real system in digital twin. In our case, we use the Cryton tool [2] to orchestrate simulated environment. Other orchestrators that conform to our format of scenarios can be naturally used as well. Finally, based on the result of orchestration developers can observe whether the new version in a digital twin behaves as expected.

The framework can also be quite easily extended to support the performance testing of the digital twins. In particular, we propose to mine selected performance metrics (e.g., among others, the duration of communications) from the captured logs. The metrics are then used for comparison of runs from different environments or from different versions to detect, e.g., anomalies in the performance.

Related work. There has been several different approaches for modelling communication in manufactory and deriving new test cases. [20] uses Finite Automata to model the communication in systems, however, their approach is limited to learning only fixed number of components. Another approach is the process mining [7], a mature technique for modelling event based system. We see the technique as not suited well for MES systems as it analyses one-to-one communications and is restricted to a single thread per node [80]. Modelling of communication for anomaly detection [89] implements an approach based on probabilistic automata. The usual communication in manufactory is, however, mostly deterministic, hence, no probabilistic transitions are created in derived automaton. Finally, we can mention approaches of [40, 69, 12] which are research prototypes only and possible not mature enough for use case in real-world distributed systems.

Beside modelling communication we need also to model and generate new messages. In our use case, the messages are in JSON and XML format. There are several tools which can generate testing data in the formats. One of them is *DTM Data Generator for JSON* [4] which can generate testing JSON data having structure defined manually by user or derived from an example input document. It supports deriving relations in JSON document and contains domain specific built-in data generator (e.g., address, phones, or URLs). Moreover it can also combine more structures to new test cases. However, the software is proprietary, it is not clear how other formats than XML would be supported and it does not provide strong abstraction over the input data which would allow extrapolation for generating testing data. Another example of tool for generating input data in JSON is *JSON Schema TestData Generator* [5]. The tool is open source but quite simple. It generates test data based on a given JSON scheme and does not provide any functionality for deriving relations between data in JSON document or ability to learn and abstract from a given set of JSON documents.

For generating XML, there two class of approaches. The first one is based on generating XML according to a given XPath query (so called category-partition based) [94, 76]. The advantage is ability to change specification easily and so generated data. However, it is not

suitable for our purposes since we want learn syntactic structure of logged messages, not to write their specification manually. The second one is based on generating XML documents based on a given XML schema [19] instance which is more close to our needs. But it does not solve generalization of the same class of message to one generic syntactic structure.

7.3 Modelling Messages

In distributed system, components usually communicate through messages. We assume, that each message that was captured in the log has the following parts: (1) a timestamp (when the message was sent), (2) data (what was sent), and, (3) a type (what kind of message was sent). A suitable data representation of such messages can be a challenging task, especially, when modelling the communication among different components. The representation should be unified for different kinds of data formats (such as JSON, XML or YAML), should preserve the original semantics, and should allow generating new test cases from the observed data (e.g., extrapolating extreme values from the underlying domains).

Communication logs usually contain lots of subsets of messages that are structurally similar to each other and differ only in a certain aspects (mainly in the data that were sent and the type of the message). Thus, we propose classification of the messages into a groups of similar messages before creating abstract models. In particular, we classify the seen messages based on the so called fingerprint of the message (i.e., the spanning tree of the nested structure with respect to the fields of the data) and based on the type of the message. The idea is that messages having similar structure (but that differ in, e.g., number of items in a list, or values in leaves) should have the same abstract model. For each such class, we construct an abstract model that represents all seen messages of the given class. Such a model can then be used not just to reproduce the communication but also to create new (potentially unseen) messages, e.g., by generating syntactically-similar messages.

We propose to model the messages using the following representation (simplified for the sake of presentation). We assume two types of nodes: (1) a leaf node is a quadruple $n = \langle k, l, u, V \rangle$, where k is a key associated with the node (e.g., as in JSON key-value pairs), l (resp. u) is the minimum (resp. maximum) number of occurrences of the node in the given part of message, and V is the set of all seen values for the node; and (2) a composite node is a quadruple $n = \langle k, l, u, N \rangle$, where k, l, and u are defined the same as previously and N is a set of child nodes (either leaves or composite). We assume that the root of every message is represented by the *root* key. Note that we support also other types of nodes, e.g., the attribute node, used in XML format, but due to the space constrains we omit their description.

To create abstract models, we process input log message by message (which are in XML or JSON format). Atomic values correspond to leaf nodes and composite values (lists, dictionaries, nested tags) correspond to composite nodes. Further, we will work with predicate c over node n written as $c(n)$ (e.g., representing that node n has a specific key). We denote set of all possible predicates as C.

We define the function *reduce* over a set of leaves $\{n_1, \ldots, n_m\}$ with the same key k as $reduce(\{\langle k, l_1, u_1, V_1 \rangle, \ \ldots, \ \langle k, l_n, u_n, V_n \rangle\}) = \langle k, \sum_1^n l_i, \sum_1^n u_i, \ \bigcup_1^n V_i \rangle$; similarly, we define the *reduce* of a set of composite nodes $\{n_1, \ldots, n_m\}$ corresponding to a key k as $reduce(\{\langle k, l_1, u_1, N_1 \rangle, \ldots, \langle k, l_n, u_n, N_n \rangle\}) = \langle k, \sum_1^n l_i, \sum_1^n u_i, \bigcup_1^n N_i \rangle$. Then, for composite nodes, we define the *group and reduce* function as $grpreduce(\langle k, l, u, N \rangle, \langle c_1, \ldots, c_m \rangle) = \langle k, l, u, N' \rangle$, where $\langle c_1, \ldots, c_m \rangle$ are predicates and $N' = \bigcup_1^m \left\{ reduce(\{n \in N \mid c_i(n)\}) \right\}$. Basically, the operation groups the children nodes according to a given predicate (e.g.,

it groups children named with the same key), merges their values and aggregates their occurrences.

Finally, we define the *merge* of two nodes $(n \circ n')$ with the same key k as follows: (1) $\langle k, l_1, u_1, V_1\rangle \circ \langle k, l_2, u_2, V_2\rangle = \langle k, min(l_1, l_2), max(u_1, u_2), V_1 \cup V_2\rangle$, and (2) $\langle k, l_1, u_1, N_1\rangle \circ \langle k, l_2, u_2, N_2\rangle = \langle k, min(l_1, l_2), max(u_1, u_2), Merged(N_1, N_2) \cup Copy(N_1, N_2) \cup Copy(N_2, N_1)\rangle$, where $Merged(N, N') = \{n \circ n' \mid \exists c \in C\, \exists n \in N\, \exists n' \in N' : c(n) \wedge c(n')\}$ and $Copied(N, N') = \{n \mid \exists c \in C\, \exists n \in N\, \forall n' \in N' : c(n) \wedge \neg c(n')\}$. We choose values of minimal and maximal number of occurrences to cover both nodes. In the composite nodes, we group the children satisfying the same criteria and recursively merge them. If there is a child node from N_1 (resp. N_2) with no node from N_2 (resp. N_1) that matches the same criterion the first node is simply copied to the result. For simplicity, we assume that a criterion c is satisfied by maximally one node in one subtree. Finally, for each class and its messages with the root nodes $r_1, \ldots, r_n$, we compute the final abstract node n representing the class as $n = grpreduce(r_1, \langle c_1, \ldots, c_m\rangle) \circ \ldots \circ grpreduce(r_n, \langle c_1, \ldots, c_m\rangle)$.

7.4 Modelling Communication of Monitored System

Once we derived the models of messages communicated in a system, we further learn the communication protocol used in the environment. We first use an intermediate data structure called the *event calendar* to represent messages in the monitored system where each event corresponds to one message. The messages are ordered chronologically in the calendar by their timestamps. This way we can represent the communication using different protocols and data formats in a unified and regular manner and we are not limited to a fixed number of components. That would not be possible with other representations which need predefined topology of a represented system. The calendar is later used to generate scenario precisely reproducing the learnt communication by transforming each event to a single step in a scenario for orchestrating digital twin.

Moreover, we want to generate new test cases allowing to experiment on scenarios which have not yet been seen but are similar to a real-world situations. Such scenarios bring sometimes more testing value since they are relatively easy to generate in contrast to the time demanding process of writing tests manually. Hence, we propose to transform the event calendar to *finite automaton* and apply, e.g., length abstraction which over-approximates language of the automaton. In the following paragraphs, we define our method in a formal way.

An event is a tuple $e = (t, s, r, time, m)$ where t is the type of communication protocol (i.e., OPC-UA or REST), s is the identification of the sender, r is the identification of the receiver, *time* is a timestamp, and m is an abstract representation of the sent message described in the previous section. Event calendar c is a list of events $c = (e_1, \ldots, e_n)$. A finite automaton is tuple $\mathcal{A} = (Q, \Sigma, \delta, I, F)$ where Q is a finite set of states, Σ is a finite alphabet, $\delta \subset Q \times \Sigma \times 2^Q$ is a transition relation, $I \subseteq Q$ is a set of initial states, $F \subseteq Q$ is a set of final states. A language L of automaton $\mathcal{A}$, denoted by $L(\mathcal{A})$, is a subset of Σ^*. A run ρ of automaton $\mathcal{A}$ is a sequence of states $(q_1, \ldots, q_n)$ such that $\forall 1 \leq i \leq n-1 : \exists a \in \Sigma : q_{i+1} \in \delta(q_i, a)$. A word $w = a_1, \ldots, a_n$ is accepted by the automaton $\mathcal{A}$ iff there is a run $\rho = (q_1, \ldots, q_{n+1})$ of $\mathcal{A}$ such that $\forall 1 \leq i \leq n : q_{i+1} \in \delta(q_i, a_i)$ and $q_{n+1} \in F$. A language L_q of a state $q \in Q$ is a set $\{w = a_1, \ldots, a_n \mid w \textit{ is accepted by a run } \rho = q_1, \ldots, q_{n+1} \textit{ such that } q_1 \in I \wedge q_{n+1} \in F\}$.

Event calendar $c = (e_1, \ldots, e_n)$ is transformed to a finite automaton $\mathcal{A}_c = (Q^c, \Sigma, \delta^c, I^c, F^c)$ as follows: the set of states is $Q^c = \{q_1, \ldots, q_{n+1}\}$, the alphabet Σ is obtained by transform-

ing each event $e = (t, s, r, time, m)$ to an unique symbol a^e by applying a hashing function over $(t, s, r, time)$, i.e., giving away m, the set of initial states is $I^c = \{q_1\}$ and the set of final states is $F^c = \{q_{n+1}\}$. Finally, $\forall 2 \leq i \leq n+1 : (q_{i-1}, a^{e_i}, \{q_i\})$ is added to δ^c.

In order to create the new scenarios, we need to overapproximate the models. In particular, we propose to use the length abstraction transforming the automaton $\mathcal{A}$ to an abstracted automaton $\mathcal{A}^k$ by merging all states with the same language with respect to a given length. Formally, a length abstraction over an automaton $\mathcal{A} = (Q, \Sigma, \delta, I, F)$ is an equivalence relation $\alpha^k \subseteq Q \times Q$ such that $(p, p') \in \alpha^k$ iff $L_p^n = L_{p'}^n$ where $L_q^n = \{w' \mid \exists w \in L_q : w' \textit{ is a prefix of } w \wedge \textit{length of } w' \textit{ is up to } n\}$. We denote an equivalence class of $q \in Q$ by $[\![q]\!]$. An abstracted automaton $\alpha^k(\mathcal{A}) = (Q_\alpha, \Sigma, \delta_\alpha, I_\alpha, F_\alpha)$ is obtained using α^k in the following way: $Q_\alpha = \{[\![q]\!] \mid q \in Q\}$, for each $q \in \delta(p, a)$ there is $([\![p]\!], a, X) \in \delta_\alpha$ such that $[\![q]\!] \in X$, and finally, $I_\alpha = \{[\![q]\!] \mid q \in I\}$ and $F_\alpha = \{[\![q]\!] \mid q \in F\}$.

The length abstraction overaprroximates language of the original automata, i.e., $L(\mathcal{A}) \subseteq L(\mathcal{A}_\alpha)$ meaning that there may exist a word $w = a_1, \ldots, a_n$ such that $w \in L(\mathcal{A}_\alpha) \wedge w \notin L(\mathcal{A})$. Both automata have the same alphabet which was originally derived from a set of events. Therefore, it is possible to convert the word w to series of actual messages. Supposing that w is not in the language of the original automaton, we thus obtain a series of events not present in the original system that can be used as a new test case for testing MES system in a digital twin.

7.5 Generating Scenario

Finally, we generate a scenario that will be used for orchestration of the digital twin of the tested system. We iterate over the event calendar and, for each event, we generate one step in the scenario. Each step consists of sending messages in the digital twin. The concrete messages sent during the orchestration are generated from the abstract representation. By default, we support exact replication of the seen communication, however, we provide also an experimental support for, e.g., generating syntactically or semantically similar messages.

We implemented our framework in our tool *Tyrant* [3] which generates scenarios for the orchestrating tool *Cryton*. Cryton uses as an input a configuration for creating the digital twin (i.e., a description of digital twin components) and a scenario generated by Tyrant in the YAML format.

In the following, we will illustrate the transformation of communication logs from real system to YAML scenario. We remark, we consider a system consisting of ERP system, MES system, and manufacturing machines and their corresponding terminals used by human operators. Listing 7.1 shows a message between ERP and MES. The message is stored in the file `20211207-125952.xml` which has a timestamp encoded in its name. The message is in XML format and its semantics is that there are 42 items of Material 1 in stocks. Listing 7.2 shows a message between a machine and MES. The message was sent one second after the previous one. The message semantics is that the value of the node 0 should be set to 99 in Machine 001. Finally, Listing 7.3 shows a generated scenario consisting of two steps. The first step is executed in (logical) time 0 hours, 0 minutes, 0 seconds: the orchestrator will send a message from ERP to MES using the REST protocol. The message has the XML data attached. The second steps are executed one second after the first step: the orchestrator will send another message from MES to Machine 001 using OPC-UA protocol. The message says that a value of the node with path 0 should be set to 99.

We implemented the proposed framework in the tool called the *Tyrant*. We tested our approach on the captured communication provided by a partner company that offers a MES

```
<DataSource>
 <Data>
   <Name>Material1</Name>
   <Value>42</Value>
 </Data>
</DataSource>
```

Listing 7.1: 20211207-125952.xml

```
SampleDataTime ; Value ; Name ; Path
2021-12-07 12:59:53.617 ; 99 ; Machine 001 ; 0
```

Listing 7.2: MES and Machine message

```
---
timestamps:
- delta:
    seconds: 0
    minutes: 0
    hours: 0
  steps:
  - type: ERP
    host: ERP
    target: MES
    args:
      xml: |-
        <DataSource>
          <Data>
            <Name>Data1</Name>
            <Value>42</Value>
          </Data>
        </DataSource>
- delta:
    seconds: 1
    minutes: 0
    hours: 0
  steps:
  - type: OPC-UA
    host: MES
    target: Machine 001
    args:
     value: 99
     node: 0
```

Listing 7.3: Generated scenario

Figure 7.2

as their product. We were able to successfully generate valid scenarios for the Cryton tool which then subsequently orchestrated a digital twin using our scenarios. However, our Tyrant is still prototype and it would need extensive collaboration with the company to deploy it to production.

7.6 Conclusion

In this work, we proposed a generic framework for orchestrating the digital twins of distributed systems in manufacturing environments. The main challenges of generating scenarios suitable for orchestrating digital twins is finding suitable models for (1) the communication in the systems, and (2) the actual sent messages during the communications. We proposed to use the simple approach: finite automata for communication and simple abstract representation for messages.

However, in our experience applying the framework in practice requires much more effort. Lots of testing scenarios and components require specific preparation before the orchestration: e.g., setting of the initial database or sending specific sequence of (hard-coded) messages to prepare the system that are usually not being captured in the communication log. Currently, our tool supports a concrete use case in a concrete manufacturing environment. Hence, extending our solution to a more broader class of manufacturing environments is our future work.

Chapter 8

Conclusion and Further Directions

The main achieved result in shape analysis is the application of backward run and counterexample-guided refinement (CEGAR) to shape analysis based forest automata. Backward run enables distinguishing between real and spurious (caused by abstraction) counterexamples. This made reporting of found bugs in program reliable and avoid unnecessary false alarms. The ability to detect spurious counterexamples is the first building block for CEGAR. The second one is application of predicate abstraction to domain of forest automata what makes precise refinement possible. Although the originally used height abstraction can be refined too, it does not guarantee exclusion of spurious counterexample and therefore does not guarantee termination of verification procedure. On the other hand, predicate abstraction derives the new predicates from a detected spurious counterexample what prevents reaching the counterexample again. This enables verification of a new class of data structures where a set of predicates (derived during CEGAR) is needed to describe their invariants. The concrete examples were described in Chapter 3.

The second result in shape analysis is the multiple participations of the Forester tool in software verification competition (SV-COMP). Although we did not win a medal in any major category we shown that our approach is able to compete and often superior to the other ones. Forester where able to soundly verify as the only tool in competition test cases such as skip list of 2nd and 3rd level, various trees, and also data structures with relations between nodes (for which we derive predicates using CEGAR).

The final result in automata-based shape analysis is introduction of automata over graphs with bounded tree width which are inspired by Courcelle's theorem [42]. These automata have expressivity higher than forest automata while keeping feasible algorithmic properties. Complexity of entailment between automata is singly exponential which we believe is suitable for application in practical tests cases once some heuristics will be introduced.

We also participated on new kind of shape analysis based on SMT solving. The possible shapes of data structures allocated on heap are described by logic formulae over bit vectors. Basically, it computes points-to relation between pointers used in program and abstract memory objects used by analyses to represent heap using SMT solver. This approach is more straightforward than the automata based ones and enables combination of shape analysis with other domains such as termination analysis, or domains for integer variables. On the other, it lacks generality of automata based approaches and currently supports only analysis of linked list data structures.

The second field of focus of the thesis was automated testing. We developed automated testing procedure for manufacturing execution systems (MES) in environment of digital

twins. We learn a model of communication protocol and messages sent between software components of a manufactory where MES is deployed. We automatically create test scenarios for digital twins using the learnt models where manufactory is emulated and MES deployed natively. Since we use finite automata to model communication protocol we can use abstraction to overapproximate language and create the new test cases. We can test e.g., a new version of MES before it is deployed to a real manufactory using the described method. The results of this part of the thesis are mainly from a field of applied research. We implemented the procedure in the tool Tyrant [3] which was later applied in Unis company.

8.1 Future Directions

The main potential in automata based shape analysis is in further development of graph automata over graphs with bounded tree width. We only sketched the basic concepts and it is needed to design and prove the algorithm for entailment between automata. Then one can think about using the formalism in a verification procedure such as regular model checking or abstract interpretation. Finally, it is needed to implement the concepts and evaluate it e.g., on SV-COMP benchmark. Eventually, design of heuristic for algorithm manipulating these automata may be an interesting research direction to make the verification procedure work for difficult test cases.

We would also like to apply collected experience in application of finite automata in a development of a new automata library. The library should provide an efficient methods for manipulation with finite automata. The crucial algorithms for formal verification and automated reasoning such as emptiness check or language inclusion checking will be implemented with various heuristics to make them computationally feasible. We want to keep library user friendly. The interface should be simple enough to be usable by anyone with basic programming skills and knowledge of automata theory. The source code should hide complicated heuristics and optimizations and keep basic data structures simple and easily extensible. The implementation should be in efficient language such as C++ and provide an interface to a high level programming language.

So far, we implemented the first prototype of library. It is in C++ and contains efficient representation of finite automata together with the algorithms over them such as union, intersection, complementation, determinisation, simulation based reduction, or language inclusion checking. We further extended the library by finite alternating automata representation. An interface for Python was implemented over the efficient implementation in C++. We design a text based format for description of various version of finite automata. The format support a finite automata and alternating finite automata with explicit, symbolic, or bitvector alphabet as well as transducers with mentioned alphabets. The library was already used for development of a new decision procedure for string solving.

In the field of automated testing, one may consider more domain specific abstractions over model of communication to enable generating more useful test automatically. There are often domain dependent particularities such as an order of messages sent in system during initialization or a special restriction of parameters of some messages. In current settings, we do not take that in consideration which may lead to generating scenarios for digital twin which fail early not due to a found bug in system under testing but because of e.g, wrong initialization of MES.

8.2 Publications Related to this Thesis

The work described in this thesis was published in the following papers. The backward run and counterexample-guided refinement was published in [65]. A paper describing the whole verification procedure based on forest automata was accepted for publication in book about state-of-the-art verification tool which should be published in year 2023. The papers related to FORESTER participation in SV-COMP are [64, 61, 62]. Graphs over automata with bounded tree width was sketched in [68]. The approach to shape analysis based on SMT was published in [86, 87].

During work on this thesis we participated also on another results related to the shape analysis which was not included in the thesis. It was connecting Symbiotic, a static analyser based on symbolic execution, with Predator, a successful shape analyser using symbolic memory graphs [36]. Predator won multiple editions of SV-COMP in memory safety related categories and was therefore a good candidate for providing information about memory safety invariants for each line of an analysed program. The information was further used by Symbiotic which relies heavily on slicing of program over which a symbolic execution is executed. Particularly, Symbiotic does not need to analyse lines of code which were denoted safe (from perspective of memory safety) by Predator executed in overapproximating (sound) mode. If Predator find a bug in program, Symbiotic takes a bug report and combines it with the results of its own static pointer analysis to filter out some spurious counterexamples. The viability of the approach was shown by the second place of Symbiotic and Predator combination in the memory safety category of SV-COMP 2018. However, the first place was taken by Predator Hunting Party, a tool suite of different versions of Predator (one in sound mode, several instances in bug hunting mode) running on GCC frontend which was more matured than LLVM fronted used in integration with Symbiotic. In the next editions of competition, the developers of both tools managed to make LLVM fronted for Predator as matured as GCC one what resulted to win of Symbiotic in the Memory Safety category.

The work on automated testing is accepted for publication in the proceedings of the conference EUROCAST 2022 which should be published during year 2023.

Bibliography

[1] *MarketsandMarkets Research Pvt. Ltd. „Industrial Control Systems (ICS) Security Market worth $22.2 billion by 2025“* [https://www.marketsandmarkets.com/PressReleases/industrial-control-systems-security-ics.asp, Last accessed 11. Nov 2022].

[2] *Repository of Cryton* [https://gitlabdev.ics.muni.cz/beast-public/cryton, Last accessed 11. Nov 2022].

[3] *Repository of Tyrant* [https://pajda.fit.vutbr.cz/tacr-unis/tyrant, Last accessed 11. Nov 2022].

[4] *Website of DTM Data Generator for JSON* [https://sqledit.com/jsongenerator/index.html, Last accessed 8. Sep 2022].

[5] *Website of JSON Schema TestData Generator* [https://www.npmjs.com/package/json-schema-test-data-generator, Last accessed 8. Sep 2022].

[6] *Forester*. July 2019. Available at: http://www.fit.vutbr.cz/research/groups/verifit/tools/forester/.

[7] Aalst, W. M. P. van der. *Process Mining - Data Science in Action, Second Edition*. Springer, 2016.

[8] Abdulla, P. A., Bouajjani, A., Holík, L., Kaati, L. and Vojnar, T. Computing Simulations over Tree Automata. In: *Proceedings of TACAS'08*. Springer, 2008, vol. 4963, p. 93–108. Lecture Notes on Computer Science.

[9] Abdulla, P. A., Bouajjani, A., Cederberg, J., Haziza, F. and Rezine, A. Monotonic Abstraction for Programs with Dynamic Memory Heaps. In: *CAV'08*. Springer, 2008, vol. 5123, p. 341–354. Lecture Notes on Computer Science.

[10] Abdulla, P. A., Chen, Y.-F., Holík, L., Mayr, R. and Vojnar, T. When Simulation Meets Antichains (on Checking Language Inclusion of NFAs). In: *Proceedings of TACAS'10*. Springer, 2010, vol. 6015, p. 158–174. Lecture Notes on Computer Science.

[11] Abdulla, P. A., Holík, L., Jonsson, B., Lengál, O., Trinh, C. Q. et al. Verification of heap manipulating programs with ordered data by extended forest automata. *Acta Informatica*. 2016, vol. 53, no. 4, p. 357–385.

[12] Ackermann, C. Recovering Views of Inter-System Interaction Behaviors. In: *Proceedings of Working Conference on Reverse Engineering 2009, Lille, France.* October 2009, p. 53–61.

[13] Albarghouthi, A., Berdine, J., Cook, B. and Kincaid, Z. Spatial Interpolants. In: *Proceedings of ESOP'15.* Springer, 2015, vol. 9032. Lecture Notes on Computer Science.

[14] Almeida, R., Holík, L. and Mayr, R. Reduction of Nondeterministic Tree Automata. In: *Proceedings of TACAS'16.* Springer, 2016, vol. 9636, p. 717–735. Lecture Notes on Computer Science.

[15] Appel, A. W. *Program Logics - for Certified Compilers.* Cambridge University Press, 2014.

[16] Berdine, J., Cox, A., Ishtiaq, S. and Wintersteiger, C. Diagnosing Abstraction Failure for Separation Logic-based Analyses. In: *Proceedings of CAV'12.* Springer, 2012, vol. 7358. Lecture Notes on Computer Science.

[17] Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P. W. et al. Shape Analysis for Composite Data Structures. In: *Proceedings of CAV'07.* Springer, 2007, vol. 4590, p. 178–192. Lecture Notes on Computer Science.

[18] Berdine, J., Cook, B. and Ishtiaq, S. SLAyer: Memory Safety for Systems-Level Code. In: *Proceedings of CAV'11.* Springer, 2011, vol. 6806, p. 178–183. Lecture Notes on Computer Science.

[19] Bertolino, A., Gao, J., Marchetti, E. and Polini, A. Automatic Test Data Generation for XML Schema-based Partition Testing. In: *Proceedings of AST '07.* 2007.

[20] Beschastnikh, I., Brun, Y., Ernst, M. D. and Krishnamurthy, A. Inferring models of concurrent systems from logs of their behavior with CSight. In: *Proceedings of ICSE '14.* ACM, 2014, p. 468–479.

[21] Beyer, D., Dangl, M., Dietsch, D. and Heizmann, M. Correctness Witnesses: Exchanging Verification Results Between Verifiers. In: *Proceedings of FSE '16.* ACM, 2016, p. 326–337.

[22] Beyer, D., Dangl, M., Dietsch, D., Heizmann, M. and Stahlbauer, A. Witness Validation and Stepwise Testification across Software Verifiers. In: *Proceedings of ESEC/FSE '15.* ACM, New York, 2015, p. 721–733.

[23] Beyer, D., Henzinger, T. A. and Théoduloz, G. Lazy Shape Analysis. In: *Proceedings of CAV'06.* Springer, 2006, vol. 4144, p. 532–546. Lecture Notes on Computer Science.

[24] Beyer, D., Löwe, S. and Wendler, P. Reliable Benchmarking: Requirements and Solutions. *STTT'19.* 2019, vol. 21, no. 1, p. 1–29.

[25] Blume, C., Bruggink, H. J. S., Engelke, D. and König, B. Efficient Symbolic Implementation of Graph Automata with Applications to Invariant Checking. In: *Proceedings of ICGT'12.* 2012.

[26] Botinčan, M., Dodds, M. and Magill, S. *Refining existential properties in separation logic analyses.* arXiv:1504.08309. 2015.

[27] Bouajjani, A., Habermehl, P., Moro, P. and Vojnar, T. Verifying Programs with Dynamic 1-Selector-Linked Structures in Regular Model Checking. In: *Proceedings of TACAS'05.* Springer, 2005, vol. 3440. Lecture Notes on Computer Science.

[28] Bouajjani, A., Habermehl, P., Rogalewicz, A. and Vojnar, T. Abstract Regular Tree Model Checking of Complex Dynamic Data Structures. In: *Proceedings of SAS'06.* Springer, 2006, vol. 4134. Lecture Notes on Computer Science.

[29] Bouajjani, A., Bozga, M., Habermehl, P., Iosif, R., Moro, P. et al. Programs with Lists are Counter Automata. *Formal Methods in System Design.* 2011, vol. 38, no. 2, p. 158–192.

[30] Bouajjani, A., Drăgoi, C., Enea, C. and Sighireanu, M. Accurate Invariant Checking for Programs Manipulating Lists and Arrays with Infinite Data. In: *Proceedings of ATVA'12.* Springer, 2012, vol. 7561, p. 167–182. Lecture Notes on Computer Science.

[31] Bouajjani, A., Habermehl, P., Rogalewicz, A. and Vojnar, T. Abstract Regular Tree Model Checking. *Electronic Notes in Theoretical Computer Science.* 2006, vol. 149, no. 1, p. 37–48.

[32] Bouajjani, A., Habermehl, P., Rogalewicz, A. and Vojnar, T. Abstract regular (tree) model checking. *STTT.* 2012, vol. 14, no. 2, p. 167–191.

[33] Brandenburg, F. J. and Skodinis, K. Finite Graph Automata for Linear and Boundary Graph Languages. *Theoretical Computer Science.* 2005, vol. 332, 1-3.

[34] Brookes, S. and O'Hearn, P. W. Concurrent Separation Logic. *ACM SIGLOG News.* ACM. 2016, vol. 3, no. 3, p. 47–65.

[35] Calcagno, C., Distefano, D., O'Hearn, P. W. and Yang, H. Compositional Shape Analysis by Means of Bi-Abduction. *Journal of ACM.* december 2011, vol. 58, no. 6.

[36] Chalupa, M., Jašek, T., Tomovic, L., Hruška, M., Šoková, V. et al. Symbiotic 7: Integration of Predator and More - (Competition Contribution). In: *Proceedings of TACAS'20.* Springer, 2020, vol. 12079, p. 413–417. Lecture Notes on Computer Science.

[37] Chang, B.-Y. E., Rival, X. and Necula, G. C. Shape Analysis with Structural Invariant Checkers. In: *Proceedings of SAS'07.* Springer, 2007, vol. 4634, p. 384–401. Lecture Notes on Computer Science.

[38] Clarke, E., Grumberg, O., Jha, S., Lu, Y. and Veith, H. Counterexample-Guided Abstraction Refinement. In: *Proceedings of CAV'00.* Springer, 2000, p. 154–169.

[39] Comon, H., Dauchet, M., Gilleron, R., Löding, C., Jacquemard, F. et al. *Tree Automata Techniques and Applications.* 2008.

[40] Comparetti, P. M., Wondracek, G., Krügel, C. and Kirda, E. Prospex: Protocol Specification Extraction. In: *Proceedings of S&P 2009*. IEEE Computer Society, 2009, p. 110–125.

[41] Courcelle, B. The monadic second-order logic of graphs. I. Recognizable sets of finite graphs. *Information and Computation*. march 1990, vol. 85, p. 12–75.

[42] Courcelle, B. and Engelfriet, J. *Graph Structure and Monadic Second-Order Logic: A Language-Theoretic Approach*. 1stth ed. Cambridge University Press, 2012.

[43] Cousot, P. and Cousot, R. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In: *Proceedings of POPL'77*. ACM, 1977, p. 238–252.

[44] Demri, S., Lozes, É. and Mansutti, A. The Effects of Adding Reachability Predicates in Propositional Separation Logic. In: *Proceedings of FoSSaCS'18*. Springer, 2018, vol. 10803, p. 476–493. Lecture Notes on Computer Science.

[45] Deshmukh, J., Emerson, E. and Gupta, P. Automatic Verification of Parameterized Data Structures. In: *Proceedings of TACAS'06*. Springer, 2006, vol. 3920. Lecture Notes on Computer Science.

[46] Dudka, K., Peringer, P. and Vojnar, T. Predator: A Practical Tool for Checking Manipulation of Dynamic Data Structures Using Separation Logic. In: *Proceedings of CAV'11*. Springer, 2011, vol. 6806, p. 372–378. Lecture Notes on Computer Science.

[47] Dudka, K., Peringer, P. and Vojnar, T. An Easy to Use Infrastructure for Building Static Analysis Tools. In: *Proceedings of Computer Aided Systems Theory – EUROCAST 2011*. Berlin, Heidelberg: Springer, 2012, p. 527–534.

[48] Dudka, K., Peringer, P. and Vojnar, T. Byte-Precise Verification of Low-Level List Manipulation. In: *Proceedings of SAS'13*. Springer, 2013, vol. 7935, p. 215–237. Lecture Notes on Computer Science.

[49] Echenim, M., Iosif, R. and Peltier, N. Entailment Checking in Separation Logic with Inductive Definitions is 2-EXPTIME hard. In: *Proceedings of LPAR 2020*. EasyChair, 2020, vol. 73, p. 191–211. EPiC Series in Computing.

[50] Echenim, M., Iosif, R. and Peltier, N. Decidable Entailments in Separation Logic with Inductive Definitions: Beyond Establishment. In: *Proceedings of CSL' 21*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, vol. 183, p. 20:1–20:18. LIPIcs.

[51] Echenim, M., Iosif, R. and Peltier, N. Unifying Decidable Entailments in Separation Logic with Inductive Definitions. In: *Proceedings of CADE'21*. Springer, 2021, vol. 12699. Lecture Notes on Computer Science.

[52] Enea, C., Lengál, O., Sighireanu, M. and Vojnar, T. Compositional Entailment Checking for a Fragment of Separation Logic. In: *Proceedings of APLAS'14*. Springer, 2014, vol. 8858, p. 314–333. Lecture Notes on Computer Science.

[53] Guo, B., Vachharajani, N. and August, D. I. Shape Analysis with Inductive Recursion Synthesis. In: *Proceedings of PLDI'07*. June 2007, vol. 42, no. 6.

[54] Habermehl, P., Holík, L., Rogalewicz, A., Šimáček, J. and Vojnar, T. Forest Automata for Verification of Heap Manipulation. In: *Proceedings of CAV'11*. Springer, 2011, vol. 6806, p. 424–440. Lecture Notes on Computer Science.

[55] Habermehl, P., Holík, L., Hruška, M., Lengál, O., Rogalewicz, A. et al. *Forester*. July 2019. Available at: http://www.fit.vutbr.cz/research/groups/verifit/tools/forester/.

[56] Habermehl, P., Holík, L., Hruška, M., Lengál, O., Rogalewicz, A. et al. *Forester Virtual Machine*. July 2019. DOI: 10.5281/zenodo.3339213. Available at: https://zenodo.org/record/3339213.

[57] Habermehl, P., Holík, L., Rogalewicz, A., Šimáček, J. and Vojnar, T. Forest Automata for Verification of Heap Manipulation. *Formal Methods in System Design*. Springer. 2012, vol. 41, no. 1, p. 83–106.

[58] Heinen, J., Jansen, C., Katoen, J. and Noll, T. Juggrnaut: Using graph grammars for abstracting unbounded heap structures. *Formal Methods in System Design*. 2015, vol. 47, no. 2.

[59] Heinen, J., Noll, T. and Rieger, S. Juggrnaut: Graph Grammar Abstraction for Unbounded Heap Structures. In: *Proceedings of TTSS'09*. Elsevier, 2010, vol. 266, p. 93–107. ENTCS.

[60] Holík, L., Lengál, O., Rogalewicz, A., Šimáček, J. and Vojnar, T. *Fully Automated Shape Analysis Based on Forest Automata*. FIT-TR-2013-01. FIT BUT, 2013.

[61] Holík, L., Hruška, M., Lengál, O., Rogalewicz, A., Šimáček, J. et al. Run Forester, Run Backwards! (Competition Contribution). In: *Proceedings of TACAS'16*. Springer, 2016, vol. 9636, p. 923–926. Lecture Notes on Computer Science.

[62] Holík, L., Hruška, M., Lengál, O., Rogalewicz, A., Šimáček, J. et al. Forester: From Heap Shapes to Automata Predicates (Competition Contribution). In: *Proceedings of TACAS'17*. Springer, 2017, vol. 10206, p. 365–369. Lecture Notes on Computer Science.

[63] Holík, L., Peringer, P., Rogalewicz, A., Šoková, V., Vojnar, T. et al. Low-Level Bi-Abduction. In: *Proceedings of ECOOP'22*. Schloss Dagstuhl, 2022, vol. 222, p. 19:1–19:30. LIPIcs.

[64] Holík, L., Hruška, M., Lengál, O., Rogalewicz, A., Šimáček, J. et al. Forester: Shape Analysis Using Tree Automata (Competition Contribution). In: *Proceedings of TACAS'15*. Springer Verlag, 2015, vol. 9035, p. 432–435. Lecture Notes on Computer Science. ISBN 978-3-662-46680-3.

[65] Holík, L., Lengál, O., Rogalewicz, A., Hruška, M. and Vojnar, T. Counterexample Validation and Interpolation-Based Refinement for Forest

Automata. In: *Proceedings of VMCAI'17*. Springer, 2017, vol. 10145, p. 288–309. Lecture Notes on Computer Science.

[66] Holík, L., Lengál, O., Rogalewicz, A., Šimáček, J. and Vojnar, T. Fully Automated Shape Analysis Based on Forest Automata. In: *Proceedings of CAV'13*. Springer, 2013, vol. 8044, p. 740–755. Lecture Notes on Computer Science.

[67] Holík, L., Lengál, O., Rogalewicz, A., Šimáček, J. and Vojnar, T. Fully Automated Shape Analysis Based on Forest Automata. In: *Proceedings of CAV'13*. Springer, 2013, vol. 8044, p. 740–755. Lecture Notes on Computer Science.

[68] Hruška, M. and Holík, L. Towards Efficient Shape Analysis with Tree Automata. In: *Proceedings of NETYS '21*. Springer, 2021, vol. 12754, p. 206–214. Lecture Notes on Computer Science.

[69] Hsu, Y., Shu, G. and Lee, D. A model-based approach to security flaw detection of network protocol implementations. In: *Proceedings of ICNP 2008*. IEEE Computer Society, 2008, p. 114–123.

[70] Iosif, R., Rogalewicz, A. and Vojnar, T. Deciding Entailments in Inductive Separation Logic with Tree Automata. In: *Proceedings of ATVA'14*. Springer, 2014, vol. 8837, p. 201–218. Lecture Notes on Computer Science.

[71] Iosif, R., Rogalewicz, A. and Šimáček, J. The Tree Width of Separation Logic with Recursive Definitions. In: *Proceedings of CADE'13*. Springer, 2013, vol. 7898, p. 21–38. Lecture Notes on Computer Science.

[72] Jensen, J. L., Jørgensen, M. E., Klarlund, N. and Schwartzbach, M. I. Automatic Verification of Pointer Programs using Monadic Second-Order Logic. In: *Proceedings of PLDI '97*. ACM Trans. Comput. Log., 1997, p. 226–236.

[73] Jhala, R. and McMillan, K. Lazy Abstraction with Interpolants. In: *Proceedings of CAV'06*. Springer, 2006, vol. 4144. Lecture Notes on Computer Science.

[74] Katelaan, J., Matheja, C. and Zuleger, F. Effective Entailment Checking for Separation Logic with Inductive Definitions. In: *Proceedings of TACAS '19*. Springer, 2019, vol. 11428, p. 319–336. Lecture Notes on Computer Science.

[75] Katelaan, J. and Zuleger, F. Beyond Symbolic Heaps: Deciding Separation Logic With Inductive Definitions. In: *Proceedings of LPAR'11*. EasyChair, 2020, vol. 73. EPiC Series in Computing.

[76] La Riva, C. D., Garcia Fanjul, J. and Tuya, J. A Partition-Based Approach for XPath Testing. In: *Proceedings of ICSEA'06*. 2006.

[77] Le, Q. L. Compositional Satisfiability Solving in Separation Logic. In: *Proceedings of VMCAI'21*. Springer, 2021, vol. 12597. Lecture Notes on Computer Science.

[78] Le, Q. L., Gherghina, C., Qin, S. and Chin, W. Shape Analysis via Second-Order Bi-Abduction. In: *Proceedings of CAV'14*. Springer, 2014, vol. 8559, p. 52–68. Lecture Notes on Computer Science.

[79] Lee, O., Yang, H. and Petersen, R. Program Analysis for Overlaid Data Structures. In: *Proceedings of CAV'11*. Springer, 2011, vol. 6806, p. 592–608. Lecture Notes on Computer Science.

[80] Leemans, M. and Aalst, W. M. P. van der. Process Mining in Software Systems: Discovering real-life business transactions and process models from distributed systems. In: *Proceedings of MODELS'15*. IEEE, 2015, p. 44–53.

[81] Lengál, O., Šimáček, J. and Vojnar, T. VATA: A Library for Efficient Manipulation of Non-deterministic Tree Automata. In: *Proceedings of TACAS'12*. Springer, 2012, p. 79–94.

[82] Loginov, A., Reps, T. and Sagiv, M. Abstraction Refinement via Inductive Learning. In: *Proceedings of CAV'05*. Springer, 2005, vol. 3576, p. 519–533. Lecture Notes on Computer Science.

[83] Madhusudan, P. and Parlato, G. The Tree Width of Auxiliary Storage. In: *Proceedings of POPL'11*. 2011.

[84] Madhusudan, P., Parlato, G. and Qiu, X. Decidable Logics Combining Heap Structures and Data. *ACM SIGPLAN Notices*. january 2011, vol. 46, no. 1.

[85] Magill, S., Tsai, M.-H., Lee, P. and Tsay, Y.-K. Automatic Numeric Abstractions for Heap-manipulating Programs. In: *Proceedings of POPL'10*. ACM, 2010, p. 211–222.

[86] Malík, V., Hruška, M., Schrammel, P. and Vojnar, T. Template-Based Verification of Heap-Manipulating Programs. In: *Proceedings of FMCAD'18*. IEEE, 2018, p. 1–9.

[87] Malík, V., Hruška, M., Schrammel, P. and Vojnar, T. 2LS: Heap Analysis and Memory Safety (Competition Contribution). *CoRR*. 2019, abs/1903.00712. Available at: http://arxiv.org/abs/1903.00712.

[88] Matheja, C., Jansen, C. and Noll, T. Tree-Like Grammars and Separation Logic. In: *Proceedings of APLAS'15*. Springer, 2015, vol. 9458, p. 90–108. Lecture Notes on Computer Science.

[89] Matousek, P., Havlena, V. and Holík, L. Efficient Modelling of ICS Communication For Anomaly Detection Using Probabilistic Automata. In: *Proceedings of IM'21*. IEEE, 2021, p. 81–89.

[90] McMillan, K. L. Interpolation and SAT-Based Model Checking. In: *Proceedings of CAV'03*. Springer, 2003, vol. 2725, p. 1–13. Lecture Notes on Computer Science.

[91] McPeak, S. and Necula, G. C. Data Structure Specifications via Local Equality Axioms. In: *Proceedings of CAV'05*. Springer, 2005, vol. 3576, p. 476–490. Lecture Notes on Computer Science.

[92] Møller, A. and Schwartzbach, M. I. The Pointer Assertion Logic Engine. *ACM SIGPLAN Notices*. may 2001, vol. 36, no. 5.

[93] Nguyen, H. H., David, C., Qin, S. and Chin, W.-N. Automated Verification of Shape and Size Properties Via Separation Logic. In: *Proceedings of VMCAI'07*. Springer, 2007, vol. 4349, p. 251–266. Lecture Notes on Computer Science. ISBN 978-3-540-69735-0.

[94] Ostrand, T. J. and Balcer, M. J. The Category-Partition Method for Specifying and Generating Fuctional Tests. *Communications of ACM*. New York, NY, USA: Association for Computing Machinery. 1988, vol. 31, no. 6.

[95] Pagel, J., Matheja, C. and Zuleger, F. *Complete Entailment Checking for Separation Logic with Inductive Definitions*. 2020.

[96] Pagel, J. and Zuleger, F. Strong-Separation Logic. In: *Proceedings of ESOP'21*. Springer, 2021, vol. 12648, p. 664–692. Lecture Notes on Computer Science.

[97] Piskac, R., Wies, T. and Zufferey, D. Automating Separation Logic with Trees and Data. In: *Proceedings of CAV'14*. Springer, 2014, vol. 8559, p. 711–728. Lecture Notes on Computer Science.

[98] Podelski, A. and Wies, T. Counterexample-Guided Focus. In: *Proceedings of POPL'10*. ACM, 2010, p. 249–260.

[99] Qin, S., He, G., Luo, C., Chin, W.-N. and Chen, X. Loop Invariant Synthesis in a Combined Abstract Domain. *Journal of Symbolic Computation*. Elsevier. 2013, vol. 50, p. 386–408.

[100] Raad, A., Berdine, J., Dang, H., Dreyer, D., O'Hearn, P. W. et al. Local Reasoning About the Presence of Bugs: Incorrectness Separation Logic. In: *Proceedings of CAV'20*. Springer, 2020, vol. 12225, p. 225–252. Lecture Notes on Computer Science.

[101] Rabin, M. O. and Scott, D. Finite Automata and Their Decision Problems. *IBM Journal of Research and Development*. 1959, vol. 3, no. 2, p. 114–125.

[102] Reiter, F. Distributed Graph Automata. In: *Proceedings of LICS'15*. 2015.

[103] Reynolds, J. C. Separation logic: a logic for shared mutable data structures. In: *Proceedings of LICS'02*. 2002, p. 55–74.

[104] Sagiv, S., Reps, T. W. and Wilhelm, R. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*. 2002, vol. 24, no. 3, p. 217–298.

[105] Schrammel, P. and Kroening, D. 2LS for Program Analysis. In: *Proceeding of TACAS'16*. Berlin, Heidelberg: Springer-Verlag, 2016, p. 905–907. ISBN 9783662496732.

[106] Thomas, W. On logics, tilings, and automata. In: *Automata, Languages and Programming*. Springer, 1991, vol. 510, p. 441–454. Lecture Notes on Computer Science. ISBN 978-3-540-54233-9.

[107] Šimáček, J. *Harnessing Forest Automata for Verification of Heap Manipulating Programs*. 2012. Dissertation. FIT BUT.

[108] Web Pages of the Facebook Infer. *The Facebook Infer* [`http://fbinfer.com/`]. 2017 [cit. 2022-08-24].

[109] Weinert, A. D. *Inferring Heap Abstraction Grammars*. 2012. BSc thesis. RWTH Aachen.

[110] Yang, H., Lee, O., Berdine, J., Calcagno, C., Cook, B. et al. Scalable Shape Analysis for Systems Code. In: *Proceedings of CAV'08*. Springer, 2008, vol. 5123, p. 385–398. Lecture Notes on Computer Science.

[111] Zee, K., Kuncak, V. and Rinard, M. C. Full functional verification of linked data structures. In: *Proceedings of PLDI'08*. ACM, 2008, p. 349–361.