Brno University of Technology
Faculty of Information Technology

# Verification of Programs with Complex Data Structures

Ph.D Thesis

Adam Rogalewicz

August 1, 2007

## Abstract

In this thesis, we discuss methods of model checking of infinite-state space systems based on symbolic verification—in particular, we concentrate on the use of the so-called regular tree model checking. As a part of our original contribution, we first present abstract regular tree model checking (ARTMC), a technique based on a combination of regular tree model checking with an automated abstraction using the counter-example guided abstraction refinement principle. Then, we present our original method for verification of safety properties of pointer-manipulating procedures. The method uses ARTMC as a verification framework. The method was successfully tested on a set of real-life procedures manipulating dynamic data structures (such as linked lists, doubly linked lists, trees, etc.)—some of these procedures were handled fully automatically for the first time using our approach. Finally, we present our original fully automated method for termination proofs for programs manipulating tree-like data structures. The method is based on a combination of ARTMC with counter automata and it was successfully applied on several tree manipulating procedures.

## Keywords

Formal verification, model checking, infinite state-space systems, regular tree languages, automated abstraction, shape analysis, termination checking.

## Acknowledgement

Above all, I would like to thank to my Ph.D. advisor doc. Tomáš Vojnar for an excellent cooperation in the research leading to the results presented in this thesis and for his valuable suggestions during the writing of the thesis itself. Then I would like to thank to prof. Milan Češka, the head of my research group at FIT BUT, for professional, moral and financial support. And finally, I would like to thank to my foreign research partners for a strong cooperation on the results presented in this thesis—in particular, prof. Ahmed Bouajjani and dr. Peter Habermehl from LIAFA, Université Paris 7—Denis Diderot/CNRS and dr. Radu Iosif from VERIMAG, Université Joseph Fourier/CNRS/INPG, Grenoble.

*The work presented in this thesis was supported by the Czech Grant Agency within the projects 102/05/H050, 102/04/0780, 102/03/D211, and by the Czech-French project Barrande 2-06-27.*

# Contents

# Chapter 1

# Introduction

In the last several years, computer systems are being used in more and more areas of every day life, and a lot of present human activities rely on the bug-free run of these systems. Therefore a failure of such a system can cause huge economical, ecological, or even life losses. The losses may be not necessarily direct—the failure can also cause a loss of credit for software or hardware companies. At the same time, computer systems are more and more complex and their complexity is still rising. Together with the rising complexity, the probability of a fatal error of such systems is rising too. A thorough testing is very expensive, and moreover, it is not possible to test effectively all possible behaviors of the system in all possible situations. A fatal error can occur e.g. after two years of a *bug-free* use of a system. Therefore alternative methods for ensuring reliability of systems are quite welcome.

*Formal verification* is a possible solution to guarantee the reliability of computer systems. Methods of formal verification are designed to give a proof that a system corresponds to a given specification. The research in the area of formal verification is nowadays quite live, and the existence of huge research groups in companies such as e.g. Microsoft, Intel, IBM, NASA, or Airbus, working in cooperation with prestigious universities, illustrates the importance of these methods.

## 1.1 Formal Verification

*Formal verification* is a set of methods based on formal mathematical roots designed to prove, or disprove, whether a computer system conforms to specified properties. In case of a disproof, it is welcome if the method can provide an error trace leading to a feasible false behaviour of the system being verified.

Each method of formal verification can be basically classified from the point of view of two criteria: *soundness* and *completeness*. Soundness of a method is a capability to detect all possible errors according to the specification—a method is *sound*, if the positive answer guarantees the correctness of the system (according to the given specification). Completeness of a method is an ability to provide true counterexamples—a *complete* method guarantees that the error trace that is produced really leads to a wrong behaviour of the computer system.

The most popular methods in formal verification can be classified into several categories such as theorem proving, model checking, and static analysis. An often cited framework of abstract interpretation can be included in the area of static analysis. The borders between these categories are not strict and many approaches are hard to be categorized. In the following part of this section, we provide a very basic overview of these categories to provide the reader with the context of the presented work.

**Theorem proving.** *Theorem proving* is an approach of formal verification close to the classical mathematical reasoning. It uses some inference system to deduce theorems about the system being verified from the facts known about the system and from generally valid theorems of logic, mathematics, etc. The proof of the system is supported by computer-aided tools—so-called *theorem provers*. The most known theorem provers are e.g. PVS [OSRSC01], Isabelle [NPW05], or ACL2 [KMe00b, KMe00a]. The technique is quite general, but often only semi-automated and hard to use. Also in case of the verified system fails to satisfy the given specification, it is often quite complicated to get a diagnostic information to repair the system. It can also be hard to distinguish, whether there is a real bug in the system, or the method is used in a wrong fashion. Nevertheless, in the last years, there has also been a huge development in the area of fully automated theorem proving techniques. These techniques are often combined with some other techniques, e.g. model checking. As an example of this combination, we can mention methods of *automated abstraction* (e.g. predicate abstraction [GS97]).

**Model checking.** *Model checking* [CGP99] is an automated approach designed to prove that a model of a computer system (sometimes the model is equal to the system itself) satisfies some given properties via a systematic exploration of the state space of the model. The first mentions about model checking can be found in works [CE81, QS82]. The required properties can be specified e.g. in some temporal logics as LTL [Pnu77], CTL [CE81], CTL$^*$ [EH86], $\mu$-calculus [Koz83], or there can be also used some simpler specification such as C-like assertions. An important advantage of model checking is its high degree of automation and the ability to provide diagnostic information in case of a model fails to satisfy the given specification. On the other hand, there is an exponential state space explosion (with respect to the size of the examined model), which can make the model checking approach useless for bigger systems.

Many heuristic methods were proposed to cope with the state space explosion problem [Val98, CGP99]. One of them is, e.g., *symbolic verification* which is based on dealing with efficiently encoded sets of configurations instead of single states. *Binary-decision diagrams* [Bry86, BCM$^+$92], which are used in many successful model checking tools, are the most successful currently known symbolic representation.

Another possibility how to fight with the state space explosion is *state space reduction*. Instead of the whole state space, just a part is generated—one does not examine configurations for which the computation clearly cannot break the verified properties. These methods are, e.g., based on symmetries [CFJ93, ID96, SMG97], or partial order reductions [Val88, KP88, God91]. Another possibility is *on-the-fly* model checking where the properties are checked in parallel with the state space generation. This allows to stop the computation immediately after an error configuration is discovered.

Another way to improve capabilities of model checking resides in its combination with principles such as *modular verification* [Pnu89, CLM89, CCST05, AMN05] or *automated abstraction* [CGL94, GS97, BLO98, CGJ$^+$00, CCG$^+$04, HJMS03]. A lot of successful software model checkers is based on automated abstraction, where instead of exploration of the whole state space, one just explores its abstraction. A particularly successful abstraction method is the so-called *predicate abstraction* [GS97]. The abstractions are usually constructed in such a way that a positive verification answer for the abstract model implies the positive answer for the original system. On the other hand, the negative answer can be caused by a too rough abstraction. One then needs to check, whether the obtained counterexample is feasible in the original system, or whether it is a *spurious counterexample* caused by the abstraction. In case of a spurious counterexample, the applied abstraction must be refined. The refinement is often directed by the encountered spurious counterexample—this approach is called *Counter-Example Guided Abstraction Refinement* (CEGAR) [CGJ$^+$00].

**Static Analysis.** Methods of *static analysis* try to find bugs in computer systems without directly exploring their state spaces (or state spaces of their models). These methods are usually based on intelligent source code browsing and searching for *dangerous bug patterns*. The power of the various existing static analysis methods is very different—one can find simple syntactic checks on the one hand and complex methods exploiting abstract fixpoint computations on the other hand. Methods of static analysis are not used only to prove the correctness of a system, but they are used, e.g., within compilers, or program optimizers. The methods are often highly specialized for certain types of bugs.

A number of static analysis methods are based on *abstract interpretation*, a quite general theory of a sound approximation of computer programs. It was originally introduced in the works [CC77, CC79, Cou81, Deu92]. In abstract interpretation, there are defined abstraction and concretization functions linking a program into its abstract semantics. In order to handle loops and recursion, a fixpoint computation is used. In case of infinite abstract domains, widening is used to make the fixpoint computation terminating. The framework of abstract interpretation is quite flexible, so the methods based on can include simple syntactic checks on the one side and full model checking methods on the other side.

In comparison with model checking, the main advantage of static analysis resides in the fact that it is capable to handle very huge amounts of code and a model of the environment of the examined system is (usually) not needed. A model of the environment (and usually also of a part of the system being examined) is usually needed in the case of model checking, and this need makes model checking sometimes quite expensive. Also, some bugs can be hidden and some other spurious ones introduced due to an imprecise model [EM04].

On the other hand, due to the overapproximation used, static analysis can lead to a number of false alarms, i.e., it may report suspicious behaviour of the given system which, however, needs not be really erroneous. It is then up to the user to handle such situations. For the static analysis methods based on looking for syntactic patterns, it can sometimes be difficult (or even impossible) to specify all possible *syntactic patterns* leading to an error behaviour in order to use such analyzes. An impropriate pattern used can then lead to false alarms or to omitting of some bugs.

There exists a number of works trying to increase preciseness of static analysis by remembering more and more information about values of system variables. These methods are, however, getting closer and closer to model checking with its advantages, but also disadvantages.

## 1.2 Model Checking of Infinite State Space Systems

In practice, we are often faced with systems, which have infinite state spaces. The infinity can be caused by, e.g., dealing with real time, unbounded data structures (stacks, queues, trees, etc.), or a structure of program depending on parameters (e.g., an unbounded number of cooperating processes). But most of the so-far developed methods of model checking consider large, but finite-state systems. This is not true for methods based on theorem proving, or static analysis, which do not usually make difference between finite and infinite-state spaces.

In general, the problem of model checking for infinite state systems is undecidable. Therefore model checking methods dealing with such systems are usually not fully automated, semi-algorithmic (the termination of the method is not guaranteed), or partial (with a possible *don't know* answer). However, there exist some classes of infinite-state systems for which the model checking problem is decidable. The decidability was proved for model checking of, e.g., the full modal $\mu$-calculus over *push-down systems* [Wal96, BS97], a lot of problems over *lossy channel systems* [Fin94, CFI96, AJ01, MS02], many verification problems on *timed automata* [AD94, ACD93], etc. But due to the complexity issues, it is sometimes better to use some heuristic method also for the systems with a decidable model checking problem.

General methods of model checking for infinite state systems are usually based on one of the following principles. Methods of *automated abstractions* [GS97, BBLS00, BLBS01] are used for translating a concrete infinite-state system to an abstract one. The abstraction is usually required to produce an abstract system that has a finite state space, and whose behaviour is an over-approximation of the behaviour of the original one. A lot of existing software verification methods (e.g., [HJMS03, CCG$^+$04]) is based on automated abstractions—concretely predicate abstraction [GS97]. Automated abstractions can also be used for translation of an infinite-state system to another infinite-state system, but with a simplified structure of states. After the abstraction, another infinite-state model checking method can be applied (e.g., symbolic model checking).

Another group of approaches is based on *automated induction*—e.g. [WL89, KM95, MQS00, LHR97, CR00, PRZ01]. A lot of them is based on the so-called *network invariants*. A network invariant $I$ is an abstraction of $n$ parallel processes, which is independent of the parameter $n$. One uses model checking to verify whether the network invariant $I$ covers a single process, whether a composition of a single process with the invariant $I$ is covered by the invariant $I$, and whether the invariant $I$ satisfies given properties.

Further, methods of the so-called *cut-offs* [GS92, EN96] can be used for verifying parametric systems. They are based on searching bounds of parameters appearing in the system. Increasing the parameters over these bounds has no influence on the system behaviour from the point of view of the verified properties. If it is possible to find such bounds, and they are not too high, than this method will be very useful as it translates infinite-state model checking to finite-state one (and thus allows to exploit the well-established methods of finite-state model checking).

Finally, methods of *symbolic model checking* (mentioned already in connection with handling large finite-state spaces) [KMM$^+$01, BJNT00] can be used to describe an infinite state space in a finite way, thus allowing their manipulations by a computer. The behaviour of a system is then represented by operations over this representation. A finite representation of infinite sets of reachable configurations can be based on automata, logics, grammars, etc. For example, one of the most successful symbolic model checking approaches is based on the so-call *zones* [Dil89, HNSY94] and it is used in verification of real-time systems modeled by timed automata [AD94, HNSY94]. Other approaches of symbolic model checking include, e.g., regular model checking [KMM$^+$01, BJNT00] (a technique based on finite automata) and its generalization to finite tree automata [BT02, AJMd02, Sha02] which is the approach whose development is currently a part of our original research presented in the following chapters. The advantage of symbolic model checking methods resides in the fact that they are usually fully automated and quite generic.

## 1.3 Goals and Contributions of the Thesis

The main direction of research presented in this thesis is a development of new verification methods for infinite state-space systems and increasing efficiency of the already existing ones. Especially, we are interested in methods based on the symbolic model checking. The main goals of this thesis are as follows:

- Generalization of *abstract regular model checking* (ARMC), originally designed for verification of systems with a linear topology of states, to systems with a more general topology of states (such as the tree topology).
- Developing methods for verification of programs manipulating dynamic data structures using the generalized ARMC.
- A prototype implementation and evaluation of the proposed methods.

The first part of our contribution is the generic symbolic model checking method *abstract regular tree model checking* (ARTMC) [BHRV05, BHRV06a]. This method is a combination of regular tree model checking and automated abstraction—it is a generalization of *abstract regular model checking* [BHV04] to tree languages. Our prototype implementation of ARTMC was successfully applied to several examples from the area of parameterized networks of processes. The method far outperformed other existing approaches to regular tree model checking. Further, we have proposed a technique, how to use ARTMC for model checking of safety properties of programs manipulating dynamic data structures [BHRV06b, Rog06]. This method was implemented and successfully applied on a set of case studies from the area of singly and doubly linked lists, lists of lists, and trees. The method belongs among the most general, fully automated existing approaches. Some programs were for the first time fully automatically verified using this approach. Finally, we propose an original technique based on ARTMC and counter automata for checking termination of programs manipulating trees [HIRV07a, HIRV07b]. The capabilities of the method are demonstrated on several examples of tree manipulating programs. To the best of our knowledge, we are not aware of any other generic and fully automated method designed to prove termination of programs manipulating tree-like data structures.

Most of the work presented here was done in a strong and fruitful cooperation with my colleagues from Brno University of Technology and foreign partner institutions. In particular Tomáš Vojnar from FIT, Brno University of Technology, Ahmed Bouajjani and Peter Habermehl from LIAFA, Université Paris 7—Denis Diderot/CNRS and Radu Iosif from VERIMAG, Université Joseph Fourier/CNRS/INPG, Grenoble. The work is based on extended versions of papers originally published at renowned conferences. Our original results are accompanied by a discussion of other existing approaches to the concerned area.

We especially include a more detailed description of *regular model checking*, which is not a part of our contribution, into this thesis. The description is done with a strong emphasis to abstract regular model checking which presents the starting point of the research leading to all our original results. Some of the proofs, which are not a part of our original contribution, but which are directly linked to our original work, were moved to the appendix of the thesis.

## 1.4 Structure of the Thesis

The rest of the thesis is structured as follows: In Chapter 2, we present an overview of *regular model checking*—the starting point of our research. In the following chapters, we always first introduce the specific area that the respective chapter concentrates on and we discuss the various approaches existing in the given area. Then we present our original results in the respective area. In the summary of each chapter, we discuss possible extensions of the proposed approaches. In particular, in Chapter 3, we describe our generic verification technique of *abstract regular tree model checking*. In Chapter 4, we present our original encoding of programs manipulating dynamic data structures into the framework of ARTMC and its successful application to some real-life pointer-intensive procedures. In Chapter 5, we describe our technique for termination checking of programs manipulating trees. Finally, we close the thesis by some conclusion remarks in Chapter 6.

# Chapter 2

# Regular Model Checking

*Regular model checking* [KMM$^+$01, WB98, BJNT00] is a generic method for formal verification of infinite-state systems. Configurations of systems are encoded as finite words over a finite alphabet $\Sigma$ and transitions are encoded as regular relations over words. Then, finite word automata over $\Sigma$ can naturally be used to represent and manipulate (infinite) sets of configurations and transducers over $(\Sigma \cup \{\varepsilon\}) \times (\Sigma \cup \{\varepsilon\})$ are used to represent the transition relation.[1] To verify safety properties, a reachability analysis is performed by calculating transitive closures of transducers or images of automata by iteration of transducers. Termination of this computation is usually not guaranteed even with the acceleration: If the reachability computation is implemented as a simple iterative computation as in finite-state systems, it will usually not stop. That is why various acceleration methods have been proposed. Thise methods include extrapolation [BJNT00], quotienting [AdJN03], acceleration schemas [PS00], inference of regular languages [FO97] or abstraction of automata [BHV04].

This chapter presents an overview of regular model checking with a strong emphasis on its implementation based on abstraction of automata – *abstract regular model checking* (ARMC) [BHV04], which represents the starting point of the research presented in the following chapters of this thesis. It uses the well known *counter-example-guided-abstract-refinement* (CEGAR) [CGJ$^+$00] paradigm within regular model checking. Abstractions are defined on word automata representing configurations. Then, an abstract reachability analysis which is guaranteed to terminate is performed. Suitable refinements of abstractions are defined for the case a spurious counter-example is encountered. In this way, an abstraction detailed just enough to answer a particular verification question is computed. ARMC has been successfully applied to a lot of different systems, like counter automata, parameterized networks of processes, or programs manipulating 1-selector-linked lists [BHMV05].

**Plan of the chapter.** In Section 2.1, we introduce some basic notions on finite automata and finite transducers used in the following sections. In Section 2.2, we present the basics

[1]Sometimes, some other representations are used too, like logic or specialized algoritmic representations.

of the regular model checking technique including a simple example and a short description of the existing acceleration techniques. In Section 2.3, we describe in detail abstract regular model checking (ARMC) – an acceleration technique based on abstraction on automata. Four different types of abstraction methods are presented in this section. Finally we summarize this chapter in Section 2.4.

## 2.1 Finite Automata and Transducers

Let us recall some basic notions of finite automata, and transducers.

### Finite automata

A *finite automaton* is a tuple $M = (\Sigma, Q, q_0, F, \delta)$, where $\Sigma$ is a finite alphabet, $Q$ is a finite set of states, $q_0 \in Q$ the initial state, $F \subseteq Q$ the set of final states and $\delta \subseteq Q \times \Sigma \times Q$ a finite set of rules. The transition relation $\rightarrow \subseteq Q \times \Sigma^* \times Q$ is defined as follows:

1. $\forall q \in Q : q \xrightarrow{\epsilon} q$
2. $\forall (q_1, a, q_2) \in \delta : q_1 \xrightarrow{a} q_2$
3. $q_1 \xrightarrow{a} q_2 \ \wedge \ q_2 \xrightarrow{b} q_3 \Rightarrow q_1 \xrightarrow{ab} q_3$, where $a, b \in \Sigma^*$.

A language recognized by the automaton $M = (\Sigma, Q, q_0, F, \delta)$ is a set of words $L(M) = \{w \mid w \in \Sigma^* \ \wedge \ q_0 \xrightarrow{w} q_f \ \wedge \ q_f \in F\}$. A language of a state $q_1 \in Q$ is a set of words $L(M, q_1) = \{w \mid w \in \Sigma^* \ \wedge \ q_1 \xrightarrow{w} q_f \ \wedge \ q_f \in F\}$. A set $A \subseteq \Sigma^*$ is regular if and only if there exists a finite automaton $M = (\Sigma, Q, q_0, F, \delta) : L(M) = A$. The *backward* language of a state $q$ is defined as $\overleftarrow{L}(M, q) = \{w \mid q_0 \xrightarrow{w} q\}$.. The language of words up to the bound $n \in \mathbb{N}$ as $L(M, q)^{\leq n} = \{w \in L(M, q) \mid |w| \leq n\}$ and the backward language up to the bound $n \in \mathbb{N}$ as $\overleftarrow{L}(M, q)^{\leq n} = \{w \in \overleftarrow{L}(M, q) \mid |w| \leq n\}$. We denote the set of all finite automata over the alphabet $\Sigma$ as $\mathbb{M}_\Sigma$.

### Finite transducers

A *finite transducer* is a tuple $\tau = (\Sigma_1, \Sigma_2, Q, q_0, F, \delta)$, where $\Sigma_1$ is a finite input alphabet, $\Sigma_2$ is a finite output alphabet, $Q$ is a finite set of states, $q_0 \in Q$ the initial state, $F \subseteq Q$ the set of final states and $\delta \subseteq Q \times (\Sigma_1 \cup \{\epsilon\}) \times (\Sigma_2 \cup \{\epsilon\}) \times Q$ a finite set of transition rules. The transition relation $\rightarrow \subseteq Q \times \Sigma_1^* \times \Sigma_2^* \times Q$ is defined as follows:

1. $\forall q \in Q : q \xrightarrow{\epsilon/\epsilon} q$
2. $\forall (q_1, a, b, q_2) \in \delta : q_1 \xrightarrow{a/b} q_2$
3. $q_1 \xrightarrow{a/b} q_2 \ \wedge \ q_2 \xrightarrow{c/d} q_3 \Rightarrow q_1 \xrightarrow{ac/bd} q_3$, where $a, c \in \Sigma_1^*$ and $b, d \in \Sigma_2^*$ .

We say that a transducer $\tau = (\Sigma_1, \Sigma_2, Q, q_0, F, \delta)$ is *structure-preserving* if $\delta \subseteq Q \times \Sigma_1 \times \Sigma_2 \times Q$.

For a transducer $\tau = (\Sigma_1, \Sigma_2, Q, q_0, F, \delta)$ and a word $w \in \Sigma_1^*$, we define an image $\tau(w) = \{w' \in \Sigma_2^* \mid q_0 \overset{w/w'}{\rightarrow} q_f \ \wedge \ q_f \in F\}$, and for a word $w_1 \in \Sigma_2$, the inverse image is defined as $\tau^{-1}(w_1) = \{w' \in \Sigma_1^* \mid q_0 \overset{w'/w_1}{\rightarrow} q_f \ \wedge \ q_f \in F\}$. Then for a set of words $A \subseteq \Sigma_1^*$, the image of $A$ with respect to the transducer $\tau$ is defined as the set $\tau(A) = \{w' \in \Sigma_2^* \mid \exists w \in A \ \wedge \ q_0 \overset{w/w'}{\rightarrow} q_f \ \wedge \ q_f \in F\}$.

For $\Sigma_1 = \Sigma_2$, we define $\tau^n(A) = \underbrace{\tau(\tau(...(\tau(A))))}_{n}$ and the reflexive-transitive closure $\tau^*(A) = \bigcup_{n \in \mathbb{N}} \tau^n(A)$. Note, that for a regular set $A$, a transducer $\tau$, and $n \in \mathbb{N}$, $\tau(A)$, $\tau^{-1}(A)$, and $\tau^n(A)$ are regular and effectively computable sets. $\tau^*(A)$ is in general not regular, and even if it is regular, it may not be computable.

## 2.2 Regular Model Checking

Regular model checking [BJNT00] is a highly promising method of symbolic verification. As we have already mentioned, in regular model checking, a system configuration is represented as a word over a finite alphabet. An infinite set of configurations is then represented as a regular language and can be naturally described by a finite automaton. A system behaviour is described usually by a finite transducer (where $\Sigma_1 = \Sigma_2$). Sets of initial and bad configurations are represented as finite automata *Init* and *Bad*.

Verification is then based on computing the set of all reachable states from initial configurations by repeatedly applying the transducer on the set of initial states (or on repeatedly composing the transducers with the aim of computing the reachability relation). The computation is stopped when a fixpoint is reached. The verification problem consists in deciding whether

$$\tau^*(L(\mathit{Init})) \cap L(\mathit{Bad}) = \emptyset$$

As the problem being solved in regular model checking is in general undecidable, the method *does not necessarily terminate*. To facilitate termination, various *acceleration methods* have been proposed—see Sections 2.2.2 and 2.3.

### 2.2.1 Example: Token Passing

To illustrate the basic idea of regular model checking, we use an example of a simple *token passing* protocol. The example works on a parameterized network of (linearly connected) processes where each process has two states: (i) the process has a token and (ii) the process has no token. At the beginning, only the first process has a token. In one computation step, the token can be handed to the successor. There is an unbounded number of such networks (the number of processes is not restricted). One can easily model the state of this network as a word over a finite alphabet $\Sigma = \{n, t\}$. Each symbol in the word represents a state of a single process as follows: $t$ – process has a token, $n$ – process has

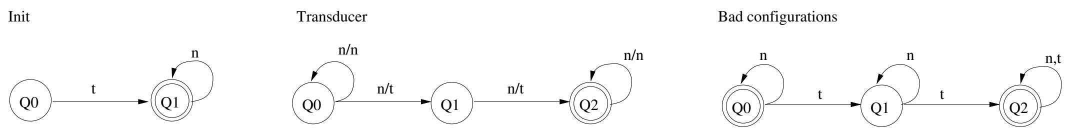


Figure 2.1: Token passing: the initial set, the transducer and the bad set

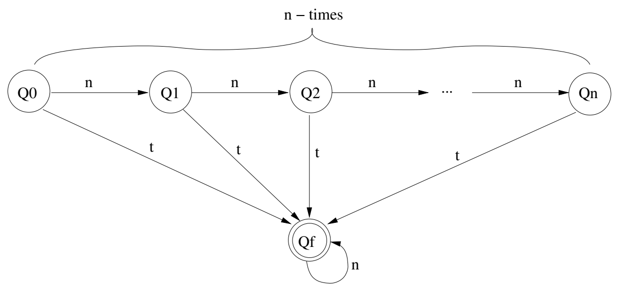


Figure 2.2: Token passing: the reachable set after n iterations

no token. A set of all possible words describing initial configurations is the set described by the regular expression $tn^*$. The transition relation allows a token to move to the right successor and can be encoded as a finite transducer. Bad configurations are words with (i) no symbol $t$ (the token was lost) or (ii) more then one symbol $t$ (the token was replicated). The automata describing the initial and the bad set of configurations as well as the finite transducer describing the behaviour of the system are depicted in Figure 2.1 (*Q0* is the initial state of all automata).

If we start applying the transducer (with the identity relation included) on the initial set, we receive a sequence of growing automata. Each of these automata describes a set of reachable configurations after $n$ applications of the transducer, where the symbol $t$ is placed on at most n-th position in the word. These sets are described by the automaton in Figure 2.2. But the fixpoint represented by the automaton in Figure 2.3 is never reached – we need some acceleration method even in this simple example.

### 2.2.2 Acceleration techniques

The acceleration methods designed for regular model checking include, extrapolation [BJNT00], quotienting – collapsing of automata states based on the history of their creation by composing transducers [AdJN03], acceleration schemas [PS00], inference of regular languages [FO97, HV04], or abstraction of automata [BHV04]. Apart from the

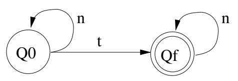


Figure 2.3: Token passing: the fixpoint

last one, these techniques are shortly described in this section. The last one, abstraction of automata, is described separately in Section 2.3.

### Extrapolation

The extrapolation (or widening) principle was originally proposed in [BJNT00]. It is based on searching for repeating patterns in the sequence $I, \varrho(I), \varrho(\varrho(I)), \ldots$, where $I = L(Init)$ and $\varrho$ is a transition relation. The technique works as follows: let $L \subseteq \Sigma^*$ be a partially computed regular set of reachable configurations and $\varrho \subseteq \Sigma^* \times \Sigma^*$ a transition relation. One checks, whether there are regular sets $L_1, L_2, \Delta$ for which the following two conditions holds:

- $L = L_1.L_2$ and $\varrho(L) = L_1.\Delta.L_2$.
- $L_1.\Delta^*.L_2 = \varrho(L_1.\Delta^*.L_2) \cup L$.

If these conditions hold, then $L_1.\Delta^*.L_2$ is added to the set of reachable configurations. The technique was extended in other works [Tou01, BLW03].

### Quotienting

The idea of this technique is based on working with the so-called history transducers whose states reflect the history of their creation. Let $\tau = (\Sigma, \Sigma, Q, q_0, F, \delta)$ be a length preserving transducer describing a single step of the examined system. We then define the so-called history transducer as an infinite-state transducer $\tau_h = (\Sigma, \Sigma, Q^+, \{q_0\}^+, F^+, \delta_h)$ as follows: $q_1 q_2 \ldots q_n \overset{a/a'}{\rightarrow} q'_1 q'_2 \ldots q'_n \in \delta_h$ for some $n \in \mathbb{N}$ iff there exists $a_1, a_2, \ldots, a_{n+1} \in \Sigma : a_1 = a, a_{n+1} = a'$ and $\forall i \in \{1, \ldots, n\} : q_i \overset{a_i/a_{i+1}}{\rightarrow} q'_i$. Intuitively the transition $q_1 q_2 \ldots q_n \overset{a/a'}{\rightarrow} q'_1 q'_2 \ldots q'_n$ corresponds to a composition of n transitions of the original transducer $\tau$ which are necessary to rewrite $a$ to $a'$. The transducer $\tau_h$ describes the transitive closure of the relation described by the transducer $\tau$. But in practise, it is useless because it is infinite-state. The idea is to find an equivalence relation $\sim$ on the states of $\tau_h$ such that the quotient transducer $\tau_{h/\sim}$ is finite-state and describes exactly the reachability relation.

A systematic framework for creating this equivalence relations usable for collapsing states of history transducers and a lot of improvements were proposed in series of works [BJNT00, JN00, Nil00, DLS01, AdJN02, AdJN03, Nil05]. The implementation of the Uppsala regular model checking tool [URM] is based on these techniques.

### Acceleration Schemas

The acceleration schemas are proposed in [PS00] for a verification of parameterized networks of processes by means of RMC. From an original transition system, a new meta-transition system is derived. This new transition system allows one to make an arbitrary number of original transitions in one step. There were defined three concrete acceleration schemas in [PS00]: (1) *local acceleration* allows one to fire an arbitrary number of transitions on a single process in one step, (2) *global acceleration of unary transitions* allows an arbitrary number of processes to fire a given transition in one step, and (3) *global acceleration of binary transitions* allows any number of processes to fire two transitions each to communicate with both its neighbours in one step. The method was implemented in the TLV[P] tool [Sha01].

**Inference of Regular Languages**

This method firstly appeared in [FO97] and then it was then extended in [HV04, HV05, VSVA04b, VSVA04a]. First, for an infinite-state system whose behaviour is described by a length-preserving transducer $\tau$, a set containing all reachable words (from the initial set) up to the given length is computed. This set is finite and therefore it can be computed by a simple iterative application of the transducer $\tau$ ($\tau$ includes identity) on the set of initial configurations. These configurations are taken as a sample of all possible reachable configurations.

Then some language inference algorithm may be applied to learn the whole reachable set (or its overapproximation) from this sample. In [HV04, HV05], there was used the Trakhtenbrot-Barzdin algorithm [TB73]. The works [VSVA04b, VSVA04a] use RPNI [OG92, Lan92] or Angluin's $L^*$ [Ang87] algorithms. As was shown in [HV04, HV05], termination of the method is guaranteed in all cases where the set of all reachable configurations is regular. (This is not the case of other acceleration methods in RMC). In [VSVA04b, VSVA04a], similar results were proved.

## 2.3 Abstract Regular Model Checking

Abstract regular model checking [BHV04] combines regular model checking with automatic abstractions. The computation is based on the counter-example guided abstraction refinement (CEGAR) principle. One defined an automata abstraction function $\alpha$ which is a mapping $\alpha : \mathbb{M}_\Sigma \to \mathbb{A}_\Sigma$ such that $\mathbb{A}_\Sigma \subseteq \mathbb{M}_\Sigma$ and $\forall M \in \mathbb{M}_\Sigma : L(M) \subseteq L(\alpha(M))$. An abstraction $\alpha'$ is called a refinement of the abstraction $\alpha$ if $\forall M \in \mathbb{M}_\Sigma : L(M) \subseteq L(\alpha'(M)) \subseteq L(\alpha(M))$. Given a transducer $\tau$ and an abstraction $\alpha$, $\tau_\alpha$ is a mapping such that $\forall M \in \mathbb{M}_\Sigma : \tau_\alpha(M) = \alpha(\hat{\tau}(M))$, where $\hat{\tau}(M)$ is the minimal deterministic automaton describing language $\tau(L(M))$. The abstraction $\alpha$ is finitary if $\mathbb{A}_\Sigma$ is finite range.

Before the computation, it is necessary to define the abstraction function $\alpha$, the set of initial configurations represented by a finite automaton *Init* and the set of bad configurations represented as an automaton *Bad*. The second step is to compute the transitive closure $\tau_\alpha^*(L)$. This transitive closure contains all possible configurations reachable from the set "Init" by the transducer $\tau_\alpha$.

If the abstraction $\alpha$ is finitary, then there exists $n \in \mathbb{N} : \forall i \geq n . \tau_\alpha^i(Init) = \tau_\alpha^{i+1}(Init)$. Therefore the transitive closure $\tau_\alpha^*(Init)$ can be computed iteratively as

$$\tau_\alpha^*(Init) = Init \cup \tau_\alpha(Init) \cup \tau_\alpha^2(Init) \cup \ldots .$$

The set $L(\tau_\alpha^*(Init))$ is an overapproximation of $\tau^*(L(Init))$. Therefore, $L(\tau_\alpha^*(Init))$ contains all possible configurations reachable from the initial configurations included in the set *Init* by means of the transducer $\tau$.

If $L(\tau_\alpha^*(Init)) \cap L(Bad) = \emptyset$, then the verification problem has a positive answer. In case of a nonempty intersection, it is necessary to check if the obtained counter-example is a real one, or just an error state caused by the used abstraction. The check of the spuriousness is showed in Figure 2.4. Suppose that we reach an intersection with the set of bad states after $l$-th application of the abstracted transition relation $\tau_\alpha$. Then, we compute the sequence of sets $L(X_i)$ where $L(X_l) = L(M_l) \cap L(Bad)$, and $L(X_i) = \tau^{-1}(L(X_{i+1})) \cap L(M_i)$. If $L(X_0) \cap L(Init) \neq \emptyset$, the counter example is a real one. Otherwise it is a spurious one and the abstraction must be refined.

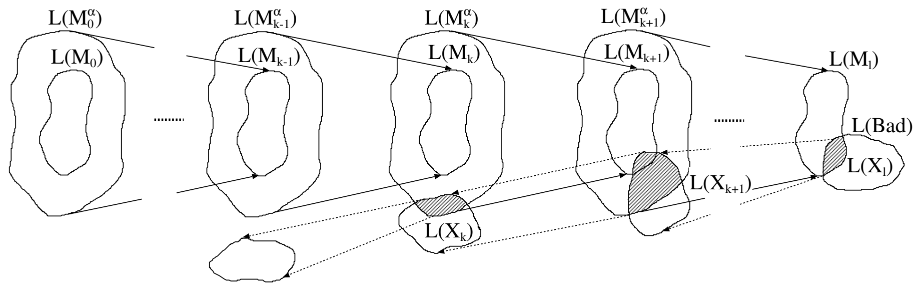


Figure 2.4: A run of abstract regular model checking leading to a spurious counterexample

## 2.3.1 Abstraction Based on Automata State Equivalence

In [BHV04], there were proposed several types of abstractions and their possible refinement. These abstractions methods are based on collapsing states of an automaton. First, the automata states are divided into (a finite number of) classes according to a given equivalence relation. Then, all states from one class are replaced by one new state.

Formally, an automata state equivalence schema $\mathbb{E}$ is defined as follows: To each finite automaton $M = (Q, \Sigma, \delta, q_0, F) \in \mathbb{M}_\Sigma$, an equivalence relation $\sim_M^\mathbb{E} \subseteq Q \times Q$ is assigned. Then the automata abstraction function $\alpha_\mathbb{E}$ corresponding to the abstraction schema $\mathbb{E}$ is defined as $\forall M \in \mathbb{M}_\Sigma : \alpha_\mathbb{E}(M) = M/\sim_M^\mathbb{E}$. Clearly $L(M) \subseteq L(\alpha_\mathbb{E}(M))$. We call $\mathbb{E}$ finitary if the abstraction function $\alpha_\mathbb{E}$ is finitary (i.e. there is a finite number of equivalence classes). We refine $\mathbb{E}$ by making $\sim_M^\mathbb{E}$ finer.

### Abstraction Based on Predicate Languages

In [BHV04], there are first considered two similar abstraction schemas based on comparing languages of states with a selected set of predicate languages $\mathcal{P} = \{P_1, \dots, P_n\}$.

One of this schemas – $\mathbb{F}_\mathcal{P}$ – uses forward languages. Two states of an automaton are equivalent if its forward languages have a non-empty intersection with an equal set of predicate languages. Formally, for a finite automaton $M = (\Sigma, Q, q_0, F, \delta)$, the equivalence relation is defined as follows: $\forall q_1, q_2 \in Q : q_1 \sim_M^{\mathcal{P}} q_2 \Leftrightarrow (\forall P \in \mathcal{P} : L(M, q_1) \cap P \neq \emptyset \Leftrightarrow L(M, q_2) \cap P \neq \emptyset)$. There is a finite number of predicates in the set $\mathcal{P}$, therefore there is $2^{|\mathcal{P}|}$ equivalence classes. Thus the abstraction schema $\mathbb{F}_{\mathcal{P}}$ defines a finitary abstraction on automata, and termination of the abstract fixpoint computation based on $\mathbb{F}_{\mathcal{P}}$ is guaranteed. In case of a spurious counterexample, we can easily refine the abstraction schema by adding new predicates into the set $\mathcal{P}$. The new predicates are languages of all states of the automaton $X_{k+1}$ (recall $X_k = \emptyset$) from the analysis of the spurious counterexample. This refinement guarantees an exclusion of the encountered spurious counterexample [BHV04].

The abstraction schema $\mathbb{B}_{\mathcal{P}}$ is similar to $\mathbb{F}_{\mathcal{P}}$, but it is defined on backward languages. For a finite automaton $M = (\Sigma, Q, q_0, F, \delta)$, the equivalence relation is defined as follows: $\forall q_1, q_2 \in Q : q_1 \overset{\leftarrow\mathcal{P}}{\sim}_M q_2 \Leftrightarrow (\forall P \in \mathcal{P} : \overleftarrow{L}(M, q_1) \cap P \neq \emptyset \Leftrightarrow \overleftarrow{L}(M, q_2) \cap P \neq \emptyset)$. The refinement is done in a similar fashion as in the case of $\mathbb{F}_{\mathcal{P}}$, and an exclusion of the spurious counterexample is guaranteed too.

### Abstraction Based on Finite Length Languages

Other possible abstraction techniques suggested in [BHV04] are based on languages of a finite length. If two states have exactly the same languages of words with length at most n, then they are claimed as equal. This abstraction schema is defined on forward languages $\mathbb{F}_n^L$ and backward languages $\mathbb{B}_n^L$. Formally, the abstraction schema $\mathbb{F}_n^L$ is defined for an automaton $M = (\Sigma, Q, q_0, F, \delta) \in \mathbb{M}_\Sigma$ as follows: $\forall q_1, q_2 \in Q : q_1 \sim_M^n q_2 \Leftrightarrow L(M, q_1)^{\leq n} = L(M, q_2)^{\leq n}$. The abstraction schema $\mathbb{B}_n^L$ is defined similarly: $\forall q_1, q_2 \in Q : q_1 \sim_M^n q_2 \Leftrightarrow \overleftarrow{L}(M, q_1)^{\leq n} = \overleftarrow{L}(M, q_2)^{\leq n}$. Both of this abstraction schemas – $\mathbb{F}_n^L$ and $\mathbb{B}_n^L$ are finitary (the number of languages up to the given bound is finite), and therefore the computation of $\tau_\alpha^*(Init)$ terminates. A refinement is done easily by an increase of the length $n$.

## 2.3.2 Configuration-Oriented Abstractions

In [BHMV05], there were proposed two another abstraction methods. Unlike the methods above that are defined on representations of sets (i.e. finite automata), the abstraction in [BHMV05] are defined on configurations – i.e. on words. There were defined two abstraction methods (i) the *piecewise 0-k counter abstraction* and (ii) the *closure abstraction*. Both of these abstraction methods (and also the previous ones) were successfully used in a prototype implementation of a tool for verifying programs manipulating 1-selector linked data structures [BHMV05].

### Piecewise 0-k Counter Abstraction

The idea behind this abstraction is to partition each word (configuration of a system) into a finite number of parts and forget the information about the ordering inside these parts. Formally, for $w \in \Sigma^*$, let $dec(w) = (a_1, w_1, a_2, w_2, \ldots, a_n, w_n)$ such that $w = a_1 w_1 a_2 w_2 \ldots a_n w_n$, $\forall i, j \in \{1, \ldots, n\}, a_i \in \Sigma \wedge a_i \neq a_j$ (for $i \neq j$) and $\forall i \in \{1, \ldots, n\}$,

$w_i \in \{a_1, \ldots, a_i\}^*$. Intuitively, the decomposition is done by the first occurrence of each symbol.

For a word $w$ and a symbol $a$, let $|w|_a$ be the number of occurrences of the symbol $a$ in $w$. For $k \in \mathbb{N}$, mapping $\alpha_k$ from words to languages is defined as follows: Let $dec(w) = (a_1, w_1, a_2, w_2, \ldots, a_n, w_n)$, then $\alpha_k(w) = a_1 L_1 a_2 L_2 \ldots a_n L_n$, where $\forall i \in \{1, \ldots, n\} : L_i = \{u \in \{a_1, \ldots, a_i\}^* \mid \forall j \in \{1, \ldots, i\}\ |w_i|_{a_j} < k \wedge |u|_{a_j} = |w_i|_{a_j}\ or\ |w_i|_{a_j} \geq k \wedge |u|_{a_j} \geq k\}$. The abstraction $\alpha_k$ can be easily generalized from words to languages. As was discussed in [BHMV05], $\alpha_k$ is regular and effectively representable by a finite transducer for all $k > 0$. For a given alphabet $\Sigma$ and $k \in \mathbb{N}$, the set of possible 0-k abstractions is finite, and therefore this abstraction schema is finitary. This abstraction can be refined by an increase of the parameter $k$.

#### Closure Abstraction

The main idea behind the closure abstraction [BHMV05] is an application of so-called extrapolating rules. This rules may be seen as rewriting rules which rewrite a word $u^k$ to a set of words $u^k u^*$. A relation $R \subseteq \Sigma^* \times \Sigma^*$ is an *extrapolating rule* wrt. a pair $(u, k)$ for $u \in \Sigma^*$ and $k \in \mathbb{N}$ if $R = \{(w, w') | w = u_1 u^k u_2 \ \wedge \ w' = u_1 u^k u^* u_2\}$. An *extrapolating system* is then a finite set of extrapolating rules. The closure abstraction is defined as a reflexive-transitive closure $R_s^*$ of the extrapolating system $R_s$. Clearly, for a regular language $L$ and an extrapolating rule $R$, $L \subseteq R(L)$.

It was proved in [BHMV05] that for every single extrapolating rule $R$ and a regular language $L$, the set $R^*(L)$ is regular and effective computable. But for an extrapolating system $R_s$, the set $R_s^*(L)$ is in general non-regular. In [BHMV05], restricting conditions on an extrapolating system $R_s$ are defined to ensure that $R_s(L)$ is a regular and effectively computable set.

Despite, the closure abstraction is in general not finitary and a termination of the fixpoint computation is not guaranteed, the experimental results on manipulation 1-selector linked structures showed its usefulness [BHMV05].

### 2.3.3 Use of ARMC in Verification

In general, abstract regular model checking does not guarantee termination of the refinement process. It is possible to infinitely many times find a spurious counter example, refine the abstraction, and start again. But in practice, it was showed to be useful for various kinds of systems. The work [BHV04] presents a series of experiments with parameterized networks of processes, push-down systems, systems with queues, Petri nets (systems with counters), etc.

In [BHMV05], there was proposed a systematic translation from programs manipulating 1-selector dynamic linked data structures into the framework of regular model checking. ARMC was successfully applied on a series of case-studies with singly-linked lists (and circular singly-linked lists) including, e.g., insertion of a new element, deletion of an element, reversion of a list, or the bubblesort algorithm.

## 2.4 Summary of RMC

In this chapter, we presented *regular model checking* – an automata based technique for verification of infinite-state systems. In RMC, the configuration of a system being examined is represented as a word over a finite alphabet and regular sets of such configurations are naturally encoded as finite automata. The behaviour of the system is represented as a finite-state transducer or, more generally, a regularity preserving relation whose image on a regular set is effectively computable. We described various techniques designed to compute the set of all reachable configurations including extrapolation [BJNT00], quotienting [AdJN03], acceleration schemas [PS00], inference of regular languages [FO97] and abstraction of automata [BHV04].

The basic approach of regular model checking can be generalized in multiple ways including the use of more general classes of automata than finite state [FP01] or automata on more complex structures than words – e.g., on trees [BT02, AJMd02, Sha02], or $\omega$-words [BLW04]. Chapter 3 of this thesis presents a generalization of abstract regular model checking into trees and Chapters 4 and 5 its successful application for verification of systems with dynamic data structures.

# Chapter 3

# Regular Tree Model Checking

In Chapter 2, we presented *Regular model checking* as a general method for formal verification of infinite-state systems, which allows one to handle linear (or easily linearizable) structures. But there is still a lot of systems based on more complicated structures then a linear ones. To handle non-linear structures, regular *tree* model checking [KMM$^+$01, BT02, AJMd02, SP02, ALdA05] has been proposed. Instead of words, configurations are finite trees and instead of word automata, tree automata are used to represent regular sets of configurations. Then, transitions are represented as tree relations, usually represented as tree transducers although sometimes some other representations are used like in clasical regular model checking too. Like in the word case, several acceleration approaches for reachability analysis exist including abstraction schemas [Sha01, SP02], extrapolation [BT02], quotienting [AJMd02, ALdA05] and abstraction on automata which is a part of our original contribution and on which we concentrate in this chapter.

Tree like structures are very common and appear naturally in many modelling and verification contexts. For example, in the case of parameterized tree networks, labelled trees of arbitrary height represent a configuration of a network: each process is a node of the tree,whose label represents its control state. Trees also arise naturally, e.g., as a representation of configurations of multithreaded recursive programs [Esp02, KvS04], as a representation structure of heaps [KS93], or when representing structured data such as XML documents [BKMW01].

As indicated above, this chapter mainly presents the extension of the ARMC framework (cf. Section 2.3) from words to trees, which was originally published in [BHRV05]. We use bottom-up tree automata and transducers. Like in ARMC, we use abstract fixpoint computations in some *finite* domain of automata. The abstract fixpoint computations always terminate and provide overapproximations of the reachability sets. To achieve this, we define techniques that systematically map any tree automaton $M$ to a tree automaton $M'$ from some finite domain such that $M'$ recognizes a superset of the language of $M$. For the case that the computed overapproximation is too coarse and a spurious counter-example is detected, we give effective principles allowing the abstraction to be refined such that the new abstract computation does not encounter the same counter-example.

We, in particular, propose two abstractions for tree automata. Similarly to ARMC, both of them are based on collapsing automata states according to a suitable equivalence relation. The first is based on considering two tree automata states equivalent if their *languages of trees up to a certain fixed height* are equal. The second abstraction is defined by a set of regular *predicate languages* $L_P$. We consider a state $q$ of a tree automaton $M$ to "satisfy" a predicate language $L_P$ if the intersection of $L_P$ with the tree language $L(M, q)$ accepted from the state $q$ is not empty. Then, two states are equivalent if they satisfy the same predicates. These abstractions correspond to abstraction based on languages of finite length and predicate based abstraction from ARMC. We, however, also show that not all abstraction schemas from ARMC can be lifted to ARTMC.

We have implemented the above abstractions in a prototype tool using the Timbuk [Gen05] tree automata library. We have experimented with the tool on various parameterized tree network protocols.

**Plan of the chapter.** In Section 3.1, we define some basics about tree languages, automata, and transducers. In Section 3.2, we describe the principles of regular tree model checking, provide a simple example and give an overview of acceleration methods. In Section 3.3, we describe in details abstract regular tree model checking – the acceleration technique based on automata abstractions. In Section 3.4, we describe our experimental results with the prototype tool. Finally we conclude in Section 3.5.

## 3.1 Regular Tree Languages and Transducers

This section is a brief introduction to regular tree languages and transducers. A more detailed description can be found, e.g., in [CDG$^+$05, Eng75].

### Terms and Trees

An *alphabet* $\Sigma$ is a finite set of symbols. $\Sigma$ is called *ranked* if there exists a *rank* function $\# : \Sigma \to \mathbb{N}$. For each $k \in \mathbb{N}$, $\Sigma_k \subseteq \Sigma$ is the set of all symbols with rank $k$. Symbols of $\Sigma_0$ are called *constants*. Let $\chi$ be a denumerable set of symbols called *variables*. $T_\Sigma[\chi]$ denotes the set of *terms* over $\Sigma$ and $\chi$. The set $T_\Sigma[\emptyset]$ is denoted by $T_\Sigma$, and its elements are called *ground terms*. A term $t$ from $T_\Sigma[\chi]$ is called *linear* if each variable occurs at most once in $t$.

A finite ordered *tree* $t$ over a set of labels $L$ is a mapping $t : \mathcal{P}os(t) \to L$ where $\mathcal{P}os(t) \subseteq \mathbb{N}^*$ is a finite, prefix-closed set of *positions* in the tree. A term $t \in T_\Sigma[\chi]$ can naturally also be viewed as a tree whose leaves are labelled by constants and variables, and each node with $k$ sons is labelled by a symbol from $\Sigma_k$ [CDG$^+$05]. Therefore, below, we sometimes exchange terms and trees. We denote $\mathcal{N}l\mathcal{P}os(t) = \{p \in \mathcal{P}os(t) \mid \exists i \in \mathbb{N} : pi \in \mathcal{P}os(t)\}$ the set of *non-leaf* positions.

### Tree Automata and Languages

A *bottom-up tree automaton* over a ranked alphabet $\Sigma$ is a tuple $A = (Q, \Sigma, F, \delta)$ where $Q$ is a finite set of states, $F \subseteq Q$ is a set of final states, and $\delta$ is a set of transitions of the following types:

1. $f(q_1, \dots, q_n) \to_\delta q$
2. $a \to_\delta q$
3. $q \to_\delta q'$

where $a \in \Sigma_0$, $f \in \Sigma_n$, and $q, q', q_1, \dots, q_n \in Q$.

**Note:** Below, we call a bottom-up tree automaton simply a tree automaton.

Let $t$ be a ground term. A run of a tree automaton $A$ on $t$ is defined as follows. First, leaves are labelled with states. If a leave is a symbol $a \in \Sigma_0$ and there is a rule $a \to_\delta q \in \delta$, the leave is labelled by $q$. An internal node $f \in \Sigma_k$ is labelled by $q$ if there exists a rule $f(q_1, q_2, \dots, q_k) \to_\delta q \in \delta$ and the first son of the node has the state label $q_1$, the second one $q_2$, ..., and the last one $q_k$. Rules of the type $q \to_\delta q'$ are called *$\varepsilon$-steps* and allow us to change a state label from $q$ to $q'$. If the top symbol is labelled with a state from the set of final states $F$, the term $t$ is accepted by the automaton $A$.

A set of ground terms accepted by a tree automaton $A$ is called a *regular tree language* and is denoted by $L(A)$. Let $A = (Q, \Sigma, F, \delta)$ be a tree automaton and $q \in Q$ a state, then we define the *language of the state q*—$L(A, q)$—as the set of ground terms accepted by the tree automaton $A_q = (Q, \Sigma, \{q\}, \delta)$. The language $L^{\leq n}(A, q)$ is defined to be the set $\{t \in L(A, q) \mid height(t) \leq n\}$.

### Tree Transducers

A *bottom-up tree transducer* is a tuple $\tau = (Q, \Sigma, \Sigma', F, \delta)$ where $Q$ is a finite set of states, $F \subseteq Q$ is a set of final states, $\Sigma$ is an input ranked alphabet, $\Sigma'$ is an output ranked alphabet, and $\delta$ is a set of transition rules of the following types:

1. $f(q_1(x_1), \dots, q_n(x_n)) \to_\delta q(u)$, $u \in T_{\Sigma'}[\{x_1, \dots, x_n\}]$
2. $q(x) \to_\delta q'(u)$, $u \in T_{\Sigma'}[\{x\}]$
3. $a \to_\delta q(u)$, $u \in T_{\Sigma'}$

where $a \in \Sigma_0$, $f \in \Sigma_n$, $x, x_1, \dots, x_n \in \chi$, and $q, q', q_1, \dots, q_n \in Q$.

**Note:** In the following, we call a bottom-up tree transducer simply a tree transducer. We always use tree transducers with $\Sigma = \Sigma'$.

A run of a tree transducer $\tau$ on a ground term $t$ is similar to a run of a tree automaton on this term. First, rules of type *3* are used. If a leaf is labelled by a symbol $a$ and there is a rule $a \to_\delta q(u) \in \delta$, the leaf is replaced by the term $u$ and labelled by the state $q$. If a node is labelled by a symbol $f$, there is a rule $f(q_1(x_1), q_2(x_2), \dots, q_n(x_n)) \to_\delta q(u) \in \delta$, the first subtree of the node has the state label $q_1$, the second one $q_2$, . . ., and the last one $q_n$, then the symbol $f$ and all subtrees of the given node are replaced according to the right-hand side of the rule with the variables $x_1, \dots, x_n$ substituted by the corresponding left-hand-side subtrees. The state label $q$ is assigned to the new tree. Rules of type *2* are called $\varepsilon$-*steps*. They allow us to replace a $q$-state-labelled tree by the right hand side of the rule and assign the state label $q'$ to this new tree with the variable $x$ in the rule substituted by the original tree. A run of a transducer is successful if the root of a tree is processed and is labelled by a state from $F$.

A tree transducer is *linear* if all right-hand sides of its rules are linear (no variable occurs more than once). The class of linear bottom-up tree transducers is closed under composition. A tree transducer is called *structure-preserving* (or a *relabelling*) if it does not modify the structure of input trees and just changes the labels of their nodes. By abuse of notation, we identify a transducer $\tau$ with the relation $\{(t, t') \in T_\Sigma \times T_\Sigma \mid t \rightarrow^*_\delta q(t')$ for some $q \in F\}$. For a set $L \subseteq T_\Sigma$ and a relation $R \subseteq T_\Sigma \times T_\Sigma$, we denote $R(L)$ the set $\{w \in T_\Sigma \mid \exists w' \in L : (w', w) \in R\}$ and $R^{-1}(L)$ the set $\{w \in T_\Sigma \mid \exists w' \in L : (w, w') \in R\}$. If $\tau$ is a linear tree transducer and $L$ is a regular tree language, then the sets $\tau(L)$ and $\tau^{-1}(L)$ are regular and effectively constructible [Eng75, CDG$^+$05].

Let $id \subseteq T_\Sigma \times T_\Sigma$ be the identity relation and $\circ$ the composition of relations. We define recursively the relations $\tau^0 = id$, $\tau^{i+1} = \tau \circ \tau^i$ and $\tau^* = \cup_{i=0}^{\infty} \tau^i$. Below, we suppose $id \subseteq \tau$ meaning that $\tau^i \subseteq \tau^{i+1}$ for all $i \geq 0$.

## 3.2 Basics of Regular Tree Model Checking

We now present in more detail the basic ideas behind regular tree model checking (RTMC) and ilustrate it on the example. In RTMC, a configuration of a system is encoded as a term (tree) over a ranked alphabet and a set of such terms as a regular tree automaton. The transition relation $\varrho$ of a system is usually encoded as a linear tree transducer $\tau$[1]. Sometimes, some other representations of the transition relation $\varrho$ are used too. The only requirement on these representations is the efficient computation of $\varrho(L)$ and $\varrho^{-1}(L)$ for each regular set $L$. We are given a tree automaton $Init$ encoding the set of initial states. For safety properties, a set of bad states (represented by a tree automaton $Bad$) is given. Then, the basic verification problem consists in deciding whether

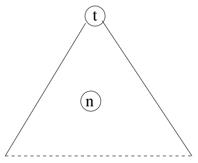
$$\tau^*(L(Init)) \cap L(Bad) = \emptyset \tag{3.1}$$

This problem is in general undecidable (an iterative computation of $\tau^*(L(Init))$ does not terminate). Therefore some acceleration techniques are needed. We give an overview of them in Sections 3.2.2 and 3.3.

### 3.2.1 Example: Tree Token Passing

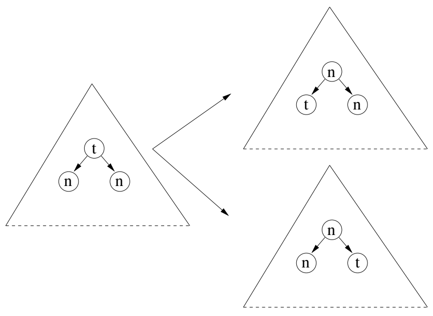
We illustrate the basic principles of regular tree model checking on a simple example of *token passing* similar to the example from Section 2.2.1 but modified to trees. The example works with a parametric network of processes with a tree shape – each process

[1]For non-linear tree transducers $\tau$ and a regular set $L$, the sets $\tau(L)$ and $\tau^{-1}(L)$ are in general not regular.



$$\begin{aligned} &Init = (\{q_n, q_t\}, \Sigma, \{q_t\}, \delta) \\ &\quad \perp \rightarrow_\delta q_n \\ &\quad n(q_n, q_n) \rightarrow_\delta q_n \\ &\quad t(q_n, q_n) \rightarrow_\delta q_t \end{aligned}$$

Figure 3.1: Tree token passing: the initial set of configurations



$$\begin{aligned} &Transd = (\{q_n, q_{set}, q_t\}, \Sigma, \Sigma, \{q_t\}, \delta) \\ &\quad \perp \rightarrow_\delta q_n(\perp) \\ &\quad n(q_n(x_1), q_n(x_2)) \rightarrow_\delta q_n(n(x_1, x_2)) \\ &\quad n(q_n(x_1), q_n(x_2)) \rightarrow_\delta q_{set}(t(x_1, x_2)) \\ &\quad t(q_{set}(x_1), q_n(x_2)) \rightarrow_\delta q_t(n(x_1, x_2)) \\ &\quad t(q_n(x_1), q_{set}(x_2)) \rightarrow_\delta q_t(n(x_1, x_2)) \\ &\quad n(q_t(x_1), q_n(x_2)) \rightarrow_\delta q_t(n(x_1, x_2)) \\ &\quad n(q_n(x_1), q_t(x_2)) \rightarrow_\delta q_t(n(x_1, x_2)) \end{aligned}$$

Figure 3.2: Tree token passing: the transducer

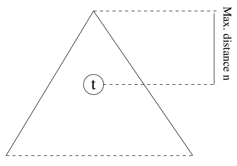
is connected to its parent (except the root process) and to at most two successor processes (we call them the left and the right successor). As in the linear case, each process is in one of the following two states: (i) a process has a token and (ii) a process has no token. At the beginning, only the root process has a token. In each computation step, the token can be handed to the right or to the left successor (if there is one).

A configuration of this parametric network can be encoded as a tree over the ranked alphabet $\Sigma = \{t, n, \perp\}$ with the ranking function $\#(t) = 2, \#(n) = 2$, and $\#(\perp) = 0$. The symbol $n$ or $t$ represents a state of a process and the symbol $\perp$ models non-existence of a (left or right) successor. As the initial set, we consider an automaton accepting all trees with the symbol $t$ in the root and an unbounded number of symbols $n$ in the rest of the tree – see Figure 3.1. The tree transducer describing the behaviour is showed in Figure 3.2.

As in the word case, when we start to apply the transducers (with the identity included) to the initial set, we will never terminate. After $n$ steps, we will obtain an automaton describing the set of trees with the token in the distance of at most $n$ from the root node – Figure 3.3. Therefore, we need to use some acceleration technique to obtain the whole reachable set, where thetoken can be at an arbitrary position in the tree.

### 3.2.2 Acceleration Techniques

In [Sha01, SP02], a tree generalization of *abstraction schemas* was proposed. Similarly to the word case, the authors define (i) a *global unary acceleration* and (ii) a *global binary acceleration*. The first one allows an arbitrary number of processes to execute a given transition in one step. The second one allows an arbitrary number of tree successors to



$$\begin{aligned}
&Reach_n = (\{q_\perp, q_1, \ldots, q_n\}, \Sigma, \{q_1, \ldots, q_n\}, \delta)\\
&\quad \perp \rightarrow_\delta q_\perp\\
&\quad n(q_\perp, q_\perp) \rightarrow_\delta q_\perp\\
&\quad t(q_\perp, q_\perp) \rightarrow_\delta q_n\\
&\quad \forall i \in \{1, \ldots, n\} : \ n(q_i, q_\perp) \rightarrow_\delta q_{i-1}\\
&\quad \forall i \in \{1, \ldots, n\} : \ n(q_\perp, q_i) \rightarrow_\delta q_{i-1}
\end{aligned}$$

Figure 3.3: Tree token passing: the reachable set after $n$ iterations

execute a given synchronous (communication) step as one transition.

The work [BT02] presents a generalization of *extrapolating*. The main advantage of this method is that it works also with non-structure-preserving transducers. As in the word case, it is based on searching for growth patterns and adding an unbounded number of occurrences of these patterns into the reachability set. The disadvantage of this method is that searching for the growth pattern is very expensive. The method was not implemented, so it is hard to evaluate in practice its behaviour.

In [AJMd02, ALdA05] there was proposed a generalization of the *quotienting* approach. As input, it has a structure preserving transducer $\tau$. Similarly to the word case, it is based on creation of a *history transducer* $\tau_h$ and searching for a finite range equivalence relation on states of this infinite state transducer. A systematic method to search for such equivalence relations is described in [ALdA05] and it was implemented as a prototype tool and evaluated on a set of parameterized tree networks of processes similar to examples described in Section 3.4.

All these techniques compute exact sets or reachable configurations (or exact reachability relations). Below, we describe a generalization of abstract regular model checking [BHV04] to tree automata originally proposed in [BHRV05]. This method computes an overapproximation of $\tau^*(L(Init))$ with a precision just sufficient to safely solve the verification problem (3.1).

## 3.3 Abstract Regular Tree Model Checking

This section presents an extension of abstract regular model checking (cf. Section 2.3) into regular tree languages. For this, the abstraction techniques designed for word automata have to be adapted to tree automata.

Let $\Sigma$ be a ranked alphabet and $\mathbb{M}_\Sigma$ the set of all tree automata over $\Sigma$. We define an abstraction function as a mapping $\alpha : \mathbb{M}_\Sigma \to \mathbb{A}_\Sigma$ where $\mathbb{A}_\Sigma \subseteq \mathbb{M}_\Sigma$ and $\forall M \in \mathbb{M}_\Sigma : L(M) \subseteq L(\alpha(M))$. An abstraction $\alpha'$ is called a *refinement* of the abstraction $\alpha$ if $\forall M \in \mathbb{M}_\Sigma : L(M) \subseteq L(\alpha'(M)) \subseteq L(\alpha(M))$. Given a tree transducer $\tau$ and abstraction $\alpha$, we define a mapping $\tau_\alpha : \mathbb{M}_\Sigma \to \mathbb{M}_\Sigma$ as $\forall M \in \mathbb{M}_\Sigma : \tau_\alpha(M) = \hat{\tau}(\alpha(M))$ where $\hat{\tau}(M)$ is a minimal automaton describing the language $\tau(L(M))$. An abstraction $\alpha$ is *finite range* if the set $\mathbb{A}_\Sigma$ is finite.

Let $Init$ be a tree automaton representing the set of initial configurations and $Bad$

be a tree automaton representing the set of bad configurations. Now, we may iteratively compute the sequence $(\tau_\alpha^i(Init))_{i\geq 0}$. Since we suppose $id \subseteq \tau$, it is clear that if $\alpha$ is finitary, there exists $k \geq 0$ such that $\tau_\alpha^{k+1}(Init) = \tau_\alpha^k(Init)$. The definition of $\alpha$ implies $L(\tau_\alpha^k(Init)) \supseteq \tau^*(L(Init))$. This means that in a finite number of steps, we can compute an overapproximation of the reachability set $\tau^*(L(Init))$.

If $L(\tau_\alpha^k(Init)) \cap L(Bad) = \emptyset$, then the verification problem (3.1) has a positive answer. Otherwise, the answer to the problem (3.1) is not necessarily negative since during the computation of $\tau_\alpha^*(L(Init))$, the abstraction $\alpha$ may introduce extra behaviors leading to $L(Bad)$. Let us examine this case. Assume that $\tau_\alpha^*(Init) \cap L(Bad) \neq \emptyset$, which means that there is a symbolic path:

$$Init,\ \tau_\alpha(Init),\ \tau_\alpha^2(Init), \cdots \tau_\alpha^{n-1}(Init),\ \tau_\alpha^n(Init) \tag{3.2}$$

such that $L(\tau_\alpha^n(Init)) \cap L(Bad) \neq \emptyset$. We analyze this path by computing the sets $X_n = L(\tau_\alpha^n(Init)) \cap L(Bad)$, and for every $k \geq 0$, $X_k = L(\tau_\alpha^k(Init)) \cap \tau^{-1}(X_{k+1})$. Two cases may occur:

- $X_0 = L(Init) \cap (\tau^{-1})^n(X_n) \neq \emptyset$, which means that the problem (3.1) has a *negative answer*
- There is a $k \geq 0$ such that $X_k = \emptyset$, and this means that the symbolic path (3.2) is actually a *spurious counter-example* due to the fact that $\alpha$ is too coarse. In this last situation, we need to refine $\alpha$ and iterate the procedure. Therefore, our approach is based on the definition of abstraction schemas allowing to compute families of (automatically) refinable abstractions.

### 3.3.1 Abstraction Based on Automata State Equivalence

Below, we discuss two possible tree automata abstraction schemas which are based on tree automata state equivalence. First, like in ARMC tree automata states are split into several equivalence classes by an equivalence relation. Then, the abstraction function collapses states from each equivalence class into one state. Formally, a tree automata state equivalence schema $\mathbb{E}$ is defined as follows: To each tree automaton $M = (Q, \Sigma, F, \delta) \in \mathbb{M}_\Sigma$, an equivalence relation $\sim_M^{\mathbb{E}} \subseteq Q \times Q$ is assigned. Then the automata abstraction function $\alpha_{\mathbb{E}}$ corresponding to the abstraction schema $\mathbb{E}$ is defined as $\forall M \in \mathbb{M}_\Sigma : \alpha_{\mathbb{E}}(M) = M/\sim_M^{\mathbb{E}}$. We call $\mathbb{E}$ finitary if $\alpha_{\mathbb{E}}$ is finitary (i.e. there is a finite number of equivalence classes). We refine $\mathbb{E}$ by making $\sim_M^{\mathbb{E}}$ finer.

### 3.3.2 Abstraction Based on Languages of Finite Height

We first present the possibility of defining automata state equivalence schemas based on comparing automata states wrt. a certain bounded part of their languages. The abstraction schema $\mathbb{H}_n$ is a generalization of a similar schema proposed for word automata in [BHV04]. This schema defines two states of a tree automaton $M$ as equivalent if their languages up to the given height $n$ are identical.

Formally, for a tree automaton $M = (Q, \Sigma, F, \delta)$, $\mathbb{H}_n$ defines the state equivalence as the equivalence $\sim^n_M$ such that $\forall q_1, q_2 \in Q : q_1 \sim^n_M q_2 \Leftrightarrow L^{\leq n}(M, q_1) = L^{\leq n}(M, q_2)$.

Formally $q_1 \sim^n q_2 \Leftrightarrow L^{\leq n}(A, q_1) = L^{\leq n}(A, q_2)$.

There is a finite number of languages of trees with a maximal height $n$, and so this abstraction is finite range. Refining of the abstraction can be done by increasing the value of $n$.

The abstraction schema $\mathbb{H}_n$ can be implemented in a similar way as minimization of tree automata. Just the main loop of the minimization procedure is stopped after $n$ iterations.

### 3.3.3 Abstraction Based on Predicate Languages

We next introduce a predicate-based abstraction schema $\mathbb{P}_\mathcal{P}$, which was inspired by the predicate based abstraction on words [BHV04].

Let $\mathcal{P} = \{P_1, P_2, \ldots, P_n\}$ be a set of *predicates*. Each predicate $P \in \mathcal{P}$ is a tree language represented by a tree automaton. Let $M = (Q, \Sigma, F, \delta)$ be a tree automaton, then two states $q_1, q_2 \in Q$ are equivalent if their languages $L(M, q_1)$ and $L(M, q_2)$ have a nonempty intersection with exactly the same subset of predicates from the set $\mathcal{P}$.

Formally, for an automaton $M = (Q, \Sigma, F, \delta)$, $\mathbb{P}_\mathcal{P}$ defines the state equivalence as the equivalence $\sim^\mathcal{P}_M$ such that $\forall q_1, q_2 \in Q : q_1 \sim^\mathcal{P}_M q_2 \Leftrightarrow (\forall P \in \mathcal{P} : L(P) \cap L(M, q_1) \neq \emptyset \Leftrightarrow L(P) \cap L(M, q_2) \neq \emptyset)$.

Formally $q_1 \sim^P q_2 \Leftrightarrow (\forall P \in \mathcal{P} : L(P) \cap L(A, q_1) \neq \emptyset \Leftrightarrow L(P) \cap L(A, q_2) \neq \emptyset)$.

Clearly, since $\mathcal{P}$ is finite and there is only a finite number of subsets of $\mathcal{P}$ representing the predicates with which a given state has a nonempty intersection, $\mathbb{P}_\mathcal{P}$ is *finitary*. This schema can be refined by adding new predicates into the set $\mathcal{P}$. The following theorem shows that we may eliminate a spurious counter-example by extending the predicate set $\mathcal{P}$ by the languages of all states of the tree automaton representing $X_{k+1}$ in the analysis of the spurious counter-example (recall that $X_k = \emptyset$) as presented in Section 3.3.

**Theorem 1** *Let us have any two tree automata $M = (Q_M, \Sigma, F_M, \delta_M)$ and $X = (Q_X, \Sigma, F_X, \delta_X)$ and a finite set of predicate automata $\mathcal{P}$ s.t. $\forall q_X \in Q_X : \exists P \in \mathcal{P} : L(X, q_X) = L(P)$. Then, if $L(M) \cap L(X) = \emptyset$, $L(\alpha_{\mathbb{P}_\mathcal{P}}(M)) \cap L(X) = \emptyset$ too.*

*Proof*: The proof is a generalization of the proof [BHV04] for word automata. We prove the theorem by contradiction. Suppose $L(\alpha_{\mathbb{P}_\mathcal{P}}(M)) \cap L(X) \neq \emptyset$. Let $t \in L(\alpha_{\mathbb{P}_\mathcal{P}}(M)) \cap L(X)$. As $t$ is accepted by $\alpha_{\mathbb{P}_\mathcal{P}}(M)$, $M$ must accept it when we allow it to perform a certain number of "jumps" between states equal wrt. $\sim^\mathcal{P}_M$—after accepting a subtree of $t$ and getting to some $q \in Q_M$, $M$ is allowed to jump to any $q' \in Q_M$ such that $q \sim^\mathcal{P}_M q'$ and go on accepting from there (with or without further jumps).

Let $i > 0$ be the minimum number of jumps needed for accepting a tree from $L(\alpha_{\mathbb{P}_\mathcal{P}}(M)) \cap L(X)$ in $M$ and let $t'$ be such a tree. When looking at the acceptance of $t'$ in $M$ (with some jumps allowed), we can identify maximum subtrees of $t'$ that may be accepted without jumps—in the worst case, they are just the leaves. Let us take any of such subtrees. Such a subtree $t_1$ is accepted in some $q_1$, from which $M$ jumps to some $q_2$

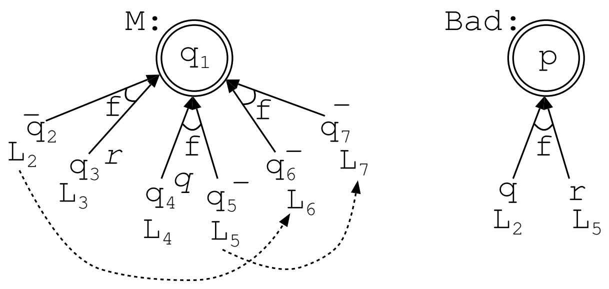


Figure 3.4: A problem with the forward tree predicate abstraction

and goes on accepting the rest of the input. Suppose that $t_1$ is accepted in some $q_X \in Q_X$ in $X$. As $t_1 \in L(M, q_1)$, $L(M, q_1) \cap L(P) \neq \emptyset$ for the predicate $P \in \mathcal{P}$ for which $L(P) = L(X, q_X)$. Moreover, as $q_1 \sim_M^{\mathcal{P}} q_2$, $L(M, q_2) \cap L(P) \neq \emptyset$ too. This implies there exists $t_2 \in L(P)$ such that $t_2 \in L(M, q_2)$ and $t_2 \in L(X, q_X)$. However, this means that the tree $t''$ that we obtain from $t'$ by replacing its subtree $t_1$ with $t_2$ and that clearly belongs to $L(\alpha_{\mathbb{P}_\mathcal{P}}(M)) \cap L(X)$ can be accepted in $M$ with $i - 1$ jumps, which is a contradiction to the assumption of $i$ being the minimum number of jumps needed. □

The abstraction of an automaton $M$ wrt. the state equivalence based on predicate languages $\mathbb{P}_\mathcal{P}$ can be implemented as labelling each state of $M$ by the predicates with which its language has a non-empty intersection, and then collapsing states with an equal labelling. Here, let us stress that when refining $\mathbb{P}_\mathcal{P}$, it is not necessary to store each of the newly introduced predicates corresponding to the states of $X_{k+1}$ independently and then perform the labelling independently for each of them. We may keep just $X_{k+1}$ and then perform labelling not by just $X_{k+1}$ but by each of its states. Moreover, this labelling may be implemented by one simultaneous run through $M$ and $X_{k+1}$, which corresponds to an efficient simultaneous labelling by all the predicates contained in $X_{k+1}$.

Let us add that $\mathbb{P}_\mathcal{P}$ has been in particular inspired by the backward predicate-based abstraction schema $\mathbb{B}_\mathcal{P}$ from [BHV04], which considers words between a given automaton state and the initial state.

It is interesting that as we illustrate in Figure 3.4, it is not possible to obtain a tree analogy with the forward predicate-based abstraction schema $\mathbb{F}_\mathcal{P}$ of the linear ARMC, which considers words between a given automaton state and the final states. The tree analogy would be to label a state with a predicate state if the languages of their contexts—i.e. trees where we substitute $\Sigma^*$ for the language of the node being labelled/used for labelling—have a non-empty intersection. However, in this way, we may not ensure that the abstraction of $M$ from Figure 3.4 will be disjoint with $Bad$ (in Figure 3.4, the upper index of the states of $M$ shows by which states of $Bad$ they are labelled).

## 3.4 Implementation and Experimental Results

In order to be able to practically evaluate the proposed methods of ARTMC, we have implemented them in a prototype tool. We have based our prototype tool on the *Timbuk* library [Gen05] written in OCaml. Timbuk provided us with the basic operations over tree automata needed in ARTMC (such as union, intersection, complementation, etc.). However, we had to extend Timbuk with a support for tree transducers. We added two implementations of tree transducers—a simpler and more efficient for structure-preserving transducers and a more complex for general transducers. The latter implementation exploits a decomposition of a tree transducer into three less complicated ones as described in the following section.

### 3.4.1 Algorithmic Decomposition of Bottom-up Tree Transducers

We distinguish the following classes of tree transducers [Eng75] each of which is relatively easy to implement: *FTA* is the class of all transducers that simply copy their input tree to the output provided it is accepted by the input part of the transducer. *RELAB* is the class of all one-state relabellings. Formally, it contains transducers that may be defined as tuples $(Q, \Sigma_A, \Sigma_B, F, \delta)$ where $Q = F = \{*\}$ and each rule in $\delta$ is of type $a(*(x_1), *(x_2), ..., *(x_n)) \to *(b(x_1, x_2, ..., x_n))$ where $a \in \Sigma_A$ and $b \in \Sigma_B$. *HOM* is the class of all homomorphisms. It contains one-state deterministic transducers. The class *LHOM* is a subclass of *HOM* that contains one-state deterministic linear transducers.

According to [Eng75], we can decompose any bottom-up tree transducer $\tau = (Q, \Sigma_A, \Sigma_B, F, \delta)$ into transducers $\tau_R = (\{*\}, \Sigma_A, \Sigma_X, \{*\}, \delta_R)$, $\tau_A = (Q, \Sigma_X, \Sigma_X, F, \delta_A)$, and $\tau_H = (\{*\}, \Sigma_X, \Sigma_B, \{*\}, \delta_H)$ as follows:

- The rules of $\delta$ are numbered $1..n$ where $n$ is the total number of rules in $\delta$.
- The alphabet $\Sigma_X$ contains $n$ symbols $r_1, r_2, ..., r_n$ where each symbol of this alphabet is associated with one rule of $\delta$.
- For each rule in $\delta$, we create a new rule in $\delta_R$, $\delta_A$, and $\delta_H$ in the following way. Let $a(p_1(x_1), p_2(x_2), ..., p_n(x_n)) \to p_p(b(T[x_1, x_2, ..., x_n]))$ be a rule of $\delta$ numbered by $t$. Then we add the below rules into $\delta_R$, $\delta_A$, and $\delta_H$:
    1. $a(*(x_1), *(x_2), ..., *(x_n)) \to *(r_t(x_1, x_2, ..., x_n))$ into $\delta_R$,
    2. $r_t(p_1(x_1), p_2(x_2), ..., p_n(x_n)) \to p_p(r_t(x_1, x_2, ..., x_n)$ into $\delta_A$, and
    3. $r_t(*(x_1), *(x_2), ..., *(x_n)) \to *(b(T[x_1, x_2, ..., x_n]))$ into $\delta_H$.

The transducer $\tau_R$ is in the class *RELAB*, $\tau_A$ in the class *FTA*, and $\tau_H$ in the class *HOM*. If the original transducer is linear, $\tau_H$ is also linear too. It can be proved [Eng75] that $\tau = \tau_R * \tau_A * \tau_H$. Obviously, the decomposition can be performed automatically for any general bottom-up tree transducer.

### 3.4.2 Verified protocols

We have tested our verification methods on several examples of protocols using a parameterized tree-shaped network cited in the literature [KMM$^+$01, ABH$^+$97, AJMd02, ALdA05] where the necessity to cover all possible values of the parameters leads to dealing with infinite state spaces:

- *Simple Token Protocol.* A token is being passed in a tree-shaped network from a leave to the root. We check that the token does not disappear nor replicate.
- *Two-Way Token Protocol.* An analogy to the previous example, but we allow the token to be passed upwards as well as downwards.
- *Percolate Protocol.* A tree-shaped network of processors computes the logical disjunction of the boolean values that appear in the leave nodes. We check that the computed value is always correct.
- *Tree Arbiter Protocol.* A tree-shaped network is used to implement mutual exclusion among the leave processors. A request to enter the critical section is propagated upwards till a node is found which has a token allowing one to enter the critical section or which knows where the token is (because it granted the token to one of its children). A node with the token can always send the token upwards or grant it to any of its children. We check the mutual exclusion property.
- *Leader Election Protocol.* One of a set of processors is to be elected a leader and a tree-shaped network is used for this purpose. The leaves are divided into candidates and non-candidates. The information about the existence of candidates is propagated upwards. In the subsequent downward phase, a path leading from the root to one of the candidate nodes is non-deterministically selected and thus a leader is established. We check that exactly one leader is chosen.

All the above examples work with a tree-shaped network of a fixed structure. In order to test the ability of our method to work with non-structure-preserving systems, we have considered a *simple broadcast protocol.* In the protocol, the root sends a message to all leave nodes. They answer and the answers are combined when travelling upwards. An intermediate node may decide to resend the message downwards and wait for new data. New nodes may dynamically join the network at leaves and also leave the network in a suitable moment. We check that there is at most one active message on each path from the root to the leaves.

The results of our experiments are summarized in Table 3.1. We performed experiments with both the finite-height abstraction as well as with the predicate-based abstraction. We considered both forward as well as backward verification—i.e. starting with the set of initial states and checking that the bad states cannot be reached or vice versa. In the table, we always present the better result of these two approaches. For the finite-height abstraction, we considered the initial height 1 (and increased it by 1 if necessary—in the cases presented in Table 3.1 this was not necessary). For the predicate-based abstraction, we considered the automaton describing the set of bad states as the only initial predicate (or—more precisely—all the automata that can be obtained from it by considering each of its states as the only accepting one—in cases presented in Table 3.1 was not necessary to refine predicates). We experimented with the empty initial set of predicates too—this turned out to be the fastest option for the Percolate protocol (one refinement is done during the computation). In comparison with the implementation of the quotienting technique published in [ALdA05], we consider our results encouraging. The running times of quotienting technique (on unspecified machine) are almost the same as our results for token passing, two-way token passing and percolate. But the fixpoints for tree arbiter and leader election protocol, it takes several minutes to compute fixpoints by means of quotionting technique.

Table 3.1: Some results of experimenting with ARTMC

| Protocol | $\mathbb{H}_n$ | $\mathbb{P}_{\mathcal{P}}$ |
|---|---|---|
| Token passing | backwards: 0.08s | forwards: 0.06s |
| Two-way token passing | backwards: 1.0s | forwards: 0.09s |
| Percolate | backwards: 20.8s | forwards: 2.4s |
| Tree arbiter | backwards: 0.31s | backwards: 0.34s |
| Leader election | backwards: 2.0s | forwards: 1.74s |
| Broadcasting | backwards: 9.1s | forwards: 1.0s |

The verification times presented in Table 3.1 were obtained on an Intel Centrino 1.6GHz machine with 768MB of memory.

## 3.5 Summary of RTMC

In this chapter, we presented *regular tree model checking*, we gave a brief overview of acceleration methods used within this framework and then we concentrate on *abstract regular tree model checking*, which is one of our original contributions. In particular, we have proposed two kinds of abstractions over tree automata based on collapsing in some sense equivalent states of these automata. One of the abstractions decides which states are equivalent by comparing their languages of trees of a bounded height while the second one compares the states wrt. whether their languages satisfy (i.e. have a non-empty intersection with) a set of predicates having the form of regular tree languages. Both of these abstractions are automatically refinable when a spurious counter-example is found and allow one to deal with an overapproximation of the state space precise just enough to verify a given property of interest. In this way, the state explosion in automata representing the reachability set is fought. The above abstractions were inspired by some of the schemas used in the original ARMC. However, on an example, we have also shown that not all abstraction schemas applicable in the linear ARMC can be lifted to the domain of ARTMC. We have implemented the proposed methods in a prototype tool and evaluated them on multiple verification examples. Our experimental results showed that ARTMC is a promising option in area of regular tree model checking and motivated us to the next research based on this technique.

We have expected that ARTMC can be used in several areas of verification including parametrized networks of processes, manipulations of XML documents, cryptographic protocols, etc. One of the most promising areas is verification of programs with dynamic linked data structures. ARMC has been shown useful for verification of programs with 1-selector linked dynamic data structures [BHMV05]. The use of ARTMC allows us to handle much more general structures as we describe in Chapter 4, which presents our original contribution in this area.

# Chapter 4

# Abstract Regular Tree Model Checking of Complex Dynamic Data Structures

Automated verification of programs manipulating dynamic linked data structures is currently a very live research area. This is partly due to the fact that programs manipulating pointers are often complex and tricky, and so methods for automatically analyzing them are quite welcome, and also because automated verification of such programs is not easy. Programs manipulating dynamic linked data structures are typically infinite-state systems, their configurations have in general the form of unrestricted graphs (often referred to as the *shape graphs*), and the shape invariants of these graphs may be temporarily broken by the programs during destructive pointer updates.

This chapter describes our fully-automated method for analyzing various important properties of programs manipulating dynamic linked data structures. The results are based on works [BHRV06b] and [Rog06]. We consider non-recursive C programs (with variables over finite data domains) manipulating dynamic linked data structures with possibly *several* next pointer selectors. The properties we consider are basic consistency of pointer manipulations (no *null pointer assignments*, no use of *undefined pointers*, no references to *deleted elements*). Further undesirable behaviour of the verified programs (e.g., breaking of certain *shape invariants* such as an introduction of undesirable sharing, cycles, etc.) may be detected via *testers written in C* and attached to the verified procedures. Moreover, for a more declarative way of specifying undesirable behaviour of the considered programs, we introduce a special-purpose logic LBMP (*logic of bad memory patterns*) and we show that its formulae may be automatically translated into C testers. Then, verification of these properties reduces to reachability of a designated error location.

Our verification method is based on using the approach of *abstract regular tree model checking* (ARTMC, cf. Chapter 3). In order to be able to apply ARTMC for verification of programs manipulating dynamic linked data structures whose configurations (shape graphs) need not be tree-like, we propose an *original encoding of shape graphs based on tree automata*. We use trees to encode the *tree skeleton* of a shape graph. The edges of the shape graph that are not directly encoded in the tree skeleton are then represented by *routing expressions* over the tree skeleton—i.e., regular expressions over directions in a tree (as, e.g., left up, right down, etc.) and the kind of nodes that can be visited on the way. Both the tree skeletons and the routing expressions are automatically discovered by our method. The idea of using routing expressions is inspired by PALE [MS01] and graph types [KS93] although there, they have a bit different form (see below) and are defined manually.

Next, we show how all *pointer-manipulating statements* of the C programming language (without pointer arithmetics, recursion, and with finite-domain non-pointer data) may be *automatically translated to tree transducers* over the proposed tree-automata-based representation of sets of shape graphs. We argue that the use of ARTMC over the proposed representation of shape graphs and program statements is sound.

We implemented our method in a prototype tool based on the Mona tree libraries [KM01]. We have tested it on a number of non-trivial procedures manipulating singly-linked lists (SLL), doubly-linked lists (DLL), trees (including the Deutsch-Schorr-Waite tree traversal), lists of lists, and also trees with linked leaves. To the best of our knowledge, verifying some properties on trees with linked leaves have so-far not been considered in any other fully automated tool. The experimental results obtained from our tool are quite encouraging (and, moreover, we believe that there is still a lot of room for further improvements as we have, e.g., not used the mechanism of Mona's guided tree automata, we have used general-purpose, not specialized abstractions as in [BHV04], etc.).

**Plan of the chapter.** This chapter is organized as follows: In Section 4.1, we shortly present existing techniques for verification of programs with dynamic data structures. In Section 4.2, we present concerned class of programs and concerned properties. Then in Section 4.3, we describe the logic LBMP that we proposed for describing bad memory shapes. In Section 4.4, we present our verification framework based on tree automata, transducers and ARTMC. In Section 4.5, we describe basics of our prototype implementation, our case studies and obtained results. In Section 4.6, we describe the implementation based on Mona tree automata library in more details. In Section 4.7, we discuss a way how to implement garbage checking in our framework, and finally, we summarize the results and propose possible extensions in Section 4.8.

## 4.1 Existing Alternative Approaches

There exist many various approaches for verification of programs manipulating dynamic data structures. These approaches differ in principles used, scalability, or level of automation. We can basically divide them to approaches based on (i) logics, (ii) automata and (iii) other formalisms (grammars, patterns, etc.). This section presents a short overview of the existing techniques, but the research in this area is quite live, new approaches are appearing continuously, and so it is hard to provide a complete overview.

### 4.1.1 Techniques Based on Logics

#### PALE

PALE [MS01] is an often cited approach for semi-automated verification of programs manipulating dynamic data structures – the semi-automation means that it is necessary to provide invariants for all loops in the program code. (Loop-free parts are handled automatically.)

PALE is capable of manipulating structures which can be expressed by means of the so-called *graph types* [KS93]. Graph types allow one to describe tree structures with an unbounded number of additional edges which are, however, of a sort of regular nature. There is defined a finite number of types of nodes. Each type is associated with a finite number of next pointers leading to nodes of a specified type. Two types of next pointers are considered: (1) *tree edges* – they are represented directly as edges in a tree and (2) *extra pointers*. The extra pointers are represented by regular expressions over directions in the tree (describing the path to the destination node), types of the nodes on the path, and predicates *leaf* (valid on leaf nodes) and *root* (valid on the root node). The description is restricted to deterministic extra edges only. Graph types are fixed at beforehand and they are not modified during a computation. But there can be defined several graph types – each to describe a memory configuration at a different line of the code being analyzed.

PALE uses a special monadic second order logic of graph types to express initial conditions, exit conditions, loop invariants and assertions. The main idea is to check for each loop-free fragment that the provided exit condition (expressed as formulae) holds for the code fragment applied on its input condition. The pointer manipulating statement are automatically translated to manipulations on formulae in this logic so, one starts with precondition, this is changed by syntactic transformation wrt. the manipulations done on it and then one checks whether the resulting formula implies the post condition.

This check is done by translating the obtained formula to an automata representation by means of Mona tool [KM01].

#### TVLA

TVLA (the *Three Valued Logic Analyzer*) [SRW02] is another often cited approach for verification of heap manipulations. It is based on using the Kleene's 3-valued interpretation [Kle87] of predicate logic with transitive closure on graphs. The predicate logic with transitive closure is used to express the so-called *core* and *instrumentation* predicates. Each predicate can be defined on a single node or on a pair of nodes. Core predicates describe the most basic and generic properties of a data structure (e.g. a node is pointed by a pointer variable, a node is a successor of another node). Instrumentation predicates are defined over the core ones, and describe some other, more complex and specific properties of the data structure (e.g. a node satisfies the doubly-linked property *x.next.back* == *x*, a memory node is shared, etc.). To encode a (possibly infinite) set of memory configurations, the following abstraction is used. A set of memory nodes, for which an equal set of designated predicates is valid, is abstracted as one *summary node*. Some these designated predicates can loose their values due to this abstraction, so therefore the 3rd *may-be* logical value was introduces.

The semantics of program statements is represented by *predicate update formulae*. Predicate update formulae are provided automatically for core predicates. For instrumentation predicates, it must be provided manually, or one can use some heuristics to derive it [RSL03]. To make the abstract computation more precise, the *focus* operation was introduced. It splits a single structure to several more precise ones where some summary nodes can be divided into a set of concrete and summary nodes. The appropriate operation is then performed on the concrete materialized nodes. The *coerce* operation is subsequently used to rule out impossible cases, followed by summarization in order to abstract the resulting structure again.

The method is generic, but the user must come up with the right instrumentation predicates which is not straightforward and easy. In [LRS05], a heuristic method to generate instrumentation predicates based on inductive learning was proposed. This method was successfully tested on some sorting algorithms over binary trees. A more comprehensive evaluation of the possibility of the proposed predicate learning technique is currently not available.

### Separation Logic

*Separation logic* [Rey02] is an extension of Hoare logic designed for reasoning about correctness of programs manipulating dynamic data structures. The main newly added part is the *separating conjunction* "*". A formula $P * Q$ holds if there exist two disjoint parts of the memory, and $P$ holds for the first part and $Q$ for the second one. The separating conjunction allows one to perform a local (concerning only a small part of the heap) and modular (disjoint parts of the heap are manipulated separately) reasoning about the heap structures.

Separation logic provides a set of axioms and inference rules which allow manual or semi-automated proofs of pointer manipulations. Moreover, in the last two years, there appeared first methods using separation logic in fully automatic approaches, and a research in this area is now quite live. The work [DOY06] presents a program analysis for programs manipulating singly-linked lists. In [DBCO06], there was proposed a method for a termination check of programs manipulating singly-linked structures. The work [CDOY06] tackles the problem of pointer arithmetics. The work [BCC$^+$07a] proposes use of the separation logic for an automatic verification of so-called *composite data structures* – i.e. lists of lists, where each list inside this structure can be cyclic, doubly-linked, a list with head/tail pointers, etc. In [NDQC07], there was proposed a semi-automated method for checking shape and size properties of data structures and the method was applied to linked lists and tree examples, where the user provides loop invariants.

### Alias Logic

*Alias logic* [BIL03] presents another possibility how to reason about heap manipulations. The logic is defined on a storeless memory model – a heap is not represented as a graph. Instead, each heap node is described by the regular language of paths going to it through the heap. This language is then represented as a finite automaton. Alias logic then allows one to express regular properties over the heap.

### Predicate Abstraction

Most of the current successful software model checkers are based on technique of *predicate abstraction* [GS97, BR01, HJMS02, HJMS03, CCG$^+$04]. The main idea is to transform an infinite state program into a boolean (finite state) one. The transformation is done by means of a finite set of predicates – abstract states represent validity/invalidity of certain predicates. The predicates can represent conditions like "the integer value of $x$ is greater then the value of $y$", "the integer value of $x$ is greater than zero", etc. However, the traditional works using predicate abstraction do not consider programs manipulating dynamic data structures.

The first attempt to use predicate abstraction for such programs was proposed in [DN03]. This approach uses various *reachability predicates*, and from them, it generates some other ones as sharing and cyclicity. A human interaction is necessary to obtain sufficient predicates to prove specified properties. The method was tested on some classical examples with singly-linked lists.

Another possibility was proposed in [BPZ05]. This work considers verification of programs manipulating singly-linked lists. Authors use two types of predicates – (i) predicates of the form "a pointer variables is null" and (ii) predicates of the form "from a node pointed by a given variable, other node pointed by some other given variables is transitively reachable". The abstract program is derived automatically. The authors also consider verification of termination of the given class of programs by manually provided ranking functions.

Predicate abstraction for (possibly) cyclic singly-linked data structures is discussed in [MYRS05]. The main observation is that there exists a finite number of shared nodes in such data structures. Therefore the used predicates capture the existence of uninterrupted sequences of length one, length two or longer then two.

In [BHT05] in context of verifying programs with SLLs, authors combine predicate abstraction with a shape analysis based on canonical abstraction (used in TVLA). The idea is to use the shape analysis only locally in places where it is necessary for verifying the given program. The proposed CEGAR loop allows one to increase the precision of the shape analysis by adding more predicates tracking where certain variables point and tracking the values shared in the lists. The method was implemented as an integration of the TVLA and BLAST [HJMS03] tools.

Predicate abstraction was also used in the work [BPZ07]. It allows one to verify programs manipulating trees without sharing. The key idea is to reverse all parent-son pointers. Then one receives one-selector linked structure. The system and a verified property are automatically transformed to handle the reversed structures and predicate abstraction is then applied. The technique was tested on some examples of trees with fixed arity as balancing of a binary search tree and inserting into a such tree.

### 4.1.2 Automata-Based Techniques

The techniques based on automata include, of course, our technique based on abstract regular tree model checking presented below. This technique allows us to verify safety properties of programs manipulating data structures, which can be encoded as trees with an infinite number of regular *extra edges*. Moreover, in Chapter 5, we present a technique based on a combination of ARTMC with counter automata. This technique allows us to verify termination of programs manipulating tree structures. However, before we get to these techniques, we first overview other existing automata-based techniques.

#### ARMC of One-Selector-Linked Structures

In [BHMV05], there was proposed a method for a fully-automated verification of programs manipulating 1-selector linked structures based on *abstract regular model checking* (cf. Section 2.3). The technique is capable of handling all possible singly-linked structures with sharing and with data values (from a finite domain) in nodes. A memory configuration is encoded as a word over a finite alphabet and a regular set of such configurations as a finite automaton. The program statements are automatically transformed to finite transducers. Then the ARMC technique is used to generate an overapproximation of the set of all reachable configurations. To increase the success of ARMC, two new types of abstraction were defined in addition to the first abstractions originally proposed in [BHV04] (cf. the configuration oriented abstractions in Section 2.3.2).

The technique was implemented and a series of experiments with singly-linked lists (and circular singly-linked lists) including insertion of a new element, deletion of an element, reversion of a list or bublesort were performed.

#### Automata in May-Alias Analysis

The work [Ven99] (continuation of earlier works [Jon81, Deu94]) considered the special problem of *may-alias analysis* – i.e. discovering of a potential sharing. In particular, this analysis searches for pointers and pointer sequences, which can point to an equal location at different times of a program execution. Memory structures are represented as tuples of finite automata (one automaton for each pointer variable) and an alias relation. To accelerate the fixpoint computation, a special widening designed for these structures was proposed.

#### Top-Down Parity Tree Automata

Another automata based approach was proposed in [DEG06]. It is based on parity tree automata (top-down tree automata with the parity acceptance condition). These automata work on memory shape graphs unfolded into infinite trees in a natural way. Each procedure of a program being verified must have a special pointer called *cursor* and all modifications of the heap are done in a finite neighborhood of it. Procedures are limited to a finite number of destructive passes through the memory. Such procedures can be automatically translated into the considered automata. Then, a product of the automaton

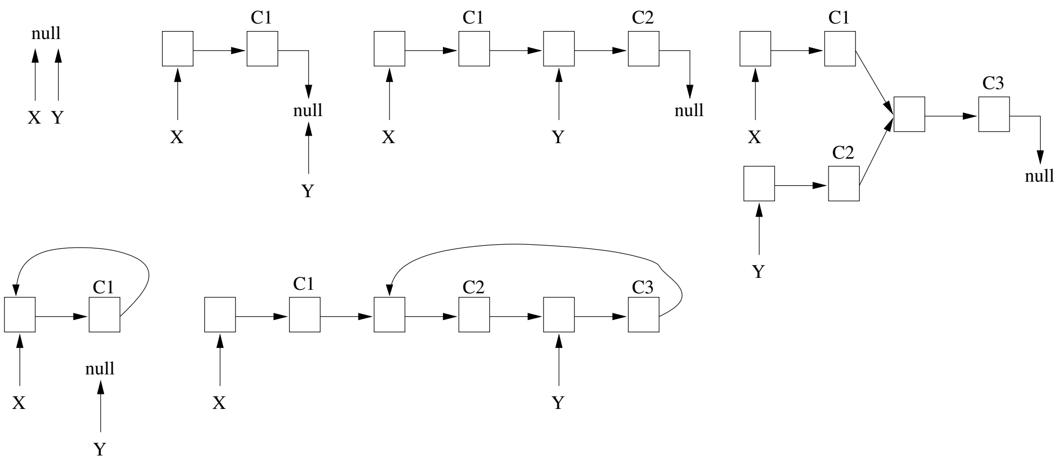


Figure 4.1: Examples of abstract graphs with counters

describing the input data structure, the automaton describing the procedure, and the bad property automaton is performed. The resulting automaton is checked for emptiness. Parity tree automata allow one to encode and check properties like reachability of a memory node, cycles in the memory, sharing, dangling pointers, null pointer dereferences, etc.

## Counter Automata

The use of counter automata (CA) as a model for 1-selector linked structures was preliminary studied in [BAN04, BI05] and then used in [BBH$^+$06, BFL06]. The data structure is abstracted in the following way: A sequence of nodes which are (i) direct successors, (ii) not pointed by any pointer variable, and (iii) not shared, is replaced by one node with an attached counter to carry information about the number of collapsed nodes. If one omits values of the counters, then there is a finite number of abstract graphs for a finite number of pointer variables. Examples of such abstract graphs for two pointer variables $x$, and $y$ are displayed in Figure 4.1 (three counters $C1$, $C2$, and $C3$ are used to carry information about lengths of corresponding segments).

One can then encode the semantics of a list manipulating program as a CA such that (i) the reachable abstract graphs are used as a basics of the control states of the CA (together with program lines), and (ii) counters are used to track lengths of segments. This CA is bisimular to the original program and therefore each common property valid on CA is valid also on the original pointer system and vice versa. This allows one to use some model checker for counter systems to prove safety as well as termination properties of the constructed CA.

### 4.1.3 Other Techniques

#### Memory Patterns

In [YKB02], a method for verification of programs manipulating dynamic-linked data structures was proposed. This method is based on the so-called *memory patterns*. A memory pattern is a finite graph representing a small part of a graph of memory structure (called shape graph). Then, an unbounded number of adjacent occurrences of this pattern in the shape graph is replaced by one *summary node*. Originally, the patterns were provided by the user of the method. In [EV05, ČEV06, ČEV07a], a heuristic method for an automatic generation of patterns was proposed. This heuristic method works for structures with *linear skeleton* (including cyclic lists, doubly-linked lists, lists extended by head/tail pointers, etc.). Moreover, in general, memory patterns can be used for more complicated data structures too. Recently, a first attempt to generalize the approach to trees was done in [ČEV07b].

#### Graph Rewriting

Graph rewriting is a quite general framework which can be used also in verification of programs manipulating dynamic data structures described as the so-called *graph transition systems* (GTS). In [BCK01], a method for analyzing reachability in GTS was proposed. It is based on the so-called *Petri graphs*. A Petri graph is a hypergraph $G_{\mathcal{P}}$ with an associated Petri net $P_{\mathcal{P}}$. There is a one-to-one mapping between hyper-edges of $G_{\mathcal{P}}$ and places in the Petri net $P_{\mathcal{P}}$. An overapproximation of the behaviour of a GTS is then defined as a pair $(P, \iota)$, where $P$ is a petri graph, and $\iota : G \rightarrow G_{\mathcal{P}}$ is a graph morphism from the set of all graphs to the graph $G_{\mathcal{P}}$. A graph $G$ is reachable (coverable) in $(P, \iota)$, if the set of edges of $\iota(G)$ corresponds to a reachable (coverable) marking in the underlaying Petri net $P_{\mathcal{P}}$. The initial marking of the underlaying Petri net $P_{\mathcal{P}}$ corresponds to the initial set of graphs (via the morphism $\iota$).

A reachability analysis for Petri nets can be applied to obtain the overapproximation of the set of reachable graphs. This method was implemented and applied for verification of insertion into red-black trees [BCE$^+$05]. In [KK06], the authors use a CEGAR loop to make the abstraction of GTS into the pair $(P, \iota)$ more precise in the case, when a spurious counterexample is encountered.

#### Grammar-Based Shape Analysis

The method presented in [LYY05] uses *context-free grammars with a single attribute in combination with contracted shape graphs*. The key idea is to contract a part of a shape graph, which is not directly pointed by some pointer variable and which is not shared, into a single summary node with an attached nonterminal. Then the part of the shape graph, hidden behind this summary node, is described by the grammar rules starting with the attached nonterminal.

If the execution of a program statement need to modify the contracted part, the summary node is (partially) unfolded according to the grammar rules, and the statement is performed on the exact part of the shape graph. After the execution of the statement, the new data structure is contracted (folded). Grammar rules are automatically derived during the contraction of the shape graph. Then a widening is defined on grammar rules – it is done by merging of nonterminals or a nonterminal with a terminal representing null. The method was implemented and tested on some linked-list and tree examples as creation of DLL, creation of tree with parent pointers, Schorr-Waite tree traversal, etc.

### Contracted Shape Graphs

In [LAIS06], the authors propose another encoding of memory graphs in order to verify safety properties. It allows one to encode structures without cycles and with a limited sharing. On the top, it is allowed to have some *simple loops* to encode structures as cyclic singly-linked lists, (cyclic) doubly-linked lists and trees with parent pointers. Moreover, it is allowed to temporary break this restriction inside loop boundaries. The main idea is to represent parts of a memory structure which are not directly pointed by some pointer variable and which are not shared, by summary nodes. Instead of edges in-between the nodes collapsed to a summary node, self-loop edges are attached to the corresponding summary node. Authors proposed a way, how to manipulate these structures according to program statements. The method computes an overapproximation of the reachable set and shape properties (specified as formulae in first-order logic) are checked. The method was in a prototype ways implemented an tested on a set of examples with trees (inserting and deleting from a search tree, insertion into red-black trees, etc.) and linked lists (insertion, deletion, reversion, Bublesort, Insertsort, etc.).

### Boolean Heaps

In [WKZ$^+$06], the authors combine several decision procedures and theorem provers with some newly added concepts to create an approach for verification of programs with pointer manipulations. Heaps are abstracted as *boolean-heaps*. A boolean heap is a disjunction of universally quantified boolean combinations of unary predicates over heap nodes. The links between heap nodes are hidden inside unary predicates valid for a particular node. The representation is extended by nullary predicates with a meaning such as two pointer variables points to an equal node, a pointer variable is null, etc. A concrete heap is represented by a boolean heap, if one of the top-level disjunctions is a formula, which satisfies the concrete heap. Predicates are partly generated automatically and partly provided by the user. The whole approach was implemented as a verification tool and was tested on a set of experiments (list reversion, list sorting, adding into a tree, skiplists, etc.).

### Local Abstraction

The work [CR07] presents a different approach for representing a heap. Instead of modeling the entire configuration of the memory, it captures just small parts around all pointer variables. The model contains (1) all nodes directly pointed by a pointer variable – the *tracked cells*, (2) all neighbours of these nodes, i.e. nodes directly pointed by some next pointer from a tracked cell. Moreover, each cell (of the type (1) or (2)) contains a set of counters which track how many next pointers (of a given type) point to a given cell – the *reference counts*. In case of a move of a pointer variable (e.g., via a statement `y=x.next`), one needs to guess a part of the new local neighborhood and set the reference counts. (The local neighborhood of new pointer positions is not necessary contained in the previous local configuration.) A systematic way of guessing and checking the correctness of the guess is proposed. Due to the analysis, there also appear nodes which lost the property (1) or (2). These nodes are then removed from the model. A set of experiments with doubly-linked lists was performed including insertion, deletion, copying, concatenation, etc.

## 4.2 ARTMC of Programs with Recursive Data Structures: The Basic Framework

We now describe the class of programs, properties, and the verification problem considered in our original approach to verification of programs with dynamic data structures based on ARTMC.

### 4.2.1 The Considered Programs

$l, l_1, l_2 \in \mathit{Lab},\ x, y, z \in \mathcal{V},\ d \in \mathcal{D},$
$\quad \mathit{next} \in \mathcal{S}$
$\mathit{Program} := \{l : \mathit{Stmnt};\}^*$
$\mathit{Stmnt} := \mathit{IfStmnt} \mid \mathit{Update} \mid \mathit{Asgn} \mid \mathit{Goto}$
$\mathit{IfStmnt} := \texttt{if}\ (\mathit{Cond})\ \texttt{then goto}\ l_1;$
$\qquad \texttt{else goto}\ l_2;$
$\mathit{Cond} := x == y \mid x == \texttt{null} \mid$
$\qquad x\texttt{->data} == d$
$\mathit{Update} := x = \texttt{malloc}() \mid \texttt{free}(x)$
$\mathit{Asgn} := x = y \mid x = \texttt{null} \mid x = y\texttt{->}\mathit{next}$
$\qquad x\texttt{->}\mathit{next} = y \mid x\texttt{->data} = d$
$\mathit{Goto} := \texttt{goto}\ l$

```
// Doubly-Linked Lists
typedef struct {
    DLL *next, *prev;
} DLL;

DLL *DLL_reverse(DLL *x) {
    DLL *y,*z;
    z = NULL;
    y = x->next;
    while (y!=NULL) {
        x->next = z;
        x->prev = y;
        z = x; x = y;
        y = x->next;
    }
    return x;
}
```

Figure 4.2: (a) Abstract syntax of considered programs and (b) Reversing a DLL

We consider standard, non-recursive C programs manipulating dynamic linked data structures (with possibly *several* next pointer selectors). We do not consider pointer arithmetics, and we suppose all non-pointer data to be abstracted to a finite domain by some of the existing techniques before our method is applied. The abstract syntax of the considered programs is given in figure 4.2(a), where *Lab* is a finite set program labels (one for each control location), $\mathcal{V}$ is a finite set of pointer variables, $\mathcal{D}$ is a finite set of data values,

```
x = aDLLHead;
while (x != NULL && random())
    x = x->next;
if (x != NULL && x->next->prev != x)
    error();
```

Figure 4.3: Checking the consistency of the next and previous pointers

and $\mathcal{S}$ is a finite set of selectors. We suppose some further, commonly used statements (such as `while` loops or nested dereferences) to be encoded by the listed statements. An example of a typical program that our method can handle is the reversion of doubly-linked lists (DLL) shown in Figure 4.2(b), which we also use as our running example.

#### Memory Configurations

Memory configurations of the considered programs with a finite set of pointer variables $\mathcal{V}$, a finite set of selectors $\mathcal{S} = \{1, ..., k\}$, and a finite domain $\mathcal{D}$ of data stored in dynamically allocated memory cells can be described as shape graphs of the following form. A *shape graph* is a tuple $SG = (N, S, V, D)$ where $N$ is a finite set of memory nodes, $N \cap \{\perp, \top\} = \emptyset$ (we use $\perp$ to represent null, and $\top$ to represent an undefined pointer value), $N_{\perp,\top} = N \cup \{\perp, \top\}$, $S : N \times \mathcal{S} \to N_{\perp,\top}$ is a successor function, $V : \mathcal{V} \to N_{\perp,\top}$ is a mapping that defines where the pointer variables are currently pointing to, and $D : N \to \mathcal{D}$ defines what data are stored in the particular memory nodes.

### 4.2.2 The Considered Properties

First of all, the properties we intend to check include *basic consistency of pointer manipulations*, i.e. absence of null and undefined pointer dereferences and references to already deleted nodes. Further, we would like to check various *shape invariance properties* (such as absence of sharing, acyclicity, or, e.g., the fact that if `x->next == y` (and `y` is not null) in a DLL, then also `y->prev == x`, etc.). To define such properties we propose two approaches described below.

#### Shape Testers

First, we use the so-called shape testers written in the C language. They can be seen as instrumentation code trying to detect violations of the memory shape properties at selected control locations of the original program. We extend slightly the C language used by the possibility of following next pointers backwards and by non-deterministic branching. For our verification tool, the testers are just a part of the code being verified. An error is announced when a line denoted by an error label is reached. This way, we can check a whole range of properties (including acyclicity, absence of sharing and other shape invariants as the relation of next and previous pointers in DLLs—cf. Fig 4.3).

The shape testers can be either directly written by the user, or they can be generated from a more declarative specification based, e.g., on the special purpose logic for describing bad shapes that we propose below.

In theory, bad shapes may be described directly using a tree automata memory encoding. The problem is not to miss any of their possible encodings. A safe solution is to encode good configurations and take the complement—then, errors cannot be missed, but if some possible encoding of a good configuration is omitted, false alarms may be reported. These problems do not arise when using shape testers as in their case, reachability of a certain line is only tested (and the choosing of a suitable encoding is subject to the automatic abstraction refinement).

**A Logic of Bad Memory Patterns**

Second, in order to allow the undesired violations of the memory shape properties to be specified more easily, we propose a logic-based specification language—namely, a *logic of bad memory patterns* (LBMP)—that is a fragment of the existential first order logic on graphs with (regular) reachability predicates (and an implicit existential quantification over paths). When defining the logic, our primary concern is not to obtain a decidable logic but rather to obtain a logic whose formulae may be automatically translated to the above mentioned C testers allowing us to *efficiently* test whether some bad shapes may arise from the given program by testing reachability of a designated error control line of a tester. The complete description of the LBMP is located in Section 4.3.

### 4.2.3 The Verification Problem

Our verification problem is model checking of the described undesirable existential properties against the given program. Above, we explain that for the specification of a violation of shape invariants, we use shape testers or LBMP whose formulae are translated into shape testers. For shape testers, we need to check unreachability of their designated error location. Moreover, we model all program statements such that if some basic memory consistency error (like a null pointer assignment) happens, the control is automatically transferred to a unique error control location. Thus, we are in general interested in *checking unreachability of certain error control locations* in a program.

## 4.3 Logic of Bad Memory Patterns

*Logic of bad memory patterns* (LBMP) is a fragment of the existential first order logic on graphs with (regular) reachability predicates (and an implicit existential quantification over paths). It was proposed to easy creation of testers controlling shape properties of a data structure. Formulae in this logic can be automatically translated into the C testers, which are attached at the end of the verified procedure. In case of an invariant violation, the tester reaches an error location.

### 4.3.1 Syntax of LBMP

Let $\mathcal{V}$ be a finite set of program variables and $\mathcal{S}$ a finite set of selectors. The *formulae of LBMP* have the form $\Phi ::= \exists w_1, ...w_n.\varphi$ where $\mathcal{W} = \{w_1, ..., w_n\}$, $\mathcal{V} \cap \mathcal{W} = \emptyset$, is a set of formulae variables, $\varphi ::= \varphi \vee \varphi \mid \psi$, $\psi ::= \psi \wedge \psi \mid x\varrho y$, $x, y \in \mathcal{V} \cup \mathcal{W}$, and $\varrho$ is a reachability formula defined below. To simplify the formulae, we allow $y$ in $x\varrho y$ to be skipped if it is not referred to anywhere else. We suppose such a missing variable to be implicitly added and existentially quantified. Given a $\psi$ formula, we define its associated graph to be the graph $G_\psi = (\mathcal{V} \cup \mathcal{W}, E)$ where $(x, y) \in E$ iff $x\varrho y$ is a conjunct in $\psi$. To avoid guessing in the tester corresponding to a formula, we require $G_\psi$ of every top level $\psi$ formula to have all nodes reachable from elements of $\mathcal{V}$.

An LBMP *reachability formula* has the form $\varrho ::= \xrightarrow{s} \mid \xleftarrow{s} \mid \varrho + \varrho \mid \varrho.\varrho \mid \varrho^* \mid [\sigma]$ where $s \in \mathcal{S}$ and $\sigma$ is a local neighborhood formula. Finally, an LBMP *local neighborhood* formula has the form $\exists u_1, ..., u_m.BC(x \xrightarrow{s} y, x = y)$ where $\mathcal{U} = \{u_1, ..., u_m\}$ is a set of local formula variables, $\mathcal{U} \cap (\mathcal{V} \cup \mathcal{W} \cup \{p\}) = \emptyset$, $p \notin \mathcal{V} \cup \mathcal{W}$, $s \in \mathcal{S}$, $x \in \mathcal{V} \cup \mathcal{W} \cup \mathcal{U} \cup \{p\}$, $y \in \mathcal{V} \cup \mathcal{W} \cup \mathcal{U} \cup \{p, \perp, \top\}$, and $BC$ is the Boolean closure. Here, $\perp$ represents `NULL`, $\top$ an undefined value, and $p$ is a special variable that always represents the *current position* in a shape graph. Moreover, to avoid guessing in the evaluation of the local neighborhood formulae, we require that if $\sigma$ is transformed into $\sigma'$ in DNF, and we construct a graph based on the positive $\xrightarrow{s}$ literals for each disjunct of $\sigma'$, each node of such a graph is reachable from $p$.

### 4.3.2 Semantics of LBMP

To define the *semantics of the LBMP formulae*, let us start with local neighborhood formulae. Suppose we have a shape graph $SG = (N, S, V, D)$ (cf. Section 4.2.1) with a set of pointer variables $\mathcal{V}$ and that we use formula variables $\mathcal{W} = \{w_1, ..., w_n\}$ and local formula variables $\mathcal{U} = \{u_1, ..., u_m\}$. We say that an LBMP local neighborhood formula $\sigma = \exists u_1, ..., u_m.BC(x \xrightarrow{s} y, x = y)$ holds in $SG$ for a valuation of the formula variables $W : \mathcal{W} \rightarrow N_{\perp,\top}$ and a current position $P : \{p\} \rightarrow N_{\perp,\top}$, denoted $SG, W, P \models \sigma$, iff there exists a valuation of the local formula variables $U : \mathcal{U} \rightarrow N_{\perp,\top}$ such that for $\nu = V \cup W \cup P \cup U$, $BC(x \xrightarrow{s} y, x = y)$ holds when we interpret its atomic formulae in such a way that $x \xrightarrow{s} y$ holds iff $\nu(x) \in N \wedge S(\nu(x), s) = \nu(y)$, and $x = y$ holds iff $\nu(x) = \nu(y)$.

The alphabet of an LBMP reachability formula $\varrho$, $\Sigma(\varrho)$, is the set of the $\xrightarrow{s}$, $\xleftarrow{s}$, and $[\sigma]$ terms of $\varrho$. The language of $\varrho$, $L(\varrho)$, is the regular language defined by $\varrho$ when interpreted as a regular expression over $\Sigma(\varrho)$. We say that a reachability formula $\varrho$ is satisfied in $SG$ between variables $x, y \in \mathcal{V} \cup \mathcal{W}$ for a valuation $W$ of the formula variables $\mathcal{W}$, denoted $SG, W \models x\varrho y$ iff there exists a sequence $n_1 r_1 n_2 ... r_l n_{l+1} \in N_{\perp,\top}(\Sigma(\varrho) N_{\perp,\top})^*$ such that (1) $r_1 r_2 ... r_l \in L(\varrho)$, (2) $(V \cup W)(x) = n_1$, (3) $(V \cup W)(y) = n_{l+1}$, and (4) the following holds for all $i \in \{1, ..., l\}$:

- if $r_i = \xrightarrow{s}$ for some $s \in \mathcal{S}$, $n_i \in N \wedge S(n_i, s) = n_{i+1}$,
- if $r_i = \xleftarrow{s}$ for some $s \in \mathcal{S}$, $n_{i+1} \in N \wedge S(n_{i+1}, s) = n_i$, and

- if $r_i = [\sigma]$, then $n_i = n_{i+1}$ and $SG, W, P \models \sigma$ for $P : \{p\} \to n_i$.

Finally, we say that an LBMP formula $\Phi = \exists w_1, ...w_n. \bigvee_i \bigwedge_j x_{ij}\varrho_{ij}y_{ij}$ is *satisfied in a shape graph* $SG = (N, S, V, D)$, i.e. $SG \models \Phi$, iff there exists a valuation $W : \mathcal{W} \to N_{\perp,\top}$ of the formula variables $\mathcal{W} = \{w_1, ..., w_n\}$ such that $\exists i \forall j. SG, W \models x_{ij}\varrho_{ij}y_{ij}$.

### 4.3.3 An Example: Doubly-Linked Lists

We illustrate the semantics of LBMP formulae on several examples expressing undesirable phenomena that we would like to avoid when manipulating *acyclic doubly-linked lists*. In their case, it is undesirable if one of the following happens after some operation (as, for instance, reversion) on a given list—we suppose the resulting list to be pointed via the program variable $l$:

1. the list does not end with null, which can be tested via $l \overset{n}{\rightarrow}^* [p = \top]$,
2. the predecessor of the first element is not null, which corresponds to $l[\neg(p \overset{b}{\rightarrow} \perp)]$,
3. the predecessor of the successor of a node $n$ is not $n$, which can be detected via the formula $l \overset{n}{\rightarrow}^* [\exists x.\, p \overset{n}{\rightarrow} x \,\wedge\, x \neq \perp \,\wedge\, \neg(x \overset{p}{\rightarrow} p)]$, or
4. the list is cyclic, i.e. $\exists x.\, l \overset{n}{\rightarrow}^* [p = x] \overset{n}{\rightarrow}\overset{n}{\rightarrow}^* [p = x]$. (Note that this property is in fact implied by items 2 and 3.)

All the given formulae can be joint by disjunction into a single LBMP formula. More examples of LBMP formulae can be found in section describing our case studies 4.5.2.

### 4.3.4 Evaluating LBMP Specifications

As we have already mentioned above, LBMP is designed such that its formulae can be easily transformed to a tester that can be encoded in a slightly extended C code, put after the program being verified, and used in an efficient way for checking safety of the given program. The needed extension of C includes non-deterministic branching and the possibility of following next pointers backwards. Both of these features may efficiently be handled in our verification framework. We describe here the idea of translating LBMP formulae to a C code.

The *disjunction* $\varphi ::= \varphi \vee \varphi \mid \psi$ in LBMP formulae $\Phi ::= \exists w_1, ...w_n.\varphi$ can be implemented via non-deterministic branching. Next, when transforming the *conjunctions* $\psi ::= \psi \wedge \psi \mid x\varrho y$, we construct the graph $G_\psi$ associated to a top level formula $\psi$ in Section 4.3.1. We can then start from the memory nodes pointed to by program variables and follow the arcs of the graph when evaluating the particular reachability formulas $x\varrho y$. This way we step by step find suitable values for the particular formula variables (if such values exist) without having to guess any nodes in the shape graph—of course, provided we do not guess in evaluating $x\varrho y$, which we do not as explained below.

When translating *reachability formulae* $\varrho ::= \overset{s}{\rightarrow} \mid \overset{s}{\leftarrow} \mid \varrho + \varrho \mid \varrho.\varrho \mid \varrho^* \mid [\sigma]$, we transform $\overset{s}{\rightarrow} \mid \overset{s}{\leftarrow}$ to following the appropriate selectors forward or backward from the current position in the shape graph (for which we can use the transducers described in Section 4.4.3 either directly or in a reverse way). The evaluation of $\varrho + \varrho$ corresponds to non-deterministic branching, the evaluation of $\varrho.\varrho$ is translated simply to a sequence of evaluations, and the evaluation of $\varrho^*$ is done again via non-deterministic branching (we either evaluate $\varrho$ once again or we finish the evaluation of $\varrho^*$).

Finally, when evaluating a *local neighborhood formula* $\sigma = \exists u_1, ..., u_m.BC(x \overset{s}{\rightarrow} y, x = y)$, we may transform it into DNF, transform the top level disjunction into a non-deterministic branching, and drive the evaluation of the chosen conjunction by its graph based on the positive appearances of the $x \overset{s}{\rightarrow} y$ literals. Due to the definition of LBMP, by following (forward or backward) the $x \overset{s}{\rightarrow} y$ links of the positive literals, we may find possible values of all the local formula variables and then just check whether the negative $x \overset{s}{\rightarrow} y$ literals and the $x = y$ literals hold.

Note that in the described evaluation of LBMP formulae, we avoid any guessing of positions in the shape graphs, which can be potentially very costly as it can lead to having to consider many different relative positions of the chosen values of formula variables (often not sensible for the satisfaction of the given formula). The only involved non-determinism is in the non-deterministic branching of the generated C tester that is resolved by evaluating the branches in some chosen order.

## 4.4 Tree Automata Encoding of Pointer Manipulating Programs

In this section, we describe the encoding of memory configurations into trees and tree automata and the encoding of program statements into tree transducers that we propose. This encoding allows us to apply *abstract regular tree model checking* (ARTMC, cf. Chapter 3) – our automatic verification framework.

### 4.4.1 Encoding of Sets of Memory Configurations

As was described in Section 4.2.1, memory configurations of the considered programs with a finite set of pointer variables $\mathcal{V}$, a finite set of selectors $\mathcal{S} = \{1, \ldots, k\}$, and a finite domain $\mathcal{D}$ of data stored in dynamically allocated memory cells can be described as *shape graphs* $SG = (N, S, V, D)$. We suppose $\top \in \mathcal{D}$—the data value $\top$ is used to denote "zombies" of deleted nodes, which we keep and detect all erroneous attempts to access them.

To be able to describe the way we encode sets of shape graphs using tree automata, we first need a few auxiliary notions. First, to allow for dealing with more general shape graphs than tree-like, we do not simply identify the next pointers with the branches of the trees accepted by tree automata. Instead, we use the tree structure just as a backbone over which links between the allocated nodes are expressed using the so-called *routing expressions*, which are regular expressions over directions in a tree (like move up, move left down, etc.) and over the nodes that can be seen on the way. From nodes of the trees described by tree automata, we refer to the routing expressions via some symbolic names called *pointer descriptors* that we assign to them—we suppose dealing with a finite set of pointer descriptors $\mathcal{R}$. Moreover, we couple each pointer descriptor with a unique *marker* from a set $\mathcal{M}$ (and so $||\mathcal{R}|| = ||\mathcal{M}||$). The routing expressions may identify several target nodes for a single source memory node and a single selector. Markers associated with the target nodes can then be used to decrease the non-determinism of the description (only nodes marked with the right marker are considered as the target).

Let us now fix the sets $\mathcal{V}$, $\mathcal{S}$, $\mathcal{D}$, $\mathcal{R}$, and $\mathcal{M}$. We use a *ranked alphabet* $\Sigma = \Sigma_2 \cup \Sigma_1 \cup \Sigma_0$ consisting of symbols of ranks $k = ||\mathcal{S}||$, 1, and 0. Symbols of rank $k$ represent allocated memory nodes that may be pointed by some pointer variables, may be marked by some markers as targets of some next pointers, they contain some data and have $k$ next pointers specified either as null, undefined, or via some next pointer descriptor. Thus, $\Sigma_2 = 2^{\mathcal{V}} \times 2^{\mathcal{M}} \times \mathcal{D} \times (\mathcal{R} \cup \{\bot, \top\})^k$. Given an element $n \in \Sigma_2$, we use the notation $n.var$, $n.mark$, $n.data$, and $n.s$ (for $s \in \mathcal{S}$) to refer to the pointer variables, markers, data, and descriptors associated with $n$, respectively. $\Sigma_1$ is used for specifying nodes with undefined and null pointer variables, and so $\Sigma_1 = 2^{\mathcal{V}}$. Finally, in our trees, the leaves are all the same (with no special meaning), and so $\Sigma_0 = \{\bullet\}$.

We can now specify the *tree memory backbones* we use to encode memory configurations as the trees that belong to the language of the tree automaton with the following rules[1]: (1) $\bullet \rightarrow q_i$, (2) $\Sigma_2(q_i/q_m, ..., q_i/q_m) \rightarrow q_m$, (3) $\Sigma_1(q_m/q_i) \rightarrow q_n$, and (4) $\Sigma_1(q_n) \rightarrow q_u$. Intuitively, $q_i$, $q_m$, $q_n$, and $q_u$ are automata states, where $q_i$ accepts the leaves, $q_m$ accepts the memory nodes, $q_n$ accepts the node encoding null variables, and $q_u$, which is the accepting state, accepts the node with undefined variables. Note that there is always a single node with undefined variables, a single node with null variables, and then a sub-tree with the memory allocated nodes. Thus, every memory tree $t$ can be written as $t = undef(null(t'))$ for $undef, null \in \Sigma_1$. We say a memory tree $t = undef(null(t'))$ is *well-formed* if the pointer variables are assigned unique meanings, i.e. $undef \cap null = \emptyset \wedge \forall p \in \mathcal{N}lPos(t') : t'(p).var \cap (null \cup undef) = \emptyset \wedge \forall p_1 \neq p_2 \in \mathcal{N}lPos(t') : t'(p_1).var \cap t'(p_2).var = \emptyset$ where $\mathcal{N}lPos$ are non-leaf positions—cf. Section 3.1.

We let $\mathcal{S}^{-1} = \{s^{-1} \mid s \in \mathcal{S}\}$ be a set of "inverted selectors" allowing one to follow the links in a shape graph in a reverse order. A *routing expression* may then be formally defined as a regular expression on pairs $s.p \in (\mathcal{S} \cup \mathcal{S}^{-1}).\Sigma_2$. Intuitively, each pair used as a basic building block of a routing expression describes one step over the tree memory backbone: we follow a certain branch up or down and then we should see a certain node (most often, we will use the node components of routing expressions to check whether a certain marker is set in a particular node).

A *tree memory encoding* is a tuple $(t, \mu)$ where $t$ is a tree memory backbone and $\mu$ a mapping from the set of pointer descriptors $\mathcal{R}$ to routing expressions over the set of selectors $\mathcal{S}$ and the memory node alphabet $\Sigma_2$ of $t$. An example of a tree memory encoding for a *doubly-linked list* (DLL) is shown in Fig. 4.4.

Let $(t, \mu)$, $t = undef(null(t'))$, be a tree memory encoding with a set of selectors $\mathcal{S}$

[1]If we put a set into the place of the input symbol in a transition rule, we mean we can use any element of the set. Moreover, if we use $q_1/q_2$ instead of a single state, one can take either $q_1$ or $q_2$, and if there is a $k$-tuple of states, one considers all possible combinations of states.

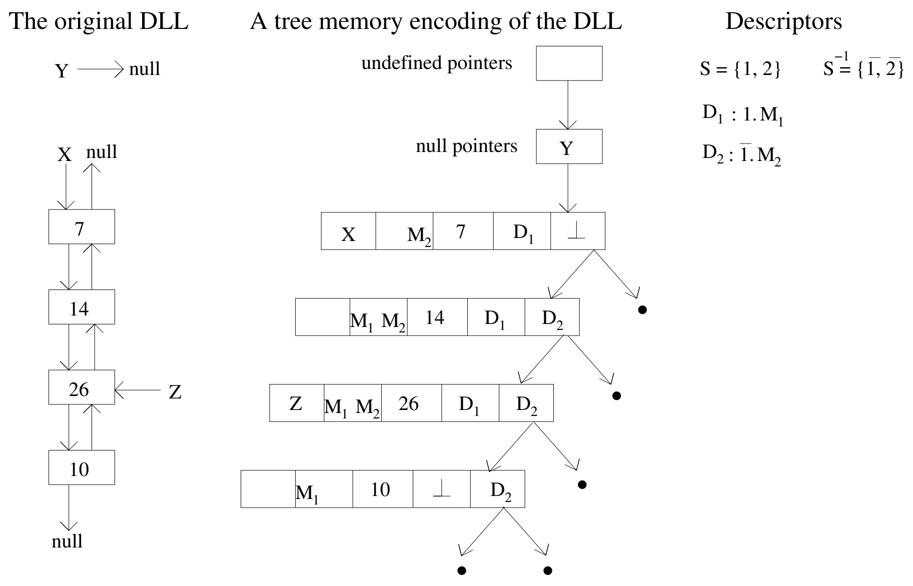


Figure 4.4: An example of a tree memory encoding—a doubly linked list (DLL)

and a memory node alphabet $\Sigma_2$. We call $\pi = p_1 s_1 ... p_l s_l p_{l+1} \in \Sigma_2.((\mathcal{S} \cup \mathcal{S}^{-1}).\Sigma_2)^l$ a *path in* $t$ of length $l \geq 1$ iff $p_1 \in \mathcal{P}os(t')$ and $\forall i \in \{1, ..., l\} : (s_i \in \mathcal{S} \wedge p_i.s_i = p_{i+1} \wedge p_{i+1} \in \mathcal{P}os(t')) \vee (s_i \in \mathcal{S}^{-1} \wedge p_{i+1}.s_i = p_i)$. For $p, p' \in \mathcal{N}l\mathcal{P}os(t')$ and a selector $s \in \mathcal{S}$, we write $p \xrightarrow{s} p'$ iff (1) $t'(p).s \in \mathcal{R}$, (2) there is a path $p_1 s_1 ... p_l s_l p_{l+1}$ in $t$ for some $l \geq 0$ such that $p = p_1, p_{l+1} = p'$, and (3) $s_1 t'(p_2) ... t'(p_l) s_l t'(p_{l+1}) \in \mu(t'(p).s)$.

The *set of shape graphs represented by a tree memory encoding* $(t, \mu)$ with $t = undef(null(t'))$ is denoted by $[\![(t, \mu)]\!]$ and given as all the shape graphs $SG = (N, S, V, D)$ for which there is a bijection $\beta : \mathcal{P}os(t') \to N$ such that:

1. $\forall p, p' \in \mathcal{N}l\mathcal{P}os(t') \, \forall s \in \mathcal{S} : (t'(p).s \notin \{\bot, \top\} \wedge p \xrightarrow{s} p') \Leftrightarrow S(\beta(p), s) = \beta(p')$.
(The links between memory nodes are respected.)

2. $\forall p \in \mathcal{N}l\mathcal{P}os(t') \, \forall s \in \mathcal{S} \, \forall x \in \{\bot, \top\} : t'(p).s = x \Leftrightarrow S(\beta(p), s) = x$.
(Null and undefined successors are respected.)

3. $\forall v \in \mathcal{V} \, \forall p \in \mathcal{P}os(t') : v \in t'(p).var \Leftrightarrow V(v) = \beta(p)$.
(Assignment of memory nodes to variables is respected.)

4. $\forall v \in \mathcal{V} : (v \in null \Leftrightarrow V(v) = \bot) \wedge (v \in undef \Leftrightarrow V(v) = \top)$.
(Assignment of null and undefinedness of variables are respected.)

5. $\forall p \in \mathcal{N}l\mathcal{P}os(t') \, \forall d \in \mathcal{D} : t'(p).data = d \Leftrightarrow D(\beta(p)) = d$.
(Data stored in memory nodes is respected.)

A *tree automata memory encoding* is a tuple $(A, \mu)$ where $A$ is a tree automaton accepting a regular set of tree memory backbones and $\mu$ is a mapping as above. Naturally, $A$ represents the set of shape graphs defined by $[\![(A, \mu)]\!] = \bigcup_{t \in L(A)} [\![(t, \mu)]\!]$.

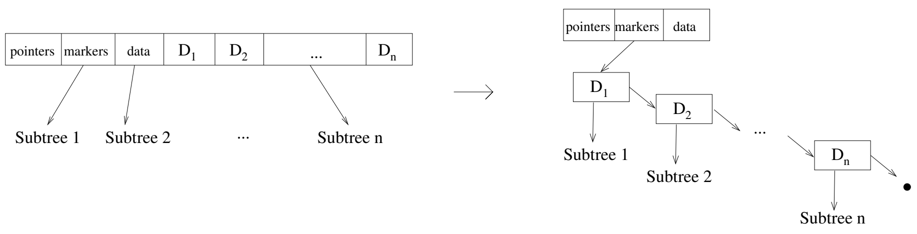


Figure 4.5: Splitting memory nodes in Mona into data and next pointer nodes

**Remarks**

Let us now discuss a bit more properties of the described encoding from the point of view of manipulating it in ARTMC. ARTMC syntactically manipulates tree automata $A$ whose languages can be interpreted as shape graphs using our encoding. Notice that $(A, \mu)$ and $[\![(A, \mu)]\!]$ are two different notions since the encoding is not canonical as a given shape graph can be possibly obtained by several different tree memory encodings. In Section 4.4.3, we argue that program statements can, nevertheless, be encoded faithfully as tree transducers. Another important property of the encoding is that given a tree automata memory encoding $(A, \mu)$, the set $[\![(A, \mu)]\!]$ can be empty although $L(A)$ is not empty (since the routing expressions can be incompatible with the tree automaton). Of course, if $L(A)$ is empty, then $[\![(A, \mu)]\!]$ is also empty. Therefore, checking emptiness of $[\![(A, \mu)]\!]$ (which is important for applying the ARTMC framework, see Section 4.4.4) can be done in a sound way by checking emptiness of $L(A)$.

## 4.4.2 Tree Memory Configurations in Mona

In our implementation, we use the tree automata library from the Mona project [KM01]. As the library supports binary trees only, and we need n-ary ones, we *split each memory node* labelled with $\Sigma_2 = 2^{\mathcal{V}} \times 2^{\mathcal{M}} \times \mathcal{D} \times (\mathcal{R} \cup \{\bot, \top\})^k$ in the above definition of a tree memory encoding into a data node labelled with $2^{\mathcal{V}} \times 2^{\mathcal{M}} \times \mathcal{D}$ and a series of $k$ next pointer nodes, each labelled with $\mathcal{R} \cup \{\bot, \top\}$—cf. Figure 4.5.

As for the set of *pointer descriptors* $\mathcal{R}$, we currently fix it by introducing a unique pointer descriptor for each destructive update `x->`$s$ `= y` or `x->`$s$ `= new` that appears in the program. This is because they are the statements that establish new links among the allocated memory nodes. In addition, we might have some further descriptors if they are a part of the specification of the input configurations (see Section 4.4.4).

Further, in our Mona-based framework, we encode *routing expressions* using tree transducers. A transducer representing a routing expression $r$ simply copies the input tree memory backbone on which it is applied up to: (1) looking for a data node $n_1$ that is labelled with a special token $\blacklozenge \notin \mathcal{V} \cup \mathcal{M} \cup \mathcal{D}$ and (2) moving $\blacklozenge$ to a data node $n_2$ that is the target of the next pointer described by $r$ and that is also marked with the appropriate marker. As described in the next section, we can then implement program statements that follow the next pointers (e.g., `x = y->`$s$) by putting the token $\blacklozenge$ to a node pointed to by `x`, applying the transducer implementing the appropriate routing expression, and making y point to the node to which ⧫ was moved. Due to applying abstraction, the target may not always be unique—in such a case, the transducer implementing the routing expression simply returns a set of trees in which ⧫ is put to some target data node such that all possibilities where it can get via the given routing expression are covered.

Note that the use of tree transducers for encoding routing expressions allows us in theory to express more than using just regular expressions. In particular, we can refer to the tree context of the nodes via which the given route is going. In our current implementation, we, however, do not use this fact.

## 4.4.3 Encoding Program Statements as Tree Transducers

We encode every of the considered pointer-manipulating statements as a tree transducer. In the transducer, we expect the tree memory encoding to be extended by a new root symbol which corresponds to the *current program line* or to an error indication when an error is found during the analysis. We now briefly describe how the transducers corresponding to the program statements work. Each transducer is constructed in such a way that it simulates the effect of a program statement on a set of shape graphs represented by a tree automata memory encoding: if a shape $SG$ represented by a tree memory encoding is transformed by the program statement to a shape graph $SG'$, then the transducer transforms the tree memory encoding such that it represents $SG'$. This makes sure that although the encoding is non canonical (see end of Section 4.4.1), we simulate faithfully a program statement.

### Non-destructive Updates and Tests

The simplest is the case of the `x = NULL` assignment. The transducer implementing it just goes through the input tree and copies it to the output with the exception that (1) it removes `x` from the labelling of the node in which it currently is and adds it to the labelling of the $null$ node and (2) changes the current line appropriately. The transducer implementing an assignment `x = y` is similar, it just puts `x` not to the $null$ node, but to the node which is currently labelled by `y`.

The transducers for the tests of the form `if (x == null) goto l1; else goto l2;` are very similar to the above—they just do not change the node in which `x` is, but only change the current program line to either `l1` or `l2` according to whether or not `x` is in the $null$ node. If `x` is in $undef$, an error indication is used instead of `l1` or `l2`. The transducers for `if (x == y) goto l1; else goto l2;` are similar—they just test whether or not `x` and `y` appear in the same node (both different from $undef$).

The transducer for an `x = y->`$s$ statement is a union of several complementary actions. If `y` is in $null$ or $undef$, an error is indicated. If `y` is in a regular data node and its $s$-th next pointer node contains either $\perp$ or $\top$, the transducer removes `x` from the node it is currently in and puts it into the $null$ or $undef$ node, respectively. If `y` is in a regular data node $n$ and its $s$-th next pointer node contains some pointer descriptor $r \in \mathcal{R}$, the ⧫ token is put to $n$. Then, the routing expression transducer associated with $r$ is applied.

Finally, x is removed from its current node and put into the node to which ⧫ was moved by the applied routing expression transducer.

**Destructive Updates**

The destructive pointer update `x->`$s$` = y` is implemented as follows. If x is in $null$ or $undef$, an error is indicated. If x is defined and if y is in $null$ or $undef$, the transducer puts $\perp$ or $\top$ into the $s$-th next pointer node below x, respectively. Otherwise, the transducer puts the pointer descriptor $r$ associated with the particular `x->`$s$` = y` statement being fired into the $s$-th next pointer node below x, and it marks the node in which y is by the marker coupled with $r$. Then, the routing expression transducer associated with $r$ is updated such that it includes the path from the node of x to the node of y.

One could think of various strategies how to *extract the path* going from the node of x to the node of node y. Currently, we use a simple strategy, which is, however, successful in many practical examples as our experiments show: We extract the shortest path between x and y on the tree memory backbone, which consists of going a certain number of steps upwards to the closest common parent of x and y and then going a certain number of steps downwards. (The upward or the downward phase may also be skipped when going just down or up, respectively.) When extracting this path, we project away all information about nodes we see on the way and about nodes not directly lying on the path. Only the directions (left/right up/down) and the number of steps are preserved.

Note that we, in fact, perform the operation of routing expression extraction on a tree automaton, and we extract all possible paths between where x and y may currently be. The result is transformed into a transducer $\tau_{\mathrm{xy}}$ that moves the token ⧫ from the position of x to the position of y, and $\tau_{\mathrm{xy}}$ is then united with the current routing expression transducer associated with the given pointer descriptor $r$. The extraction of the routing paths is done partly by rewriting the input tree automaton via a special transducer $\tau_\pi$ that in one step identifies all the shortest paths between all x and y positions and projects away the non-necessary information about the nodes on the way. The transducer $\tau_\pi$ is simple: it just checks that we are going one branch up from x and one branch down to y while meeting in a single node. The transition relation of the resulting transducer is then post-processed by changing the context of the path to an arbitrary one which cannot be done by transducing in Mona where structure preserving transducers may only be used. More detailed description is located in Section 4.6.

**Dynamic Allocation and Deallocation**

The `x = malloc()` statement is implemented by rewriting the right-most ● leaf node to a new data node pointed to by x. Below the node, the procedure also creates all the $k$ next pointer nodes whose contents is set to $\top$.

In order to exploit the regularity that is always present in algorithms allocating new data structures, which typically add new elements at the end/leaves of the structure, we also explicitly support an `x.`$s$` = malloc()` statement. We even try to pre-process programs and compact all successive pairs of statements of the form `x = malloc();`

```
aDLLHead = malloc();
aDLLHead->prev = null;
x = aDLLHead;
while (random()) {
  x->next = malloc();
  x->next->prev = x;
  x = x->next;
}
x->next = null;
```

Figure 4.6: Generating DLLs

`y->`$s$ `= x` (provided `x` is not used any further) to `y->`$s$`= malloc()`. Such a statement is then implemented by adding the new element directly under the node pointed to by $y$ (provided it is a leaf) and joining it by a simple routing expression of the form "one level down via a certain branch". This typically allows us to work with much simpler and more precise routing expressions.

Finally, a `free(x)` statement is implemented by a transducer that moves all variables that are currently in the node pointed to by `x` to the $undef$ node (if `x` is in $null$ or $undef$, an error is indicated). Then, the node is marked by a special marker as a deleted node, but it stays in our tree memory encoding with all its current markers set. In addition to all the other tests mentioned above as done within the transducer implementing an `x = y->`$s$ assignment, it is also tested whether the target is not deleted—if so, an error is indicated.

### 4.4.4 Verification of Programs with Pointers using ARTMC

#### Input structures

We consider two possibilities how to encode the input structures. First, we can directly use the tree automata memory encoding—e.g., a tree automata memory encoding (with two pointer descriptors *next* and *prev* and the corresponding routing expressions) describing all possible doubly-linked lists pointed to by some program variable. Such an encoding can be provided manually or derived automatically from a description of the concerned linked data structure provided, e.g., as a graph type [KS93]. The main advantage is that the verification process starts with an exact encoding of the set of all possible instances of the considered data structure.

Another possible approach is to start with the unique "empty" shape graph where all variables are undefined. We can encode such a shape graph using a tree automata encoding where all variables are in $undef$, $null$ is empty, there are no other nodes, and all the routing expressions are empty. The set of structures on which the examined procedure should be verified is then supposed to be generated by a *constructor written in C* by the user (as, e.g., in Fig. 4.6). This constructor is then put before the verified procedure and the whole program is given to the model checker. The advantage is that no further notation is necessary. The disadvantage is that we have more code that is subject to the verification and the set of automatically obtained input structures need not be encoded in the optimal way leading to a slow-down of the verification.

#### Applying ARTMC

In Chapter 3, we have described the technique ARTMC. We have supposed that one transducer $\tau$ is used to describe the behaviour of the whole system. In the application described in this chapter, we use a variant of this approach by considering each program statement as one transducer. Then, we compute an overapproximation of the reachable configurations for each program line by starting from an initial set of shape graphs represented by a tree automata memory encoding and iterating the abstract fixpoint computation described in Chapter 3 through the program structure. The fixpoint computation stops if the abstraction $\alpha$ is finitary. In such a case, the number of the abstracted tree automata encoding sets of the memory backbones that can arise in the program being checked is finite. Moreover, the number of the arising routing expressions is also finite as they are extracted from the bounded number of the tree automata describing the encountered sets of memory backbones.[2]

During the computation, we check whether a designated error location in the program is reached or whether a fixpoint is attained. In the latter case, the property is satisfied (the error control location is not reachable). In the former case, we compute backwards to check if the counterexample is spurious as explained in Chapter 3. However, as said in Section 4.4.1, the check for emptiness is not exact and therefore we might conclude that we have obtained a real counterexample although this is not the case. Such a case does not happen in any of our experiments and could be detected by replaying the path from the initial configurations.

## 4.5 Implementation and Experimental Results

### 4.5.1 An ARTMC Tool for Tree Automata Memory Encodings

We have implemented the above proposed method in a prototype tool based on the Mona tree automata libraries [KM01]. As explained above, we implement the ARTMC framework not using a single transducer for the entire program but we keep the transducers for the particular program statements separate and we also compute a separate tree automaton for every program line encoding the memory configurations reachable at that line. We use a depth-first strategy when iterating the transducers corresponding to the particular program lines.

We have also refined the basic finite-height and predicate abstractions described in Section 3.3. In particular, we do not allow collapsing of data nodes with next pointer nodes, collapsing of next pointer nodes corresponding to different selectors, and we prevent the abstraction of allowing a certain pointer variable to point to several memory

[2]The non-canonicity of our encoding does not prevent the computation from stopping. It may just take longer since several encodings for the same graph could be added.

nodes at the same time. Experiments showed that it is better to collaps only at data nodes.

We have also proposed one new abstraction schema called the *neighbour abstraction*. Under this schema, only the tree automata states are collapsed that (1) accept equal data memory nodes with equal next pointer nodes associated with them and (2) that directly follow each other (are neighbours). This strategy is very simple, yet it proved useful in some practical cases.

Finally, we allow the abstraction to be applied either at all program lines or only at the loop closing points. In some cases, the latter approach is more advantageous due to some critical destructive pointer updates are done without being interleaved with abstraction. This way, we may avoid having to remove lots of spurious counterexamples that may otherwise arise when the abstraction is applied while some important shape invariant is temporarily broken.

## 4.5.2 Case Studies

We have performed a set of experiments with singly-linked lists (SLL), doubly-linked lists (DLL), trees, lists of lists, and trees with linked leaves. All three mentioned types of automata abstraction—the finite height abstraction (with the initial height being one), predicate abstraction (with no initial predicates), and neighbor abstraction—was proved useful (gave the best achieved result) in different examples. All examples were automatically verified for null/undefined/deleted pointer exceptions. Additionally, some further shape properties (such as absence of sharing, acyclicity, preservation of input elements, etc.) were verified in some case studies too.

In the following, we describe the particular verification case studies we did with the prototype implementation of our method. The names of the experiments used refer to the verification times presented in Table 4.1.

### Singly-Linked Lists

We consider SLLs as a starting point only as they can be handled by more light weight methods (including abstract regular word model checking [BHMV05]). We suppose the only next pointer of SLLs to be denoted as $n$.

**Creation of SLLs.** In this experiment, we consider a program that iteratively creates an SLL. As the initial set of configurations, we use the set describing the one-element SLL pointed by the variable $l$. During the creation, the tail of the list is pointed by the variable $tail$, and the new elements are attached behind it. We check that the following properties do not appear in resulting data structure (recall that LBMP formulae describe undesirable patterns):

- $l \xrightarrow{n}^* [p = \top]$. The end of the list is undefined.
- $\exists x.\, l \xrightarrow{n}^* [p = x] \xrightarrow{n}\xrightarrow{n}^* [p = x]$. The list becomes cyclic.

**Reversion of SLLs.** In this experiment, we start with the set of all possible SLLs and we use the "marking" method proposed in [BHMV05] to verify that the program implements a real reversion: i.e. no element is lost or added, and the elements are linked in exactly the opposite order as before applying the procedure.

The head of the list is pointed by the variable $l$. Before the reversion, we set the pointer variable $head$ to the head of the list. Then, we randomly choose an element of the list and point it by the variable $first$. The pointer variable $second$ is then set to the successor of $first$ (i.e., $second == first{-}{>}n$). The pointer variable $last$ is set to the tail of the list. The reversion procedure does not see these variables—its implementation is untouched. Thus, at the end of the reversion, they are pointing to exactly the same memory nodes as before the reversion. Just the order of these memory nodes in the SLL should change. It is easy to see that the reversion does not work properly if one of the following issues (or the ones mentioned in *creation of SLLs*) happen:

- $l[\neg(p = last)]$. The head of the list is not pointed by $last$.
- $second \xrightarrow{n} [\neg(p = first)]$. The order of the elements is not reversed correctly.
- $head \xrightarrow{n} [\neg(p = \bot)]$. The variable $head$ does not point to the end of the list.

**Doubly-Linked Lists**

We suppose the two selectors of DLLs to be denoted $n$ (next) and $p$ (previous).

**Deletion from DLLs.** We consider a program that removes an element from a list. The initial set describes all possible DLLs where the head is pointed by the variable $l$. We first search for an element to be deleted by going through the list from its beginning. As the data items are abstracted, we select a random node to be deleted. To ensure consistency of the resulting data structure, we use the LBMP formulae mentioned already in Sect. 4.3 (acyclicity is ensured by the 2nd, and 3rd rule):

- $l \xrightarrow{n}^{*} [p = \top]$. The tail of the list is undefined.
- $l[\neg(p \xrightarrow{p} \bot)]$. The predecessor of the first element is not null.
- $l \xrightarrow{n}^{*} [\exists x.\, p \xrightarrow{n} x \ \wedge\ x \neq \bot \ \wedge\ \neg(x \xrightarrow{p} p)]$. The list is not doubly-linked.

**Insertion into DLLs.** The program inserts a new element into a randomly chosen position in the list. The initial set describes all possible DLLs where the head is pointed by the variable $l$. We search the position for the insertion by going through the list from the head and stopping the search at a random place. The possibility of inserting a new element as a new head is also taken into account. After the insertion is performed, the same tests as in the example *delete from DLLs* are done.

**Reversion of DLLs.** The program reverses a given DLL. The initial set is the set of all possible DLLs where head is pointed by the variable $l$. We use the same method to check that the result is exactly the reversion of the input list as in the case of SLLs (and so we again introduce auxiliary variables $head$, $first$, $second$, and $last$). The LBMP formulae considered in *DLL-delete* and in *SLL-reverse* (in particular, the ones concerning the auxiliary variables) are used.

**Insertsort on DLLs.** The program implements the well-known insertion sort algorithm. The initial set of configurations contains all possible DLLs. We abstract from data values—the comparisons are done randomly. We check for null and undefined pointer exceptions.

**Binary Trees**

The two selectors of binary trees are denoted $l$ (left) and $r$ (right) in the following.

**Inserting into trees.** The program performs repeated insertions of new leaf nodes into a binary tree. Corresponding to insertions to a binary search tree with data abstracted away, the position for the new leaf is chosen by an arbitrary walk from the root to the leaves. We test for the undesirable patterns described by the following LBMP formulae:

- $t(\xrightarrow{l} + \xrightarrow{r})^*[p = \top]$. There is a node with undefined left, or right successor.
- $\exists x, y.\ t(\xrightarrow{l} + \xrightarrow{r})^* x \ \wedge\ x \xrightarrow{l} (\xrightarrow{l} + \xrightarrow{r})^*[p \neq \bot] y \ \wedge\ x \xrightarrow{r} (\xrightarrow{l} + \xrightarrow{r})^* y$. Two branches join in the tree.
- $\exists x.\ t(\xrightarrow{l} + \xrightarrow{r})^*[p = x](\xrightarrow{l} + \xrightarrow{r})^+[p = x]$. There is a cycle in the tree

**The Depth-first search.** The program implements the non-recursive version of the well-known algorithm for depth-first search. As input, we take a set of all possible binary trees with parent pointers, where all nodes are marked (have data value) as *non-visited*. The root of a tree is pointed by variable $root$. At the end of the procedure, we check that all nodes have been visited (marked as *visited*) – specified by the following LBMP formula:

- $root(\xrightarrow{l} + \xrightarrow{r})^*[p \neq \bot \wedge p.val = \text{“}nonvisited\text{”}]$

**Deutsch-Schorr-Waite (DSW).** DSW is a tree transversal algorithm close to the depth-first search. However, instead of a stack or parent pointers, it temporarily redirects the left/right next pointers to parent nodes to remember the return path. During the upward phase of the traversal, the next pointers are re-directed to their original destination. The initial set contains all possible binary trees. We check for null and undefined pointer exceptions.

**Other Structures**

**Linking leaves in trees.** In this case, we work with a data structure where each node has 4 next pointers: $l$—the left successor, $r$—the right successor, $par$—a pointer to the parent node, and $sc$—a pointer to the successor in the singly-linked list of leaves. The initial set contains all binary trees with parent pointers (with the restriction that both the left and right successors of a node are defined or null). The root is pointed by the variable $t$. In each node, $successor$ is set to null. Nodes in the tree are traversed in the depth-first order. At the beginning, we set $lastleaf == null$. Then, if we encounter a leaf during the DFS traversal, we do following steps: (1) If $lastleaf\ \ != null$, we link the previous leaf with the current one ($lastleaf{-}{>}successor = present$). (2) We set the variable $lastleaf$ to point to current leaf. We check that the leaves are correctly linked, i.e. there is a correct pointer $sc$ from a leaf to its successor. This is described by the following LBMP formulae:

- $\exists x, l, r.\ t(\xrightarrow{l} + \xrightarrow{r})^*[p \neq \bot]x \wedge x \xrightarrow{l} (\xrightarrow{r})^*[p \neq \bot]l \wedge l \xrightarrow{r} [p = \bot] \wedge x \xrightarrow{r} (\xrightarrow{l})^*[p \neq \bot]r \wedge r \xrightarrow{l} [p = \bot] \wedge l \xrightarrow{sc} [p \neq r]$. Two successor leaves in a tree are not linked in the list of leaves.

**Inserting into lists of lists.** In this case study, we consider a program that repeatedly inserts new elements into a list of lists. The data structure has two different types of nodes: (1) Nodes of the type *listhead* contain two next pointers: $nl$–a pointer to the next listhead, $n$–a pointer to the first *listelement*. (ii) Nodes of the type *listelement* are equal to nodes of SLLs with one next pointer: $n$—the next listelement. As the initial set, we use the set containing a single list with a single *listhead* element with null next pointers $nl$ and $n$. For each insertion, an existing second-level list is randomly chosen (or a new one is created), and then a new node is inserted at the end of this list. This means that we have to iteratively go through this list to its tail. We check absence of the following LBMP bad properties of the resulting data structure:

- $l \xrightarrow{nl}^* \xrightarrow{n}^* [p = \top]$. There is an undefined next pointer in the structure.
- $\exists x.\ l \xrightarrow{nl}^* first \wedge first \xrightarrow{nl}\xrightarrow{nl}^* second \wedge first \xrightarrow{n}^* [p = x] \wedge second \xrightarrow{n}^* [p = x]$. There is a sharing between two lists.

**Deletion in task-Lists.** This case study deals with one of the structures often used in operating systems [BCC$^+$07a]. The structure is showed in Figure 4.7. It consists from a singly linked list of threads (circle nodes), and doubly linked lists of tasks associated to a given thread (box nodes). All threads have a pointer to the next thread *nextthread*, and pointer to the first task *task*. The first thread is pointed by a variable *root*. There is a lock associated to each thread to ensure mutual exclusion in case of a manipulation with a list of tasks. Each task has a pointer to its predecessor *prev* and successor *next*, and a pointer to the thread *actthread* containing the given task.

We perform a delete operation on this structure. First, we randomly choose one task, and set a lock on the corresponding thread node. Then, we remove this task from the

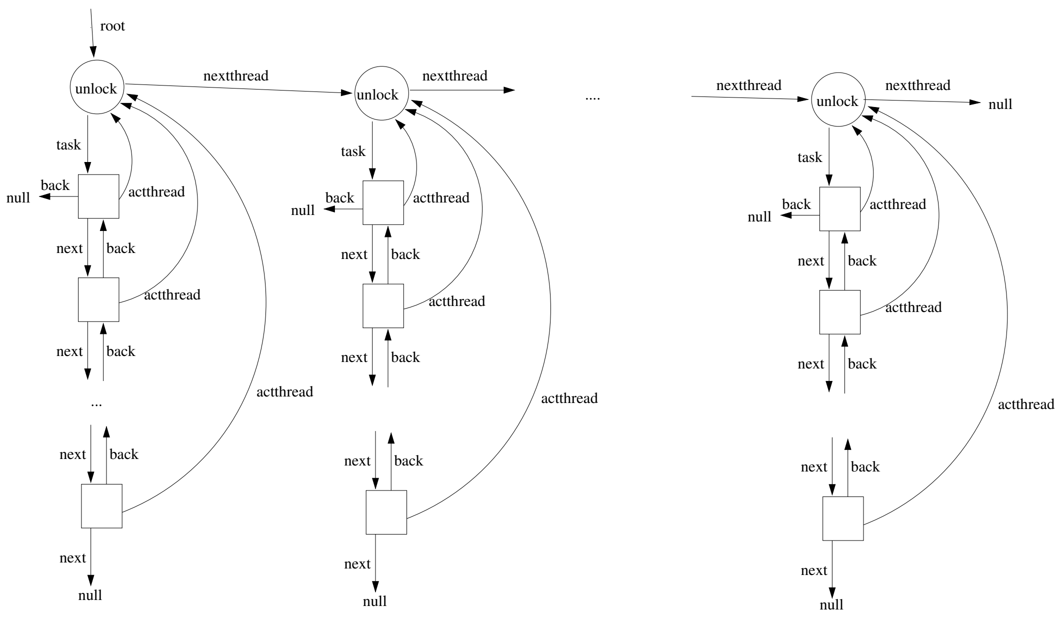


Figure 4.7: The ”task-list” data structure

structure—i.e. we correctly reconnect the *next*, *back*, and possibly *task* pointers. After this, we unlock the thread. At the resulting structure, we check that the following property does not hold:

- $\exists x.\ root \overset{nextthread^*}{\to} [p \neq \bot] f \wedge root \overset{nextthread^*}{\to} [p \neq \bot] s \wedge f \neq s \wedge f \overset{task}{\to}\overset{next^*}{\to} [p = x] \wedge s \overset{task}{\to}\overset{next^*}{\to} [p = x]$ This formula describes a sharing between two lists of tasks.

**Insertion into task-lists.** This case study works with the same data structure as the previous one. It chooses an arbitrary thread, locks it and adds a new task at the end of the list of tasks. Finally, it unlocks the thread. We check the following properties:

- $\exists x.\ root \overset{nextthread^*}{\to} [p \neq \bot] f \wedge root \overset{nextthread^*}{\to} [p \neq \bot] s \wedge f \neq s \wedge f \overset{task}{\to}\overset{next^*}{\to} [p = x] \wedge s \overset{task}{\to}\overset{next^*}{\to} [p = x]$ There is a sharing between two lists of tasks.

- $\exists x.\ root \overset{nextthread^*}{\to}\overset{task}{\to}\overset{next^*}{\to} [p = x] \overset{next}{\to}\overset{next^*}{\to} [p = x]$ There is a loop in the lists of tasks.

- $root \overset{nextthread^*}{\to} [p.val = locked]$ There is a locked thread.

- $\exists x.\ root \overset{nextthread^*}{\to} [p = x] \overset{task}{\to}\overset{next^*}{\to}\overset{actthread}{\to} [p \neq x]$ The pointers to the actual thread are not correctly set.

## 4.5.3 Experimental Results

Table 4.1 contains verification times for the experiments mentioned above. We give the best result obtained using the three mentioned abstraction schemas and say for which abstraction schema the result was obtained. The note "restricted" accompanying the abstraction method means that the abstraction was applied at the loop points only. The experiments were performed on a 64bit Opteron 2,8 GHz. The column $|Q|$ gives information about the size (a number of states) of the biggest encountered automaton, and $N_{ref}$ gives the number of refinements.

Despite the prototype nature of our tool, which can still be optimized in multiple ways (some of them are mentioned in the summary of this chapter), the results are quite competitive. For example, the Deutsch-Schorr-Waite tree traversal, TVLA (version 2) took 1 minutes on the same machine with manually provided instrumentation predicates and predicate transformers. In case of the tree with linked leaves, we are not aware of any other fully automated tool with which experiments with this structure have been performed. The results of experiments with *task-lists* are hard to compare. Structures of this type were studied only in the recent work [BCC$^+$07a] and no concrete examples (and verification times) are provided. The experiments with our prototype tool were done to show that ARTMC is capable to handle also this type of structures.

Table 4.1: Results of experimenting with the prototype implementation of the presented method

| Example | Time | Abstraction method | $\|Q\|$ | $N_{ref}$ |
|---|---|---|---|---|
| Creation of SLLs | 1s | predicates, restricted | 25 | 0 |
| Reversion of SLLs | 5s | predicates | 52 | 0 |
| Deletion from DLLs | 6s | finite height | 100 | 0 |
| Insertion into DLLs | 10s | neighbor, restricted | 106 | 0 |
| Reversion of DLLs | 7s | predicates | 54 | 0 |
| Insertsort of DLLs | 2s | predicates | 51 | 0 |
| Inserting into trees | 23s | predicates, restricted | 65 | 0 |
| Depth-first search | 11s | predicates | 67 | 1 |
| Linking leaves in trees | 40s | predicates | 75 | 2 |
| Inserting into a list of lists | 5s | predicates, restricted | 55 | 0 |
| Deutsch-Schorr-Waite tree traversal | 47s | predicates | 126 | 0 |
| Insertion into task-lists | 11m 25s | finite-height, restricted | 277 | 0 |
| Deletion in task-Lists | 1m 41s | predicates, restricted | 420 | 0 |

## 4.6 Implementation Details

This section describes some implementation details used in our prototype tool (cf. Section 4.5). First, we introduce the Mona [KM01] tree automata library, and then we describe the algorithms which are behind our tool.

### 4.6.1 Mona

Mona [KM01] is a tool which was designed for deciding validity of formulae of the WS1S and WS2S (Weak Second-order Theory of One or Two successors) logics. It is based on

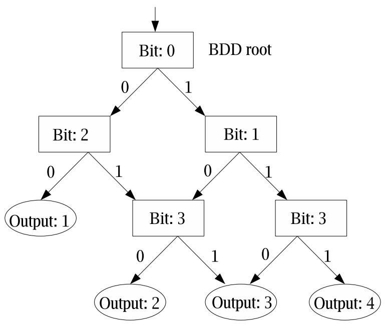


Figure 4.8: An example of a multiterminal binary decision diagram (MT-BDD)

the relation between the logics and the finite (tree) automata theory. For each WS1S formula, there exists a corresponding finite automaton that accepts all models of the formula. Similarly, for each WS2S formula there exists a corresponding finite tree automaton that accepts all models of the WS2S formula. Checking the validity of a formula then corresponds to checking non-emptiness of the corresponding automaton. The conjunction of formulae corresponds to the intersection of automata, the disjunction of formulae to the union of automata and the negation to the complementation of automata. Mona receives a formula, converts it into the corresponding automaton and checks the automaton for emptiness.

The implementation of Mona contains two interesting libraries. The first one for finite automata and the second one for binary tree automata (tree automata with branching factor two). Both are based on multiterminal binary decision diagrams (MT-BDD[3]) [Bry86] with a strong emphasis on efficiency. BDDs present an effective way how to compute functions on binary numbers. The term *multiterminal* means that the function can return more then two (true/false) output values. An example of a BDD can be seen in Figure 4.8. In this BDD, we compute the result according to the first 4 bits of the input number. Note that not all bits used in the input number must be used to distinguish the output value. Loops in BDDs are forbidden, but sharing between BDD paths is allowed. One can also define a set of functions by one huge BDD with a set of input nodes called *BDD-roots*. Each function is then defined by the corresponding *BDD-root*. The implementation of BDDs in Mona automatically merges nodes with an equal behaviour within a BDD. Therefore two equal functions are always defined by an equal BDD-root.

### Tree Automata in Mona

The Mona tree automata library allows one to manipulate *binary tree automata*. A binary tree automaton $A = (Q, \Sigma, F, \delta, init)$ is a tree automaton (cf. Section 3.1) where all rules are of the type $s(a, b) \rightarrow_\delta c$ where $s \in \Sigma$ and $a, b, c \in Q$. Instead of the *initial* rules (rules of the type 2, Section 3.1), an initial state is introduced. All initial rules are then of the type $s(init, init) \rightarrow_\delta a$ where $s \in \Sigma$, $a \in Q$, and $init \in Q$ is the initial state of the

[3]To simplify the text, we will use simply BDD to denote MT-BDD in the following.

automaton $A$. General tree automata accepting n-ary trees (with n-ary branching) can be encoded into binary ones where all n-ary rules are replaced by a set of binary ones.

Mona supports deterministic binary tree automata only—nondeterminism arises only internally during various automata manipulations and is immediately removed[4]. The implementation of binary deterministic tree automata in Mona is based on BDDs. The idea is to encode all symbols from alphabet $\Sigma$ into binary codes. Then, one can define for each pair of states on the left-hand sides of rules from $\delta$ a function $\Sigma \to Q$. This function is encoded as a BDD, and it describes rules of the type $s(a, b) \to c$ for fixed states $a, b \in Q$. There are $|Q|$ states in the automaton, so there are $|Q|.|Q|$ such functions. Some of the functions can be equal, and the implementation of the Mona BDD library guarantees that these equal functions will be represented by an equal BDD-root.

The encoding of rules into BDDs allows one to efficiently manipulate them during the operations on automata such as intersection, union, etc. The most complex operation is the projection of a selected bit, which makes the function independent of the value of this selected bit. This leads to a nondeterministic choice in nodes where the selected bit is used but the nondeterminism is immediately removed due to the fact that Mona automata are always deterministic.

### 4.6.2 Transduction

The Mona tree automata library does not provide transducers. However, it allows us to encode structure preserving transducers as automata as we describe below. The idea is the following. Regular automata use an alphabet $\Sigma$. Structural-preserving transducers may be viewed as automata on $\Sigma \times \Sigma$. In order to have the same alphabet in both cases we lift regular automata to $\Sigma \times \Sigma$ by putting the original symbols to the left and combining them with all possible symbols on the right. In Mona, this means that we use twice as many bits. We encode the symbols of the alphabet $\Sigma \times \Sigma$ as follows: Even bits (including the bit 0) are used for the left-hand side part of a symbol and odd bits for the right-hand side[5]. The application of a transducer on an automaton is then done in the following way:

1. intersection between the automaton and the transducer,
2. projection of all even bits (the even bits are always set to a nondeterministic choice and hence will not restrict the input any more),
3. remapping of odd bits to even bits (i.e. swap $(a, b) \in \Sigma \times \Sigma$ to $(b, a) \in \Sigma \times \Sigma$)[6].

The result is an automaton where each accepted tree can be obtained (using the transducer) from some tree accepted by the original automaton. The reverse application of a transducer is done by the same three steps, but in a different order—(1) remapping (even bits to odd), (2) intersection with the transducer, (3) projection on odd bits.

[4]All bottom-up tree automata are deteminizable.

[5]The interleaving of the left-hand side symbol with the right-hand side symbol in the binary encoding is used for technical reason. Our transducers often use copying rules—i.e. symbols $(a, a) \in \Sigma \times \Sigma$, and the interleaving produces much smaller BDDs.

[6]All even bits are undefined due to the previous step

### 4.6.3 Encoding Memory Configuration in Binary

Our encoding of shape graphs into binary trees was described in Sections 4.4.1 and 4.4.2. The encoding is based on a set of pointer variables $\mathcal{V}$, a set of pointer descriptors $\mathcal{R}$, a set of markers $\mathcal{M}$, and a set of data values $\mathcal{D}$. As was already described, we have the following types of nodes placed on specific positions in a valid tree memory configuration:

1. The node describing undefined pointer variables is the first type. This node is labeled by a symbol from the alphabet $\Sigma_1 = 2^{\mathcal{V}}$, and it is placed in the root of the tree.

2. The node describing pointer variables set to null is the second type. This node is also labeled by a symbol from the alphabet $\Sigma_1 = 2^{\mathcal{V}}$, and it is the left son of the node of Type 1.

3. Nodes describing allocated and disposed memory cells are the next type. These nodes are labeled by symbols from the alphabet $2^{\mathcal{V}} \times 2^{\mathcal{M}} \times \mathcal{D}$. They are left sons of the nodes of Type 2, or left sons of the nodes of Type 4 presented below.

4. Nodes describing the direction of next pointers represent Type 4 of the nodes we are dealing with. A set of these nodes corresponds to each node of Type 3 (see Figure 4.5). Each node of this type is labeled by a symbol from the set $\mathcal{R} \cup \{\perp, \top\}$, and it is the left son of a node of Type 3, or a right son of another node of Type 4. The number of pointer nodes between the current one and the corresponding node of Type 3 is called the *level* of the node.

5. Nodes labeled by the bottom mark $\bullet$ represent the last class. These nodes are right sons of nodes of Types 1, 2, and 3, and left and right sons of nodes of Type 4.

To connect each tree memory encoding with a control location in the program being verified, we extend the tree memory encoding by a new root node labeled by the identifier of the corresponding location. We denote the set of location identifiers as $\mathcal{L}oc$.

In order to use the Mona tree automata library, we must encode the alphabet into binary codes. In these binary codes, we use the first two bits to identify the type of the node. Nodes of Type 1 and Type 2 are coded as *00UXXXXX*. *00* defines the type of the node, *U* is an unused bit (for technical reason[7]), and *XXXXX* defines a symbol from $2^{\mathcal{V}}$—the pointer variables which are undefined or null. The length of this binary code depends on $|\mathcal{V}|$—each position *X* corresponds to one pointer variable: If its value is 1, the pointer is set, otherwise the pointer variable is unset. E.g., a node describing undefined pointer variables labeled by the binary code *0001010* means that the first and third pointer variables are undefined. There is no reason to distinguish the binary codes of nodes describing null and undefined pointer variables because these nodes have unique positions in the tree.

Nodes of Type 3 are coded as *1WUXXXXXTMMMMMDDDDD*. The first two bits set to *11* define the top-most memory nodes in the tree, and the code *10* is used for all other

[7]We need to have the *XXXXX* part defined on the same position as in nodes of Type 3.

such nodes. The bit *U* set to 1 identifies a zombie of a disposed node. An allocated memory cell has the bit *U* set to 0. The code *XXXXX* defines the pointer variables aliased to the given node—the code represents a subset of $\mathcal{V}$ is the same way as in the case of nodes of Type 1 and 2. The bit *T* is a token ⧫ used during an application of routing expressions (see Sections 4.4.2, and 4.6.7) and it is set to 0 in a valid tree memory configuration (i.e. a configuration which is not undergoing a transformation due to a statement of the form $x = y\text{->}next$). The part *MMMMM* defines a symbol from $2^{\mathcal{M}}$—i.e. markers set in the node. 1 at a position $i$ means that the marker $i$ is set, 0 means it is not set. And finally, the code *DDDDD* defines a data value from the set $\mathcal{D}$. E.g., for 3 pointer variables, 3 markers and data of the length 2, the code *100010001110* defines an allocated memory cell pointed by the second pointer variable, marked by the second and the third marker, and carrying a data value 10.

The nodes describing directions of next pointers are encoded as follows. The first two bits are always set to *01*. If the next pointer is undefined ($\top$), the code *01011* is used. The code *01010* is used for a null pointer ($\perp$). The code *011XXXXX* identifies a pointer descriptor (c.f. Section 4.4.1)—a reference to the corresponding routing expression. There is one code for each symbol from the set $\mathcal{R}$. The length of the part *XXXXX* corresponds to $|\mathcal{R}|$ (i.e the number of pointer descriptors), and only one bit in this part can be set to 1[8]. E.g., the code *0110010* identifies the next pointer with the pointer descriptor number 3.

The bottom mark $\bullet$ is encoded as *00*. There is no conflict with the nodes of Type 1 and Type 2 using the same prefix in its binary code because these nodes have a unique position in a valid tree memory encoding.

The program location at which a given tree memory configuration arise is encoded in binary with no special prefix—the location node has a unique position in the tree (the root node), so no special code is necessary to identify it.

### 4.6.4 Correctness of Tree Memory Encoding After an Abstraction

In Sections 3.3.2 and 3.3.3, we have proposed two generic abstraction methods to accelerate the fixpoint computation of ARTMC. If we use them without any restriction, the methods may produce a corrupted tree memory encoding. In order to ensure that the result of the abstraction is a correct tree memory encoding, we do not allow collapsing of nodes of the different types discussed in Section 4.6.3. Moreover, we need to maintain a unique position of all pointer variables. Therefore we apply a special *initial partitioning* of automata states, and we restrict the result of the abstraction by means of specially created tree automata.

#### Initial Partitioning of Automata States

We use a special *initial partitioning* of automata states to guarantee that only states accepting symbols of the same type (see Section 4.6.3) can be collapsed. Within the initial partitioning, we simply examine rules of the input automaton and we partition its states

[8]One can also compress the encoding by using of binary coded numbers to identify the pointer descriptors. We use the simpler unary encoding, because it is simpler to parse.

into classes according to the types of nodes accepted by these states—i.e. according to the first two bits of the binary encoding of these nodes and according to their position in the tree. There are no states accepting two types of nodes in an automaton accepting a set of correctly encoded tree memory configurations. This is due to the fact that the ordering of the types of nodes in a correct tree memory encoding is strictly given (cf. Section 4.6.3). E.g., the node of Type 3 is always a left son of a node of Type 2, or 4. Hence, if there exists one state of automaton accepting nodes of Types 3 and 4, then we will be able to create a tree accepted by this automaton which is not a correct tree memory encoding—the node of the Type 3 (accepted by this state) can be also connected as a left son of a node of Type 3 or as a right son of a node of Type 4. The result of the initial partitioning is taken as the starting point of the abstraction used (in the case of the finite height abstraction) or it is combined with the result of the abstraction (in the case of the predicate-based abstraction).

The initial partitioning can be improved by one of the following two possibilities. (1) The states accepting next pointer nodes (nodes of Type 4, Section 4.6.3) are separated into several classes according to the *level* (see Section 4.6.3, and Figure 4.5) of the next pointer node—then, next pointer nodes of different levels will not be collapsed, and the shape of a tree memory encoding will be preserved. (2) Each state accepting a next pointer node is put into a separate class. This second possibility completely excludes collapsing of states accepting next pointer nodes. Note that this step will not hamper the finiteness of the abstraction because we still collaps the memory nodes above the next pointer nodes. Our experimental results showed that this variant of the initial partitioning usually leads to the best results.

### Restricting the Result of an Abstraction by Specially Created Automata

The initial partitioning guarantees that the types of nodes in memory configurations are not mixed after the abstraction. But this is not enough to obtain a valid tree memory encoding because the position of each pointer variable must be unique. The abstraction can collaps a node bellow the node defining the position of a pointer variable with a node above it. This creates a loop and causes that a tree without a defined position of the pointer variable[9] as well as a tree with multiple defined positions of the pointer variable appear in the abstracted automaton. Therefore we automatically create a set of restricting automata (one for each pointer variable) defining all possible tree encodings with a unique position of the given pointer variable. To ensure the correctness of the tree memory encoding, we can (1) make an intersection of the result of abstraction with all of these restricting automata, or (2) use the restricting automata as initial predicates of the predicate-based abstraction used.

[9]This is different from the case of an undefined pointer where the pointer identifier must be placed in the designated node in the top of the tree, Section 4.6.3

### 4.6.5 Implementation of the Finite-Height Abstraction

In Section 3.3.2, we proposed the abstraction method based on *languages of trees of a finite height*. To recap, this abstraction method defines two states equivalent if their languages up to a given height $n$ are equivalent. A refinement is done by an increase of the height $n$.

Our implementation of the finite-height abstraction is done over binary tree automata (cf. Section 4.6.1)—a subclass of general tree automata (c.f. Section 4.6.1). Let $A = (Q, \Sigma, F, \delta, init)$ be a binary tree automaton. We define the function $class : Q \to \mathbb{N}$ as a mapping from automata states to equivalence classes (each class is represented as a natural number). The initial mapping is taken from the initial partitioning of states of the automaton $A$ (c.f. Section 4.6.4). Then, we iteratively make the mapping *class* more precise as described bellow. The number of the iterations depends on the given height $n$ used as the parameter of the abstraction.

In each iteration step, we create a new (more precise) mapping $\overline{class}$ in order to replace the old one at the end of the step. The mapping $\overline{class}$ will make two states $p$ and $q$ equivalent if (1) $p$ and $q$ are equivalent according to the old mapping $class$, (2) for each rule $s(a_1, a_2) \to p \in \delta$, we can find a corresponding rule $s(b_1, b_2) \to q \in \delta$ where the states $a_1$ and $b_1$, and the states $a_2$ and $b_2$ are equivalent according to the old mapping $class$, and (3) for each rule $s_1(c_1, c_2) \to q \in \delta$, there must also exist a corresponding rule $s_1(d_1, d_2) \to p \in \delta$ where $c_1$ is equivalent to $d_1$ and $c_2$ is equivalent to $d_2$ according to the old mapping $class$. Formally, $\overline{class}(p) = \overline{class}(q) \overset{def}{\Longleftrightarrow} class(p) = class(q)$ and $\forall s(a_1, a_2) \to p \in \delta$
$\exists s(b_1, b_2) \to q \in \delta . class(a_1) = class(b_1) \ \wedge \ class(a_2) = class(b_2)$ and $\forall s_1(c_1, c_2) \to q \in \delta \ \exists s_1(d_1, d_2) \to p \in \delta . class(c_1) = class(d_1) \ \wedge \ class(c_2) = class(d_2)$.

In order to create the new mapping $\overline{class}$ efficiently, we use the following principle. For each state $q \in Q$ of the automaton $A$, we compute a matrix $M_q$ of the size $Cl \times Cl$ where $Cl$ is the number of classes defined by the current mapping $class$. Each cell of this matrix then contains a subset of symbols of the alphabet $\Sigma$. The semantics of the matrix is the following: A presence of a symbol $s \in \Sigma$ in a cell $[i, j]$ of the matrix $M_q$ is equivalent to the existence of a rule $s(a, b) \to q \in \delta$ where $class(a) = i$ and $class(b) = j$. Using these matrices, we can easily check that for a rule $s(a, b) \to p \in \delta$, there exist a corresponding rule with $q$ on the right-hand side by looking into the matrix of $q$ to the cell $[class(a), class(b)]$—the presence of the symbol $s$ in this cell implies the existence of such a rule. Moreover, we can put $p$ and $q$ into the same classes iff the matrices of $p$ and $q$ are equal without checking the equality rule by rule. This is due to the fact that all rules are recorded in these matrices. The whole algorithm for computing the finite-height abstraction using the described matrices is presented in Figure 4.9. In order to easily identify states with equal associated matrices, we first sort the states according to their matrices by using a straightforward ordering on the matrices comparing their corresponding elements from smaller to bigger indices.

To represent the subsets of $\Sigma$ placed in cells of the matrices, we use BDDs. As was already described (cf. Section 4.6.1), a BDD is an efficient representation of a function from binary numbers to some finite set. As the symbols from an alphabet $\Sigma$ are coded as binary numbers in our implementation (cf. Section 4.6.3), we can use a BDD encoding a function $\Sigma \rightarrow \{true, false\}$ to describe which symbols are in the corresponding subset. In cells of the matrices, we then simply record links to appropriate BDD-roots. Moreover, the Mona implementation of BDDs automatically merges parts of BDDs with equal behaviour. Therefore, equal sets will be represented as an equal reference to some BDD-root. The references to BDD-roots are represented as integer values, and so in the end, we are comparing matrices of integers when creating the new mapping $\overline{class}$.

```
type $set\_of\_symbols$ = values of this type are subsets of symbols of $\Sigma$
$class$=Initial partitioning of states of the input automaton $A$
for $i = 1$ to $abstraction\_height$ {
    $Cl$ = number of classes defined by the mapping $class$
    type $act\_matrix$ = matrix of the size $Cl \times Cl$ of the type $set\_of\_symbols$
    $mtx$ = array of size $|Q|$ of the type $act\_matrix$
    set all cells of all matrices in $mtx$ to $\emptyset$
    forall $s(a, b) \rightarrow c \in \delta$ {
        add symbol $s$ into the cell [$class(a)$,$class(b)$] of the matrix $mtx[c]$
    }
    create the mapping $\overline{class}$ : $\overline{class}(p) = \overline{class}(q) \Leftrightarrow mtx[p] = mtx[q]$
    $class = \overline{class}$
}
Collaps states of the automaton $A$ according to the classes defined by the mapping $class$
```

Figure 4.9: An efficient implementation of the finite-height abstraction

### 4.6.6 Implementation of Predicate-Based Abstraction

In Section 3.3.3, we propose the *abstraction based on predicate tree languages*. To recall the basic principle, let $\mathcal{P} = \{P_1, P_2, \ldots, P_n\}$ be a set of *predicates* where each predicate $P \in \mathcal{P}$ is a tree language represented by a tree automaton. Let $A = (Q, \Sigma, F, \delta)$ be a tree automaton. Then, two states $q_1, q_2 \in Q$ are equivalent iff their languages $L(A, q_1)$ and $L(A, q_2)$ have a nonempty intersection with exactly the same subset of predicates from the set $\mathcal{P}$.

In our implementation, one automaton defines a set of predicates—each state can be chosen as a final state of the predicate automaton[10]. If there are used several automata describing predicates (each one of them representing a set of predicates), then our implementation of the predicate-based abstraction will treat each of them separately, and the results will be combined. As in the case of the finite-height abstraction, the implementation is done over binary tree automata (c.f. Section 4.6.1). Let $M = (Q, \Sigma, F, \delta, init_m)$ be a binary tree automaton on which the abstraction is applied, and $P = (Q_p, \Sigma, F_p, \delta_p, init_p)$ be an automaton describing predicates. The following algorithm attaches to each state of the automaton $M$ a subset of states of the automaton $P$. We denote the set attached to

[10]Note that in the refining process (cf. Section 3.3.3), languages of all states of a given automaton are added as new predicates.

the state $q \in Q$ as $S(q)$. The presence of a state $q_p \in Q_p$ in the $S(q)$ means that the state $q$ satisfies the predicate described by the automaton $P$ with the finite state $q_p$, i.e. $L(M,q) \cap L(P,q_p) \neq \emptyset$.

The algorithm starts with empty sets $S(q)$ for all states from $M$ except the set attached to the initial state $S(init_m)$ containing the state $init_p$. Then it searches for rules $s(a_1,a_2) \to a_3 \in \delta$ and $s(b_1,b_2) \to b_3 \in \delta_p$ where $s \in \Sigma$, $b_1 \in S(a_1)$, $b_2 \in S(a_2)$, and $b_3$ is not in the set $S(a_3)$. We call this pair of rules a *corresponding pair of rules*. An existence of the corresponding pair of rules means that $a_3$ satisfies the predicate described by $b_3$, and this fact is not yet recorded. If we find such a pair of rules, then we will add $b_3$ into the set corresponding to $a_3$. There is a finite number of states in $Q$ and $Q_p$, and therefore we can do at most $|Q| \times |Q_p|$ of such steps.

The implementation is described in Figure 4.10. We suppose that states of $Q$ and $Q_p$ are numbered from 0 to $|Q|-1$, and from 0 to $|Q_p|-1$ respectively. To represent the sets attached to the states of the automaton $M$, we use a matrix *block* of the size $|Q| \times |Q_p|$ where each cell carries a value 0 or 1. The presence of a state $q_p \in Q_p$ in the set attached to $q \in Q$ is represented by the value 1 in the cell *block*$[q,q_p]$, and its absence by the value 0. To increase the efficiency of searching for the rules and to cut down the number of repeated examination of the same rule, we introduce a stack of pairs $(q_1,q_2)$ where $q_1 \in Q$ and $q_2 \in Q_p$. In the beginning, we push the pair of initial states $(init_m, init_p)$ into this stack. Then, in each computation step, we pop a pair $(q_1,q_2)$ from the stack and we examine ($\forall s \in \Sigma$, $\forall q_a, q_b \in Q$, and $\forall q_c, q_d \in Q_p$) the rules from $\delta$ of the form $s(q_1,q_a) \to q_b$ and $s(q_a,q_1) \to q_b$ and rules from $\delta_p$ of the form $s(q_2,q_c) \to q_d$ and $s(q_c,q_2) \to q_d$. If we find a corresponding pair of rules $s(q_e,q_f) \to r_1 \in \delta$ and $s(q_g,q_h) \to r_2 \in \delta_p$ between the examined rules, then we add the state $r_2$ into the set $S(r_1)$ and we push the pair $(r_1,r_2)$ into the stack.

The use of BDDs to represent sets of automata rules (see Section 4.6.1) allows us to examine rules of the type $s(q_1,q_2) \to q_3$ by one search through the corresponding BDD for all symbols $s \in \Sigma$. This allows us to get rid of the loop *forall* $s \in \Sigma$ in the algorithm described in Figure 4.9.

## 4.6.7 Implementation of Routing Expressions

As was described in Section 4.4.2, we encode routing expressions as transducers moving the token ⧫ (represented as one selected bit in the binary representation, see Section 4.6.3) from a source node describing an allocated memory cell to a destination one. The destination node must contain the marker paired with the routing expression. An execution of the statement $x.next = y$ can introduce new links in the tree memory encoding which are not covered by the current routing expressions. Therefore it is necessary to update the routing expression associated with the executed $x.next = y$ statement (recall that we have a routing expression for each such statement). This routing expression must cover all newly added combinations of source and destination nodes. The source nodes contain the pointer variable $x$ and the destination ones the variable $y$.

As we describe below, the new routing expression can be created automatically. The creation is an expensive process due to the projection operations used. However, in a lot of

```
*block*=matrix of size $|Q| \times |Q_p|$ over the type $\{0, 1\}$
set all cells of *block* to the value 0
*stack*.init($int \times int$)  //create an empty stack of pairs of integers
*block*$[init_m, init_p] = 1$
*stack*.push($init_m, init_p$)
while (not *stack.empty*) {
    (a,b)=*stack*.pop
    for $i = 0$ to $|Q| - 1$ {
        for $j = 0$ to $|Q_p| - 1$ {
            forall $s \in \Sigma$ {
                if $((\exists x \in Q.s(a, i) \to x \in \delta) \wedge (\exists y \in Q_p.s(b, j) \to y \in \delta_p) \wedge$ *block*$[i, j] = 1$
                        $\wedge$ *block*$[x, y] = 0$) {
                            *block*$[x, y] = 1$
                            *stack*.push$(x, y)$
                }
                if $((\exists x \in Q.s(i, a) \to x \in \delta) \wedge (\exists y \in Q_p.s(j, b) \to y \in \delta_p) \wedge$ *block*$[i, j] = 1$
                        $\wedge$ *block*$[x, y] = 0$) {
                            *block*$[x, y] = 1$
                            *stack*.push$(x, y)$
                }
            }
        }
    }
In the automaton $M$, collaps all states $i, j \in Q$ where *block*$[i]$ ==*block*$[j]$.
}
```

Figure 4.10: An implementation of the predicate-based abstraction

cases, it is not necessary to run it because the routing expression already associated with the given $x.next = y$ statement is sufficient. Therefore, we introduce a coverage checking procedure to decide whether it is really necessary to extract a new routing expression.

### Automatic Creation of Routing Expressions

To obtain the new routing expression, we need to extract paths between the nodes where the two pointer variables $x$ and $y$ are placed in all so-far computed tree memory encodings. For this extraction, we use a special transducer. This transducer is defined on the alphabet $\Sigma \times \{00, 01, 10, 11\}$ and has the following rules:

- A node without $x$ and $y$ is transduced to $00$.
- A node containing $x$ is transduced to $10$.
- A node containing $y$ is transduced to $01$.
- A node containing both $x$ and $y$ is transduced to $11$.

Note that there is a unique position of $x$ and $y$ in each tree of a correct tree memory encoding. Therefore, after an application of the above transducer, we obtain an automaton describing trees where the position of the variables $x$ and $y$ is still recorded, but the other variables, data, markers, etc. are abstracted away. We call this automaton a *shape automaton*. In the trees, we still also remember their exact original number of nodes and their interconnections. However, in order to create a new routing expression, we need to extract just the mutual position of $x$ and $y$ in trees—the exact position (linked to the root of the tree) of the variables is undesirable. Therefore we subsequently modify the shape automaton in the following way: Everything below and above the shortest path between $x$ and $y$ is substituted by $00^*$ (an unbounded number of reading of the symbol $00$). This cannot be done by a structure preserving transduction, and so a special procedure is used. This procedure goes through the automaton and searches for states below and over the shortest path. All states over the shortest path are collapsed into a single state $q_{over}$, and the states below the shortest path are collapsed into a single state $q_{below}$. After application of this procedure, we obtain a *modified shape automaton* where $x$ can be placed almost everywhere (not all positions of $x$ allow to also accordingly place $y$) in the tree, and $y$ is placed at the position determined by the extracted shortest path[11].

Now, we can create a routing expression represented by a transducer. First, we lift the modified shape automaton to the alphabet $\{00, 01, 10, 11\} \times (\Sigma \times \Sigma)$ by a simple combination of symbols from the original alphabet with all possible symbols from the alphabet $\Sigma \times \Sigma$. Then, we define the following 1-state *template transducer* over the alphabet $\{00, 01, 10, 11\} \times (\Sigma \times \Sigma)$:

- $00$ is translated to a copying rule,
- $10$ is translated to a rule checking for the presence of the token ⧫ and deleting it,
- $01$ is translated to a rule checking that the token ⧫ is not set and setting it,
- $11$ is translated to a copying rule that additionally checks for the token ⧫ which corresponds to the case when the next pointer is a self loop,

After the intersection of the modified shape automaton and the template transducer, we project out the first two bits to get an ordinary transducer represented as an automaton over the alphabet $\Sigma \times \Sigma$ (see Section 4.6.2).

### Checking Coverage of Routing Expressions

Coverage checking is a simple procedure designed to check whether we can skip the creation of a new routing expression. At the beginning we have a tree memory encoding over which we need to perform the $x.next = y$ statement. In order to be able to avoid the creation (or extension) of the routing expression associated with the $x.next = y$ statement, we need to check that the shortest path between nodes pointed by variables $x$ and $y$ in each tree from the tree memory encoding is covered by the routing expression already associated to the executed statement. The coverage checking procedure works as follows:

[11]Note, that we work with a set of tree memory encodings, so we can, in general, extract a set of different shortest paths. $y$ is then placed on a position determined by nondeterministically chosen one of these paths. The nondeterminism is restricted using destination markers (cf. Section 4.4.1).

1. It sets the token ⧫ to the node containing the variable $x$.
2. It deletes markers paired with the tested routing expression from nodes without the variable $y$
3. The transducer describing the tested routing expression is applied on the result of Step 2.
4. A reverse application of the transducer describing the tested routing expression is performed on the result from Step 3.
5. Finally, a check whether the original set obtained by Step 2 is included in the result of Step 4 is performed.

If there is some tree with the shortest path between $x$ and $y$ not covered by the routing expression, then this tree will be lost after Step 3, and the loss will be detected at Step 5 (there will be no inclusion). Therefore the inclusion of the automaton obtained by Step 2 in the automaton obtained by Step 4 guarantee the coverage, and the creation of a new routing expression can be skipped.

## 4.7 Garbage Checking

By *garbage*, we understand allocated memory cells that were not freed and that are no more (transitively) accessible from any pointer variable. In languages like, e.g., Java or C#, such cells are automatically garbage-collected. In C or C++, however, they cause memory leakage. In our prototype implementation, we have not yet considered checking for garbage. It is, nevertheless, possible to do so, and we plan to experiment with it as a part of our future work.

To *check for garbage*, we may introduce a special mark saying that a certain memory node is reachable from the program variables. Then, we may mark all nodes directly pointed to by some program variable, by this mark. Subsequently, we may use a special transducer that if a node is marked, will mark all its successors. We may use ARTMC to iterate this transducer till a fixpoint. Out of the resulting tree memory encodings, we take those in which at least one node remains unmarked (which can be done simply by intersection with a suitable tree automaton). The fixpoint, however, records even the history of the marking process, and so some of the unmarked nodes may be unmarked just because they belong to a tree from the past where the marking process has not finished yet. Nevertheless, we can make one more marking step on the selected trees and take those trees that stay the same. If there is at least one such tree, we know that some garbage may arise in our program.

## 4.8 Summary of ARTMC for Dynamic Data Structures

In this chapter, we presented a fully automated method for verification of programs manipulating complex dynamic linked data structures. The method is based on the framework of ARTMC. In order to be able to use ARTMC, we proposed a new representation of sets of shape graphs based on tree automata and a representation of the standard C pointer manipulating statements as tree transducers (with some extensions). In particular, we considered verification of the basic memory consistency properties (no null pointer assignments, etc.) and of shape invariants whose corruption may be described in an existential fragment of a first-order logic on graphs. We formalized this fragment as a special-purpose logic called LBMP whose formulae may be translated to C-based testers that may be attached to the verified programs, thus transforming the verification problem to be considered to the control line reachability. We have implemented the technique in a prototype tool and obtained some quite promising experimental results. The method belongs to the most general and at the same time fully automated methods. Our experimental results showed that the method is capable to handle real pointer manipulating procedures. Moreover, using the method, we were able to handle procedures working on structures which were not automatically verified by any other method.

There is still a lot of things to be done in this area. The performance of our Mona-based prototype tool can be increased in several ways, e.g., by exploiting the concept of *guided tree automata* that are suggested as very helpful in many situations by the authors of Mona [BKR97] and that we have not used yet. Further, it is interesting to try to come up with some *special purpose automata abstractions* for the considered domain—so-far we have used mostly general purpose tree automata abstractions, and there is an experience from [BHMV05] that special purpose abstraction may bring very significant speed-ups (in [BHMV05], it was sometimes two orders of magnitude or even more). Another next possibility is to improve the automata machinery by using nondeterministic automata, or BDDs shared between several automata. We would also like to do experiments with other encodings of memory. For example, in Chapter 5, we use a very simple encoding designed just for binary trees in order to build a termination proof on the top of invariants generated by ARTMC. Other similar encodings could also be probably proposed for various more specialized classes of programs yielding more efficient analyzes for them.

# Chapter 5

# Proving Termination of Tree Manipulating Programs

To prove complete correctness of a program, one needs both safety and termination. Therefore verification of termination (and possibly other liveness properties) is an important problem in computer science. Errors in termination may be even worse to debug. The reason is that if a program crashes, an error report may at least be generated and analysed. In case of a termination bug, no such error report arise. Errors in termination may at the same time arise as a consequence of handling dynamic data structures (see e.g. [DBCO06]). In case of dynamic data structures, the verification problem is complicated by unboundedness of the data structures and tricky pointer manipulations.

In this chapter, we tackle the termination problem of programs manipulating tree data structures. Namely, we are interested in proving that such a program terminates for any input tree out of a given set described as an infinite regular tree language over a finite alphabet. The results presented in this chapter are based on [HIRV07a, HIRV07b].

We handle sequential, non-recursive programs working on trees with parent pointers and data values from a finite domain. The basic statements we consider are data assignments, non-destructive pointer assignments, and tree rotations. This is sufficient for verifying termination of many practical programs over tree-shaped data structures (e.g., AVL trees or red-black trees) used, in general, for storage and a fast retrieval of data. Moreover, many programs working on singly- and doubly-linked lists fit into our framework as well. In most of the chapter, we do not consider dynamic allocation, but insertion/removal of leaf nodes, common in many practical tree manipulating programs, can be easily added as is discussed in the last section of the chapter.

We represent a given program as a control flow graph whose nodes are annotated with (overapproximations of) sets of reachable configurations computed using abstract regular tree model checking (ARTMC, cf. Chapter 3). From the annotated control flow graph, we build a counter automaton (CA) that simulates the program. The counters of the CA keep track of different measures within the working tree: the distances from the root to nodes pointed to by certain variables, the sizes of the subtrees below such nodes, and the numbers of nodes with a certain data value. Termination of the CA is analyzed by existing tools, e.g., [BnHS05, CPR05, Ryb].

Our analysis uses an *Counter-example Guided Abstraction Refinement* (CEGAR) loop [CGJ$^+$00]. If the tool we use to prove termination of the CA succeeds, this implies that the program terminates on any input from the given set. Otherwise, the CA checker tool outputs a lasso-shaped counterexample. For the class of CA generated by our translation scheme, we prove that it is decidable whether there exists a non-terminating run of the CA over a given lasso[1].

However, even if we are given a real lasso in the generated CA, due to the abstraction involved in its construction, we still do not know whether this implies also non-termination of the program. We then map the lasso over the generated CA back into a lasso in the control of the program, and distinguish two cases. If (1) the program lasso does not contain tree rotations, termination of all computations along this path is decidable. Otherwise, (2) if the lasso contains tree rotations, we can decide termination under the additional assumption that there exists a CA (not necessarily known to us) that witnesses termination of the program (i.e., intuitively, in the case when the tree measures we use are strong enough to show termination). In both cases, if the program lasso is found to be spurious, we refine the abstraction and generate a new CA from which an entire family of counterexamples (including this particular one) is excluded.

The analysis loop is not guaranteed to terminate even if the given program terminates due to the fact that our problem is, in fact, not recursively enumerable. However, experience with our implementation of a prototype tool shows that the method is successfully applicable to proving termination of various real-life programs.

**Plan of the Chapter.** In Section 5.1, we discuss the existing approaches to verification of termination properties of programs with dynamic data structures. In Section 5.2, we introduce some basic notions about trees, relations, and counter automata. In Section 5.3, we describe the syntax and semantics of the programs we work with (including a description of a simple pre-processing phase that we apply over the input programs for some technical reasons related to our technique). In Section 5.4, we introduce our CEGAR loop. In Section 5.5, we describe our encoding of program configurations and statements in order to apply ARTMC to generate line invariants. Next, in Section 5.6, we describe the way we translate the considered programs into counter automata. In Section 5.7, we discuss how spuriousness of lasso-shaped counterexamples can be checked over the generated counter automata. Subsequently, in Section 5.8, we explain how an abstraction that yields a spurious lasso-shaped counterexample may be refined. In Section 5.9, we discuss a prototype implementation of our technique and the experimental results that we obtained from it. In section 5.10, we discuss the possibility of adding insert and delete statements into our framework. Finally, we summarize the chapter in Section 5.11.

[1]If the analyzer used returns a spurious lasso-shaped counterexample for the termination of the CA, we suggest choosing another tool.

## 5.1 Existing Approaches

As we have already seen in Chapter 4, the area of research on automated verification of programs manipulating dynamic linked data structures is recently quite live. Various approaches to verification of such programs differing in their principles, degree of automation, generality, and scalability have been proposed. In section 4.1, we briefly mentioned the approaches based on, e.g. monadic second order logic [MS01], 3-valued predicate logic with transitive closure [SRW02], separation logic [Rey02, DOY06], other logics [BIL03, BPZ05, MYRS05], automata [BHMV05, BFL06, DEG06], or other symbolic representations such as [YKB02, BCE$^+$05, LYY05, ČEV06]. Moreover, in Chapter 4, we in detail described our own method based on ARTMC (cf. Chapter 3).

With few exceptions, all existing methods tackle verification of safety properties. In [YRSW03], specialized ranking functions over the number of nodes reachable from pointer variables were used to verify termination of programs manipulating linked lists. Termination of programs manipulating lists has further been considered in [DBCO06, BBH$^+$06] using constraints on the number of nodes in particular list segments not having internal nodes pointed from outside. To the best of our knowledge, automated checking of termination of programs manipulating trees has so-far been considered in [LRS06] only, where the Deutsch-Schorr-Waite tree traversal algorithm was proved to terminate using a manually created progress monitor, encoded in first-order logic. We now describe these approaches in more details.

### Evolution Logic

In [YRSW03], the authors propose a special *evolution temporal logic*. It is a first-order linear temporal logic designed to specify safety as well as liveness temporal properties of programs with heap manipulations and thread creation. The authors showed, how the approach can be used to verify termination of programs manipulating linked lists. The termination proofs are based on specialized ranking functions. The approach was implemented in TVLA [SRW02] and tested on a series of examples manipulating linked lists such as searching in lists, insertion into lists, deletion from lists, merging of lists, list rotation, etc.

### Simulation by Integer Programs

Concerning termination of programs with recursive data structures, the available termination checkers for integer programs (e.g. [CPR06, Ryb]) can be used *provided* that there is a suitable abstraction of such programs into programs over integers, i.e., counter automata. Such abstraction can be obtained by recording some numerical characteristics of the heap in the counters, while keeping the qualitative properties of the heap in the control of the CA. Indeed, this is the approach taken in [BBH$^+$06] for checking termination of programs over singly-linked lists. The approach [BBH$^+$06] allows one to check both safety and termination, and it is in more details described in Section 4.1. Just to recall, the abstraction used in [BBH$^+$06] is based on compacting each list segment into a single abstract node and recording its length in the counters of the generated CA. The number of abstract heap graphs that one obtains this way is finite (modulo the absence of garbage)—therefore they can be encoded in the control of the CA. The translation produces a CA that is *bisimilar* to the original program, and therefore any (positive or negative) result obtained by analysing the CA holds for the program.

However, in the case of programs over trees, one cannot use the idea of [BBH$^+$06] to obtain a bisimilar CA since the number of branching nodes in a tree is unbounded. Therefore, the translation to CA that we propose in this chapter loses some information about the behaviour of the program, i.e., the semantics of the CA overapproximates the semantics of the original program. Then, if a spurious non-termination counterexample is detected over the generated CA, the translation is to be refined. The refinement is done by a specialised CEGAR loop that considers also structural information about the heaps. To the best of our knowledge, no such CEGAR loop was proposed before in the literature.

## Variance Analysis

The technique proposed in [DBCO06] is based on separation logic and it allows one to prove termination of programs manipulating singly-linked structures. As was said already above, the approach of [DBCO06] is similar to [BBH$^+$06] in that it tracks the length of the uninterupted list segments. However, it does not generate a CA simulating the original program. Instead, it is based on the following so-called *variance analysis*:

1. A safety check via *local shape analysis based on separation logic* [DOY06] is performed to obtain line invariants.

2. A set of *cutpoints* is defined—one cutpoint for each loop.

3. For each cutpoint, one performs the following steps:

    - Isolate the smallest strongly-connected part of the control flow graph (CFG) containing this cutpoint
    - Create the so-called *seed*—the line invariant of the cutpoint is enriched by constants carrying the current positions of pointer variables.
    - The analysis from Point 1 is executed only at the isolated part of the CFG with the seed taken as the initial set. At the end, the newly computed invariant for the line representing the cutpoint contains a reachability relation for all pointer variables – this relation (so-called *variance relation*) contains the mutual position of pointer variables and the constants representing their original position.
    - If the reachability relation is proved to be well-founded, then there is a bounded number of occurrences of this cutpoint in all possible runs.

The check of the well-foundness is a crucial point—as was already mentioned above, the authors use lengths of uninterrupted list segments here. Unlike the approach of [BBH$^+$06] (bisimulation preserving) and the approach we present in this chapter (based on CEGAR), the analysis of [DBCO06] fails if the initial abstraction is not precise enough.

The approach of [DBCO06] was recently generalised in [BCC$^+$07b] to a general framework that one can use to extend existing invariance analyses to variance analyses that can in turn be used for checking termination. Up to now, this framework has not been instantiated for programs with trees (by providing the appropriate measures and their abstract semantics). Moreover, it is not clear how the variance analysis framework fits with the CEGAR approach.

**Proving Termination of the Deutsch-Schorr-Wait Tree Traversal Using TVLA**

Termination of one specific program manipulating trees has been considered in [LRS06] where the Deutsch-Schorr-Waite tree traversal algorithm was proved to terminate. The key idea is to use a *state-depending* abstraction. In case of DSW, each node in a tree passes during the computation through the following four phases: (1) not yet visited, (2) the program works in the left subtree, (3) the program works in the right subtree, and (4) processing of the whole subtree (including the node itself) has been finished. At the beginning, all tree nodes are in the phase (1). When the program manipulates a tree node, it changes its state to the higher one. This observation allows one to establish bounds on the number of iterations.

## 5.2 The Notions Used

**Binary trees**

Given a finite set of *colors* $\mathcal{C}$, we define the *binary alphabet* $\Sigma_\mathcal{C} = \mathcal{C} \cup \{\Box\}$ where the *arity* function is $\#(c) = 2$ and $\#(\Box) = 0$. $\Pi$ denotes the set of tree positions $\{0,1\}^*$, i.e., the set of all sequences of 0's and 1's. Let $\lambda \in \Pi$ denote the empty sequence, and $p.q$ denote the concatenation of sequences $p, q \in \Pi$. $p \preceq q$ denotes the fact that $p$ is a prefix of $q$. A *binary tree* $t$ over $\mathcal{C}$ is a partial mapping $t : \Pi \to \Sigma_\mathcal{C}$ such that $dom(t)$ is a finite prefix-closed subset of $\Pi$, and for each $p \in dom(t)$:

- if $\#(t(p)) = 0$, then $t(p.0) = t(p.1) = \bot$,
- otherwise, if $\#(t(p)) = 2$, then $p.0, p.1 \in dom(t)$.

When writing $t(p) = \bot$, we mean that $t$ is undefined at position $p$. Let $t_\epsilon$ be the empty tree, $t_\epsilon(p) = \bot$ for all $p \in \Pi$. A *subtree* of $t$ starting at position $p \in dom(t)$ is a tree $t_{|p}$ defined as $t_{|p}(q) = t(pq)$ if $pq \in dom(t)$, and undefined otherwise. $t[p \leftarrow c]$ denotes the tree that is labelled as $t$, except at position $p$ where it is labelled with $c$. $t\{p \leftarrow u\}$ denotes the tree obtained from $t$ by replacing the $t_{|p}$ subtree with $u$. We denote by $\mathcal{T}(\mathcal{C})$ the set of all trees over the binary alphabet $\Sigma_\mathcal{C}$. A regular set of trees over the binary alphabet can be easily encoded as a tree automaton (cf. Section 3.1).

**Regularity-Preserving Tree Relations**

A relation $\varrho \subseteq \mathcal{T}(\mathcal{C}) \times \mathcal{T}(\mathcal{C})$ is a *regularity-preserving tree relation* iff for each regular tree language $L(A)$ described by some tree automaton $A$ over the alphabet $\Sigma_\mathcal{C}$, there exists a tree automaton $A'$ over $\Sigma_{\mathcal{C}}$ such that $L(A') = \varrho(L(A)) = \{t' \in \mathcal{T}(\mathcal{C}) \mid \langle t, t' \rangle \in \varrho\}$. For a sequence of regularity-preserving tree relations $\Omega = \varrho_1, \varrho_2, \ldots, \varrho_n$ where $\varrho_i \subseteq \mathcal{T}(\mathcal{C}) \times \mathcal{T}(\mathcal{C})$ for $1 \leq i \leq n$, $n \geq 1$, and a tree automaton $A$ over the alphabet $\Sigma_{\mathcal{C}}$, we define $\Omega(L(A)) = \varrho_n(\ldots(\varrho_2(\varrho_1(L(A)))))$. We also denote by $\varrho^*(L(A))$ the reflexive and transitive closure of the language $L(A)$ according to the relation $\varrho$. Note that a tree transducer (cf. Section 3.1) is a special case of a regularity-preserving tree relation.

### Counter Automata

For an arithmetic formula $\varphi$, let $FV(\varphi)$ denote the set of free variables of $\varphi$. For a set of variables $X$, let $\Phi(X)$ denote the set of arithmetic formulae with free variables from $X \cup X'$ where $X' = \{x' \mid x \in X\}$. If $\nu : X \to \mathbb{Z}$ is an assignment of $FV(\varphi) \subseteq X$, we denote by $\nu \models \varphi$ the fact that $\nu$ is a satisfying assignment of $\varphi$.

A counter automaton (CA) is a tuple $A = \langle X, Q, q_0, \varphi_0, \to \rangle$ where $X$ is the set of counters, $Q$ is a finite set of control locations, $q_0 \in Q$ is a designated initial location, $\varphi_0$ is an arithmetic formula such that $FV(\varphi_0) \subseteq X$, describing the initial assignments of the counters, and $\to \in Q \times \Phi(X) \times Q$ is the set of transition rules.

A configuration of a CA is a pair $\langle q, \nu \rangle \in Q \times (X \to \mathbb{Z})$. The set of all configurations is denoted by $\mathfrak{C}$. The transition relation $\xrightarrow[A]{\varphi} \subseteq \mathfrak{C} \times \mathfrak{C}$ is defined by $(q, \nu) \xrightarrow[A]{\varphi} (q', \nu')$ iff there exists a transition $q \xrightarrow{\varphi} q'$ such that if $\sigma$ is an assignment of $FV(\varphi)$ where $\sigma(x) = \nu(x)$ and $\sigma(x') = \nu'(x)$, we have that $\sigma \models \varphi$ and $\nu(x) = \nu'(x)$ for all variables $x$ with $x' \notin FV(\varphi)$. We denote by $\xrightarrow[A]{}$ the union $\bigcup_{\varphi \in \Phi} \xrightarrow[A]{\varphi}$, and by $\xrightarrow[A]{*}$ the reflexive and transitive closure of $\xrightarrow[A]{}$. A *run* of $A$ is a sequence of configurations $(q_0, \nu_0), (q_1, \nu_1), (q_2, \nu_2) \ldots$ such that $(q_i, \nu_i) \xrightarrow[A]{} (q_{i+1}, \nu_{i+1})$ for each $i \geq 0$ and $\nu_0 \models \varphi_0$. We denote by $\mathfrak{R}_A$ the set of all configurations reachable by $A$, i.e., $\mathfrak{R}_A = \{(q, \nu) \mid (q_0, \nu_0) \xrightarrow[A]{*} (q, \nu) \text{ for some } \nu_0 \models \varphi_0\}$.

## 5.3 Programs with Trees

### Program Syntax

We consider sequential, non-recursive C-like programs whose abstract syntax is given in Figure 5.1(a). Here, *Lab* is a finite set of program labels (control locations), *PVar* a finite set of pointer variables, and *Data* is a finite data domain. We assume that the programs work on tree-shaped data structures whose nodes contain a data element `data` with a value from the set *Data* and three next pointers, namely `left`, `right`, and `up`. For technical reasons, we require w.l.o.g. that there is no statements taking the control back to the initial line.

Programs manipulating singly-linked lists, doubly-linked lists, and trees without parent pointers may be considered a special case of the programs considered here if we suffice with the statements from Figure 5.1(a). We add syntactic sugar by allowing nesting of pointer expressions as, e.g., in the expression `x.left.data`, nesting of statements in conditional statements, composed conditions in conditional statements, and the use of

$l, l_1, l_2 \in \mathit{Lab},\ x, y, z \in \mathit{PVar},\ d, e, f \in \mathit{Data}$
$\mathit{Program} := \{l : \mathit{Stmnt};\}^*$
$\mathit{Stmnt} := \mathit{IfStmnt} \mid \mathit{Update} \mid \mathit{Asgn} \mid \mathit{Rotate} \mid \mathit{Goto}$
$\mathit{IfStmnt} := \texttt{if}\ (\mathit{Cond})\ \texttt{then goto}\ l_1;\ \texttt{else goto}\ l_2;$
$\mathit{Cond} := x == y \mid x == \texttt{null} \mid x.\mathit{data} == d$
$\mathit{Field} := \texttt{left} \mid \texttt{right} \mid \texttt{up}$
$\mathit{Asgn} := x = y\{.\mathit{Field}\} \mid x = \texttt{null} \mid x.\texttt{data} = d$
$\mathit{Rotate} := \texttt{leftRotate}(x) \mid \texttt{rightRotate}(x)$
$\mathit{Goto} := \texttt{goto}\ l$

```
x = root;
while (x != null) do
  if (x.left != null && x.left.data != marked)
  then x = x.left;
  else
    if (x.right != null && x.right.data != marked)
    then x = x.right;
    else x.data = marked;
         x = x.up;
```

Figure 5.1: (a) Abstract syntax of programs with trees and (b) a depth-first tree traversal procedure

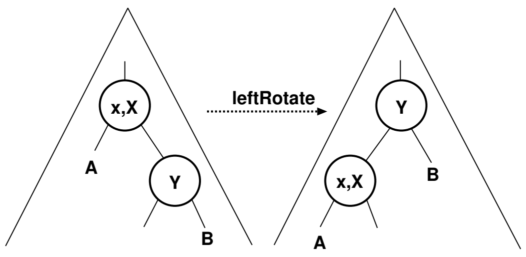


Figure 5.2: The left tree rotation

`while` statements. We consider the depth-first traversal (DFT) of a tree shown in Figure 5.1(b) as the running example of this paper.

## Concrete Semantics

Let $\mathcal{C} = \mathit{Data} \times 2^{\mathit{PVar}}$ be a set of colors and $\Sigma_{\mathcal{C}} = \mathcal{C} \cup \{\Box\}$ a binary alphabet. For any symbol $\sigma = \langle d, S\rangle \in \mathcal{C}$, we denote $\delta(\sigma) = d$ and $\nu(\sigma) = S$. By convention, $\delta(\Box) = \bot$ and $\nu(\Box) = \emptyset$. A concrete program state is a pair $s = \langle l, t\rangle$ where $l \in \mathit{Lab}$ is the current program label, and $t : \Pi \to \Sigma$ is the current working tree. We require that each variable from $\mathit{PVar}$ belongs to the set $\nu(t(p))$ for at most one position $p \in \mathit{dom}(t)$. Intuitively, memory configurations of the considered programs are represented as trees where a node is either null and represented by $\Box$, or it contains a data value and a set of pointer variables pointing to it.

We consider a special error state denoted by $\mathit{Err}$. In the state $\langle l, t\rangle$, a condition $x ==$ `null` in a conditional statement evaluates to true (and leads to a change of the control to the appropriately labelled line) iff $x \notin \nu(t(p))$ for all $p \in \mathit{dom}(t)$. A condition $\mathtt{x} == \mathtt{y}$ evaluates to true iff $x \in \nu(t(p)) \Leftrightarrow y \in \nu(t(p))$ for all $p \in \mathit{dom}(t)$. If $x \in \nu(t(p))$ for some $p \in \mathit{dom}(t)$, a condition $\mathtt{x}.\mathit{data} == d$ evaluates to true if $\delta(t(p)) = d$ and to false if $\delta(t(p)) \neq d$, else if $x \notin \nu(t(p))$ for all $p \in \mathit{dom}(t)$, its evaluation takes the control into $\mathit{Err}$. The concrete semantics of the most interesting assignment and rotation statements from Figure 5.1 is described by the rules shown in Figure 5.3 where $u_{left} = t_\epsilon\{\lambda \leftarrow t_{|p.1}\}\{0 \leftarrow t_{|p}\}\{0.1 \leftarrow t_{|p.1.0}\}$, and $u_{right} = t_\epsilon\{\lambda \leftarrow t_{|p.0}\}\{1 \leftarrow t_{|p}\}\{1.0 \leftarrow t_{|p.0.1}\}$. The semantics of the remaining assignment and rotation statements is straightforward.

$$\frac{\forall p \in dom(t) : x \notin \nu(t(p))}{\langle l,t\rangle \rightarrow \langle l',t\rangle} \qquad \frac{p \in dom(t) \quad x \in \nu(t(p))}{\langle l,t\rangle \rightarrow \langle l', t[p \leftarrow \langle \delta(t(p)), \nu(t(p)) \setminus \{x\}\rangle]\rangle}\ x = \texttt{null}$$

$$\frac{\begin{matrix} p \in dom(t) \quad y \in \nu(t(p)) \\ t(p.0) \neq \Box \quad \langle l,t\rangle \overset{x=null}{\longrightarrow} \langle l',t'\rangle \end{matrix}}{\langle l,t\rangle \rightarrow \langle l', t'[p.0 \leftarrow \langle \delta(t(p.0)), \nu(t(p.0)) \cup \{x\}\rangle]\rangle} \quad \frac{\begin{matrix} p \in dom(t) \quad y \in \nu(t(p)) \\ t(p.0) = \Box \quad \langle l,t\rangle \overset{x=null}{\longrightarrow} \langle l',t'\rangle \end{matrix}}{\langle l,t\rangle \rightarrow \langle l',t'\rangle} \quad \frac{\forall p \in dom(t)\ .\ y \notin \nu(t(p))}{\langle l,t\rangle \rightarrow Err}\ x = y.\texttt{left}$$

$$\frac{p \in dom(t) \quad x \in \nu(t(p))}{\langle l,t\rangle \rightarrow \langle l', t[p \leftarrow \langle d, \nu(t(p))\rangle]\rangle} \qquad \frac{\forall p \in dom(t)\ .\ x \notin \nu(t(p))}{\langle l,t\rangle \rightarrow Err}\ x.\texttt{data} = d$$

$$\frac{p \in dom(t) \quad x \in \nu(t(p)) \quad t(p.1) \neq \Box}{\langle l,t\rangle \rightarrow \langle l', t\{p \leftarrow u_{left}\}\rangle} \qquad \frac{\begin{matrix} (\forall p \in dom(t)\ .\ x \notin \nu(t(p))) \vee \\ (p \in dom(t)\ \wedge\ x \in \nu(t(p))\ \wedge\ t(p.1) = \Box) \end{matrix}}{\langle l,t\rangle \rightarrow Err}\ \texttt{leftRotate}(x)$$

$$\frac{p \in dom(t) \quad x \in \nu(t(p)) \quad t(p.0) \neq \Box}{\langle l,t\rangle \rightarrow \langle l', t\{p \leftarrow u_{right}\}\rangle} \qquad \frac{\begin{matrix} (\forall p \in dom(t)\ .\ x \notin \nu(t(p))) \vee \\ (p \in dom(t)\ \wedge\ x \in \nu(t(p))\ \wedge\ t(p.0) = \Box) \end{matrix}}{\langle l,t\rangle \rightarrow Err}\ \texttt{rightRotate}(x)$$

Figure 5.3: The concrete semantics of program statements

Given a program $P$, we denote by $\langle l,t\rangle \xrightarrow[P]{s} \langle l',t'\rangle$ the fact that $P$ has a transition leading from the first state to the second one caused by firing a statement $s$. We denote by $\xrightarrow[P]{}$ the union $\bigcup_{s \in Stmnt} \xrightarrow[P]{s}$, and by $\xrightarrow[P]{*}$ the reflexive and transitive closure of $\xrightarrow[P]{}$. For a program statement $s \in Stmnt$, let $post(s,I) = \{t' \mid \exists t \in I\ .\ t \xrightarrow[P]{s} t'\}$.

## 5.3.1 Program Preprocessing

Some of the considered program statements may influence the counters of the CA that we build to simulate programs in several different ways. For instance, after $x = x.\texttt{left}$, the distance of $x$ from the root may increase by one, but it may also become undefined (which we represent by the special value $-1$) if $x.\texttt{left}$ is null. Similarly, a single rotation statement may change the distance of a node pointed by some variable from the root in several different ways according to where the node is located in a concrete input tree. In particular, if some variable points into a subtree that moves up in a rotation, its distance from the root will decrease, whereas if the same variable does not point into this subtree, its distance from the root will not be modified (or it can increase if it points into the subtree moving down due to the rotation)—cf. Figure 5.2.

For technical reasons related to our abstraction refinement scheme, we need a one-to-one mapping between actions of a program and the counter manipulations simulating them. In order to ensure the existence of such a mapping, we decompose each program statement $s$ into a finite set of *instructions* $\mathcal{I}(s)$ with disjoint guards such that (1) the union of the semantics of these instructions is the semantics of the original statement, i.e., $post(s, T) = \bigcup_{i \in \mathcal{I}(s)} post(i, T)$ for any set of trees $T$, and (2) each instruction $i$ is represented in the counter automaton by a unique counter operation denoted by $\theta(i)$.

In particular, statements $x = \texttt{null}$ and $x = y$ correspond directly to instructions. Conditional statements of the form $x == \texttt{null}$ and $x == y$ are decomposed into two instructions each corresponding to their true and false branches. A conditional statement $x.\texttt{data} == d$ is decomposed into three instructions corresponding to its true and false branches and an error branch for the case $x == \texttt{null}$.

Every statement $x = y.\texttt{left}$ is split to three instructions, namely $\texttt{goLeftNull}(x, y)$ assigning null to $x$ iff $y$ is not null and $y.\texttt{left}$ is null, $\texttt{goLeftNonNull}(x, y)$ setting $x$ to $y.\texttt{left}$ iff both $y$ and $y.\texttt{left}$ are not null, and $\texttt{goLeftErr}(x, y)$ iff $y$ is null. This mainly distinguishes the cases when the distance of a variable from the root is increasing from when it becomes null. Similarly, we split each statement $x = y.\texttt{right}$ into instructions $\texttt{goRight}\{\texttt{Null}|\texttt{NonNull}|\texttt{Err}\}(x, y)$ and each statement $x = y.\texttt{up}$ into instructions $\texttt{goUp}\{\texttt{Null}|\texttt{NonNull}|\texttt{Err}\}(x, y)$.

Further, we split each statement $x.\texttt{data} = d_1$ into $\texttt{changeDataErr}(x)$ for the case of $x$ being null and $|Data|$ instructions $\texttt{changeData}(x, d_1, d_2)$ changing the value of $x.\texttt{data}$ from $d_1$ to $d_2$ for each $d_2 \in Data$. This explicitly shows the number of which data elements in the working tree is decreasing, resp. increasing.

For each choice of disjoint subsets $X, Y, A, B \subseteq PVar$ with $x \in X$, we split each statement $\texttt{leftRotate}(x)$ into instructions $\texttt{leftRotate}(x, X, Y, A, B)$ (see Figure 5.2). The set $X$ contains variables that are aliased to $x$, $Y$ contains variables pointing to the right son of $x$, $A$ contains variables pointing inside the left subtree of $x$, and $B$ contains variables pointing into the right subtree of the right son of $x$. Intuitively, counters corresponding to variables from the same set (and from $Pvar \setminus (X \cup Y \cup A \cup B)$) will be updated in the same way in the counter automaton (cf. Figure 5.2). In particular,

- for variables from $X$, the root distance will increase by one, and the subtree size will decrease by at least one,
- for variables from $Y$, the root distance will decrease by one, and the subtree size will increase by at least one,
- for variables from $A$, the root distance will increase by one, and the subtree size will stay the same,
- for variables from $B$, the root distance will decrease by one, and the subtree size will stay the same,
- for all other variables from $Pvar \setminus (X \cup Y \cup A \cup B)$, which may point either into the left subtree of the right son of $x$, point into nodes not below nor equal to $x$, or be null, the root distance as well as the subtree size do not change.

Formally, the semantics of $\texttt{leftRotate}(x, X, Y, A, B)$ is the same as the one of $\texttt{leftRotate}(x)$ with the following conditions added into the preconditions: $X = \nu(t(p))$, $Y = \nu(t(p.1))$, $A = \cup_{p.0.p' \in dom(t)} \nu(t(p.0.p'))$, and $B = \cup_{p.1.1.p' \in dom(t)} \nu(t(p.1.1.p'))$

where $x \in \nu(t(p))$. We then also add an instruction leftRotateErr$(x)$ for the case of $x$ or its right son being null. The rightRotate$(x)$ statements are split analogously.

In general, the number of instructions corresponding to a single statement may be exponentially large in the number of program variables. However, in practice, the use of program invariants (see Section 5.6) will greatly restrict the number of instructions in the model. Let $Instr = \bigcup_{s \in Stmnt} \mathcal{I}(s)$ be the set of all instructions. We introduce the notation $\xrightarrow[P]{i}$ for $i \in Instr$ in a way analogous to $\xrightarrow[P]{s}$ for $s \in Stmnt$.

## 5.4 The Termination Analysis Loop

Our termination analysis procedure based on abstraction refinement is schematically depicted in Figure 5.4. We start with the control flow graph (CFG) of the given program and use ARTMC (cf. Section 3.3) to generate invariants for the control points of the CFG and also to check that the program is *free of basic memory inconsistencies* like null pointer dereferences. If the user wants, ARTMC may also be used to check many further safety properties of the programs (shape invariants, data-oriented properties, etc.) in a similar way, as we described in Section 4.2.2. The use of ARTMC in order to generate these invariants is briefly described in Section 5.5.

Subsequently, the CFG annotated with the invariants (an abstract control flow graph—ACFG, see Section 5.6) is converted into a counter automaton (CA)[2], which is checked for termination using an existing tool (e.g., [Ryb]). If the CA is proved to terminate, termination of the program is proved too. Otherwise, the termination analyzer outputs a lasso-shaped counterexample. We check whether this counterexample is real in the CA—if not, we suggest the use of another CA termination checker (for brevity, we skip this in Figure 5.4). If the counterexample is real on the CA, it is translated back into a sequence of program instructions and analyzed for spuriousness on the program. If the counterexample is found to be real even there, the procedure reports non-termination. Otherwise, the program ACFG is refined by splitting some of its nodes (actually, the sets of program configurations associated with certain control locations), and the loop is reiterated. Moreover, ARTMC may also be re-run to refine the invariants used.

The pseudocode in Figure 5.5 presents the abstract-check-refine loop that we propose in a more detailed view. The fact that the inner-most loop of our procedure terminates for some $K$ (provided no destructive updates are used in the loop, or there exists some counter automaton of the form considered by us that may be used to show termination of the given program) is proved in Section 5.7. The number of times ACFGs may be refined purely by splitting their control locations is given by the (user-defined) constant $MaxSteps$. If $MaxSteps$ of refining ACFGs by splitting their locations are not enough, we re-run ARTMC with a more precise abstraction in order to refine the program line invariants from which ACFGs are built. In such a case, ARTMC is re-run on the refined CFG underlying the last computed ACFG in order to preserve the effect of the splitting steps done so-far.

[2]The use of invariants in the ACFGs allows us to remove impossible transitions and therefore improves the accuracy of the translation to CA.

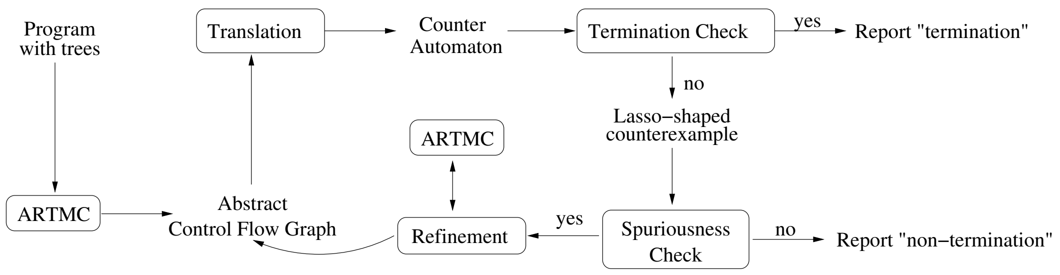


Figure 5.4: The proposed abstract-check-refine loop

If our termination analysis stops with either a positive or a negative answer, the answer is exact. However, we do not guarantee termination for any of these cases. Indeed, this is the best we can achieve as the problem we handle is not recursively enumerable even when destructive updates (i.e., tree rotations) are not allowed. This can be proved by a reduction from the complement of the halting problem for 2-counter automata [Min67].

**Theorem 2** *The problem whether a program with trees without destructive updates terminates on any input tree is not recursively enumerable.*

Therefore we do not further discuss termination guarantees for our analysis procedure and postpone the research on potential partial guarantees, in some restricted cases, for the future. However, despite the theoretical limits, the experimental results reported in Section 5.9 indicate a practical usefulness of our approach.

## 5.5 Using ARTMC in the Proposed Framework

We use *abstract regular tree model checking* (ARTMC, cf. Section 3.3) as a part of our verification framework. The idea of ARTMC is to represent each program configuration as a tree over a finite alphabet, regular sets of such configurations by finite tree automata, and program instructions as regularity-preserving tree relations. Then an overapproximation of the set of reachable configurations is fully automatically computed. Within this computation, so-far computed sets of reachable configurations (represented as tree automata) are abstracted in order to make the computation terminating.

In this section, we describe our encoding of memory configurations into trees (different from the one used in Section 4.4.1), our encoding of the considered program instructions into regularity preserving tree relations (especially the encoding of tree rotations), and the use of ARTMC in a similar way to the one described in Section 4.4.4.

### Encoding of Configurations and Instructions

The memory configurations may be encoded directly as trees from the set $\mathcal{T}(\mathcal{C})$ over the binary alphabet $\Sigma_\mathcal{C} = Data \times 2^{PVar} \cup \{\square\}$—provided that we are given a program with pointer variables $PVar$ and a set of tree data values $Data$. As explained already in

```
AbstractionHeight := 1;  // The ARTMC preciseness parameter (for the finite-height abstraction schema).
$F$ := the control flow graph of $P$;
while (true) {
        $G$ := ARTMC(AbstractionHeight, $F$, $Init$);
          // Use ARTMC to compute the ACFG $G$ over the CFG $F$ for the initial set of trees $Init$.
        if a safety error was found, report it and exit;
        $N$ := 1;
        while ($N \leq$ MaxSteps) {
                $A$ := translateACFGintoCA($G$);
                test termination of $A$ using an existing CA termination checker $TC$;
                if $TC$ reports termination for $A$, report termination of $P$ and exit;
                let $S.L$ be the lasso-shaped counterexample for $A$;  // $S$ – stem, $L$ – loop
                if $A$ terminates on $S.L$, but $TC$ fails to show this, report that $TC$ is not strong enough and exit;
                $S_P.L_P$ := translateLassoFromCABackToProgram($S.L$);
                $K$ := 1; $I$ := the set of trees reachable via $S_P$ from $Init$;
                while (true) {
                        test non-termination for the trees of height $K$ from $Init$;
                        if non-termination is witnessed, report non-termination and exit;
                        $I$ := the set of trees reachable via $L_P$ from $I$;
                        if ($I == \emptyset$) break;  // The lasso breaks for all input trees after at most $K$ iterations.
                        $K$ := $K$ + 1;
                }
                $G$ := refineACFG($G$, $S_P.L_P$, $K$);
                $N$ := $N$ + 1;
        }
        AbstractionHeight := AbstractionHeight + 1;
        $F$ := the refined CFG (some nodes are split) behind the current ACFG $G$;
}
```

Figure 5.5: A detailed view of the proposed abstract-check-refine loop

Section 5.3, a tree node is either a leaf and represented by $\square$, or it contains a data value from the finite set $Data$ and a set of pointer variables pointing to it. Each pointer variable can point to at most one tree node (if it is null, it does not appear in the tree).

Most of the program instructions that we consider can be encoded as structure-preserving tree transducers. Transducers can check conditions like $x ==$ `null`, $x == y$, or $x$.`data` $== d$ by simply checking whether $x$ appears in some node of an input tree, whether $x$ and $y$ label the same tree node, or whether a node labelled with $x$ is also labelled with the date item $d$, respectively. Transducers can also be used to move symbols representing the variables to nodes marked by some other variable ($x = y$), remove a symbol representing a variable from the tree ($x =$ `null`), move it one level up or down (`go`{`Left`|`Right`|`Up`}`NonNull`$(x, y)$), or change the data element in the node marked by some variable from one value to another one (`changeData`$(x, d_1, d_2)$).

The instructions for tree rotations cannot be modelled by finite-state tree transducers. However, they (and their inverse) can still be expressed as regularity-preserving relations whose image can be effectively computed for any regular set using a special operation on tree automata. First, the test of the distribution of the program variables in the sets $X, Y, A, B$ used in instructions {`left`|`right`}`Rotate`$(x, X, Y, A, B)$ can be easily implemented by an intersection with tree automata encoding the various possible mutual positions of the program variables.[3] The actual rotation is then the same for all the variants of the instruction. Let $M = (Q, \Sigma_\mathcal{C}, F, \delta)$ be a tree automaton describing a set of of reachable configurations. The set of configurations reachable from $\mathcal{L}(M)$ via an instruction `leftRotate`$(x, X, Y, A, B)$ can be represented by a tree automaton $M'$ that is obtained by the following algorithm (the case of `rightRotate`$(x, X, Y, A, B)$ being analogical):

- Start with $M' = M$.
- For all rules $c_x(q_1, q_2) \to q_t \in \Delta$ where $q_1, q_2, q_t \in Q$, $c_x \in \mathcal{C}$, and $x \in \nu(c_x)$ perform the following steps:
  - Remove rules of the form $\Box \to q_2$ from $\Delta'$—these rules (if present) cause the control to be taken to the special error state $Err$ by the `leftRotateErr`$(x)$ instruction.
  - Remove the rule $c_x(q_1, q_2) \to q_t$ from $\Delta'$.
  - For each rule $c_y(q_3, q_4) \to q_2 \in \Delta$ where $q_3, q_4 \in Q$ and $c_y \in \mathcal{C}$, do the following:
    - Choose a fresh state $q_m \notin Q'$.
    - Extend $\Delta'$ by the rules $c_x(q_1, q_3) \to q_m$ and $c_y(q_m, q_4) \to q_t$ and add $q_m$ into $Q'$.

The idea of the algorithm is showed in Figure 5.6. Note that there can be several rules of the type $c_x(q_1, q_2) \to q_t$ with $x \in \nu(c_x)$ in the automaton $M$, but in each tree encoding a valid memory configuration there is at most one node pointed to by the variable $x$, and hence just one of these rules can be used in any accepting run. Therefore, by removing all of these rules, we safely break all the input trees just on the requested position. Then we can reconnect the parts of the broken trees by adding new rules. For each removed rule $c_x(q_1, q_2) \to q_t$, we find all rules of the type $c_y(q_3, q_4) \to q_2$, and for each of them, we add new rules $c_x(q_1, q_3) \to q_m$ and $c_y(q_m, q_4) \to q_t$ for a fresh state $q_m$. The remaining rules of the automaton are preserved, and so the parts of the trees around the rotation area are preserved. Also, note that all rules of the type $c_y(q_3, q_4) \to q_2$ must be preserved because they can be used outside of the rotation point too.

**The ARTMC Computation Loop and the ARTMC Abstractions Used**

Similarly to the case described in Section 4.4.4, we compute a separate set of reachable configurations for each control point of the (possibly refined) CFG of the given program, we do not iterate a single transition relation as within the basic ARTMC framework (cf.

[3]As an optimization, one does not have to generate automata for all (exponentially many) theoretically possible distributions of the variables, but instead generate just automata for testing separately the position of each program variable wrt. the node pointed by $x$, intersect the current set of reachable configurations with the sets represented by these automata, and then use the obtained information to deduce which of the theoretical possibilities are feasible (in practice, there are typically only a few of them).

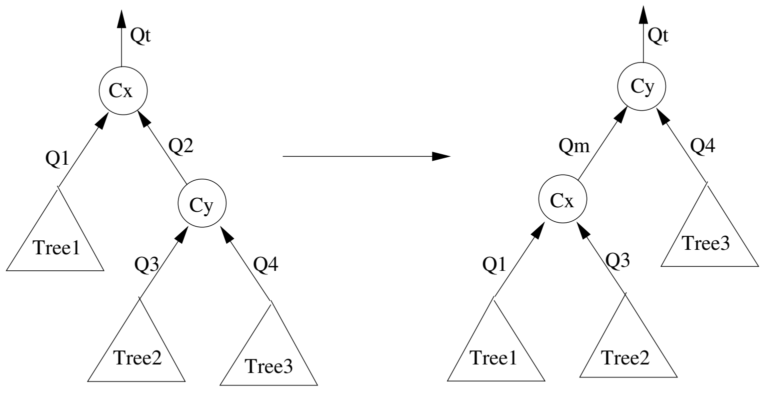


Figure 5.6: The left rotation on tree automata

3.3). Instead, we iteratively fire the instructions decorating arcs of the program CFG (i.e., we compute the image of the corresponding tree relations) whenever the set of configurations reachable at their source location changes. The computed image is then united with the already known set of reachable configurations for the appropriate target location and the obtained set (represented by a tree automaton) is abstracted.

Out of the abstractions commonly used in the ARTMC framework (cf. Section 3.3), we can use both the abstraction based on predicate languages as well as the finite-height abstraction in the initial generation of reachable configurations. These two abstractions can also be combined in such a way that two automata states are equal when they satisfy the same predicate languages and accept the same trees up to some height. If the combination is used, predicate abstraction may be refined only when generating the invariants and within the generation a spurious error is found. When refining due to a need to get rid of some lasso, one only increases the height and keeps the predicates so-far obtained ahead. However, we never use the predicate-based abstraction alone because its refinement schema requires one to have finite (not lasso-shaped) counterexamples.

## 5.6 Abstraction of Programs with Trees into Counter Automata

In this section, we provide a translation from tree manipulating programs to counter automata such that existing techniques for proving termination of counter automata can be used to prove termination of the programs. Before describing the translation, we define the simulation notion that we will use to formalize correctness of the translation.

Let $P$ be a program with a set of instructions $Instr$, an initial label $l_0 \in Lab$, a set of input trees $I_0 \subseteq \mathcal{T}(\mathcal{C})$, and a set of reachable configurations $\mathcal{R}_P \subseteq Lab \times \mathcal{T}(\mathcal{C})$. Let us also have a counter automaton $A = \langle X, Q, q_0, \varphi_0, \rightarrow \rangle$ with $\rightarrow \in Q \times \Phi(X) \times Q$, and a set of reachable configurations $\mathfrak{R}_A$. A function $M : X \times \mathcal{T}(\mathcal{C}) \rightarrow \mathbb{Z}$ is said to be a *measure* assigning counters integer values for a particular tree[4]. Let $\mathbf{M}(t) = \{M(x,t) \mid x \in X\}$.

**Definition 1** *The program $P$ is simulated by the counter automaton $A$ w.r.t. $M : X \times \mathcal{T}(\mathcal{C}) \to \mathbb{Z}$ and $\theta : Instr \to \Phi$ iff there exists a relation $\sim\ \subseteq \mathcal{R}_P \times \mathfrak{R}_A$ such that (1) $\forall t_0 \in I_0 : \mathbf{M}(t_0) \models \varphi_0 \ \wedge\ \langle l_0, t_0\rangle \sim \langle q_0, \mathbf{M}(t_0)\rangle$ and (2) $\forall (l_1,t_1),(l_2,t_2) \in \mathcal{R}_P\ \forall i \in Instr\ \forall (q_1,\nu_1) \in \mathfrak{R}_A : (l_1,t_1) \xrightarrow{i} (l_2,t_2) \ \wedge\ (l_1,t_1) \sim (q_1,\nu_1) \ \Rightarrow\ \exists (q_2,\nu_2) \in \mathfrak{R}_A : (q_1,\nu_1) \xrightarrow{\theta(i)} (q_2,\nu_2) \ \wedge\ (l_2,t_2) \sim (q_2,\nu_2)$.*

The measure $M$ ensures that the counters are initially correctly interpreted over the input trees, whereas $\theta$ ensures that the counters are updated in accordance with the manipulations done on the trees. Simulation in the sense of Definition 1 guarantees that if we prove termination of the CA, the program will terminate on any $t \in I_0$.

## 5.6.1 Abstract Control Flow Graphs

According to the termination analysis loop proposed in Figure 5.4, we construct the CA simulating a program in two steps: we first construct the so-called *abstract control flow graph* (ACFG) of a program, and then translate it into a CA. Initially, the ACFG of a program is computed from its CFG by decorating its nodes with ARTMC-overapproximated sets of configurations reachable at each line (we keep the initial set of trees exact exploiting the fact that w.l.o.g. there are no statements leading back to the initial line). These sets allow us to exclude impossible (not fireable) transitions from the ACFG and thus derive a more exact CA. Further, in subsequent refinement iterations, infeasible termination counterexamples are excluded by splitting these sets (if this appears to be insufficient, we re-run ARTMC to compute a better overapproximation of the reachable sets of configurations). Below, we first define the notion of ACFG, then we provide its translation to counter automata.

In what follows, let $P$ be a program (pre-processed according to Section 5.3.1) with instructions $Instr$, working on trees from $\mathcal{T}(\mathcal{C})$, and let $l_0 \in Lab$ be the initial line of $P$. The *control flow graph* (CFG) of $P$ is a labelled graph $F = \langle Instr, Lab, l_0, \Rightarrow\rangle$ where $l \overset{i}{\Rightarrow} l'$ denotes the presence of an instruction $i$ between control locations $l, l' \in Lab$. We further suppose that the input tree configurations for $P$ are described by the user as a (regular) set of trees $I_0 \subseteq \mathcal{T}(\mathcal{C})$. An *abstract control flow graph* (ACFG) for $P$ is then a graph $G = \langle Instr, LI, \langle l_0, I_0\rangle, \longmapsto\rangle$ where $LI$ is a *finite* subset of $Lab \times 2^{\mathcal{T}(\mathcal{C})}$, $\langle l_0, I_0\rangle \in LI$, and there is an edge $\langle l, I\rangle \overset{i}{\longmapsto} \langle l', I'\rangle$ iff $l \overset{i}{\Rightarrow} l'$ in the CFG of $P$ and $post(i, I) \cap I' \neq \emptyset$. For a finite sequence of instructions $\pi : i_1 i_2 \ldots i_n$, we denote by $l \overset{\pi}{\longmapsto} l'$ the existence of a path labelled with $\pi$ between $l$ and $l'$ in $G$.

Note that as we always work with ACFGs annotated with regular sets of configurations and we can implement the effect of each instruction on a regular set as an operation on tree automata, we can easily implement the test $post(i, I) \cap I' \neq \emptyset$ needed for computing the edges of ACFGs. As an optimization of introducing edges corresponding to

[4]Intuitively, certain counters will measure, e.g., the distance of a certain node from the root, the size of the subtree below it, etc.

$\{$`left`$|$`right`$\}$`Rotate`$(x, X, Y, A, B)$, we might first test for each pointer variable in which of the subtrees in Figure 5.2 it appears relative to the node pointed by $x$ (which can be tested by an intersection with simple tree automata encoding the various mutual positions). Then, we take into account only the sets $X, Y, A, B$ based on these possibilities (which in practice are typically only a few) rather than blindly testing all (exponentially many) theoretically possible variants of the distribution of the variables in $X, Y, A, B$. Note also that a location in $P$ may correspond to more than one locations in $G$.

We say that *$G$ covers the invariants of $P$* whose set of reachable states is $\mathcal{R}_P$ iff each tree $t \in \mathcal{T}(\mathcal{C})$ that is reachable at a program line $l \in Lab$ (i.e., $\langle l, t\rangle \in \mathcal{R}_P$), appears in some of the sets of program configurations associated with $l$ in the locations of $G$. Formally, $\forall l \in Lab : \mathcal{R}_P \cap (\{l\} \times \mathcal{T}(\mathcal{C})) \subseteq \{l\} \times \bigcup_{\langle l,I\rangle \in LI} I$. The following lemma captures the relation between the semantics of a program and that of an ACFG.

**Lemma 1** *Let $P$ be a program with trees and $G$ an ACFG that covers the invariants of $P$. Then, the semantics of $G$ simulates that of $P$ in the classical sense.*

For illustration, in Figures 5.7 and 5.8, we present the CFG and the initial ACFG of the DFT program from Figure 5.1(b) obtained by labelling the DFT by invariants computed by ARTMC. The invariants are depicted such that the shaded part represents the (possibly empty) part of a tree with only marked nodes, the white part represents the (possibly empty) part of a tree with only unmarked nodes, and the part with the question mark represents the part of the tree where the nodes may be both marked as well as unmarked. The upper circle represents the node pointed by the variable $x$ (if it is white, the node is unmarked, if it is black, the node is marked), and the lower circle represents the node pointed by the variable $y$. There is just one circle in the cases where $x == y$ or $y ==$ `null`. These circles represent the relative positions of the nodes pointed by the variables $x$ and $y$ wrt. the marked/unmarked areas of the tree.

Note that all arcs leading to the error location $Err$ as well one of the arcs between nodes 10 and 11 from the CFG are missing in the ACFG. This is a consequence of using the invariants. In fact, a removal of the arc between nodes 10 and 11 will later lead to a CA allowing us to show that DFT terminates.

## 5.6.2 Translation to Counter Automata

We now describe the construction of a CA $A_{rsc}(G) = \langle X, Q, q_0, \varphi_0, \rightarrow\rangle$ from an ACFG $G = \langle Instr, LI, \langle l_0, I_0\rangle, \longmapsto\rangle$ of a program $P$ such that $A_{rsc}(G)$ simulates $P$ in the sense of Def. 1. We consider two sorts of counters, i.e., $X = X_{PVar} \cup X_{Data}$ where $X_{PVar} = \{r_x \mid x \in PVar\} \cup \{s_x \mid x \in PVar\}$ and $X_{Data} = \{c_d \mid d \in Data\}$. For each $x \in PVar$, the counter $r_x$ keeps track of the current distance of the node $n$ pointed by $x$ from the root of $t$, whereas the counter $s_x$ keeps track of the size of the subtree below $n$ (with both $r_x$ and $s_x$ being set to $-1$ iff $x$ is null). For each $d \in Data$, $c_d$ is contains the number of $d$-valued nodes of $t$. Formally, using the notion of measures, we define for each $x \in PVar$ that $M_{rsc}(r_x, t) = |p|$ and $M_{rsc}(s_x, t) = |dom(t_{|p})|$ if there exists $p \in dom(t)$ such that $x \in \nu(t(p))$, and $M_{rsc}(r_x, t) = M_{rsc}(s_x, t) = -1$ otherwise. For each $d \in Data$, we define $M_{rsc}(c_d, t) = |\{p' \in dom(t) \mid \delta(t(p')) = d\}|$.

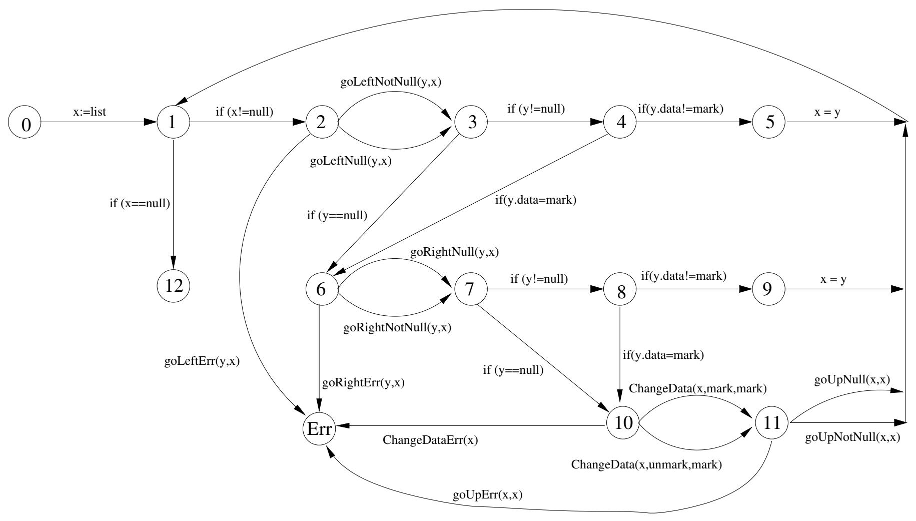


Figure 5.7: The CFG of the DFT procedure from Figure 5.1(b)

We build $A_{rsc}(G)$ from $G$ by simply replacing the instructions on edges of $G$ by operations on counters. Formally, this is done by the translation function $\theta_{rsc}$ defined in Table 5.1. The mapping for the instructions $x = y.\texttt{right}$ and $\texttt{rightRotate}(x, X, Y, A, B)$ is skipped in Table 5.1 as it is analogous to that of $x = y.\texttt{left}$ and $\texttt{leftRotate}(x, X, Y, A, B)$, respectively. Also, for brevity, we skip the instructions leading to the error state $Err$. As a convention, if the future value of a counter is not explicitly defined, we consider that the future value stays the same as the current value. Moreover, in all formulae, we assume an implicit guard $-1 \leq r_x < TreeHeight \ \wedge \ -1 \leq s_x < TreeSize$ for each $x \in PVar$[5] and $0 \leq c_d \leq TreeSize \ \wedge \ \sum_{d \in Data} c_d = TreeSize$ for each $d \in Data$. $TreeHeight$ and $TreeSize$ are parameters restricting the range in which the other counters can change according to a given input tree. They are needed as a basis on which termination of the resulting automaton can be shown.

Next, we define $Q = LI$, $q_0 = \langle l_0, I_0 \rangle$, and $q \xrightarrow{\theta_{rsc}(i)} q'$ iff $q \overset{i}{\longmapsto} q'$ for all $i \in Instr$. The initial constraint $\varphi_0$ on the chosen counters can be automatically computed[6] from the regular set of input trees $I_0$ such that it satisfies requirement (1) of Definition 1. The following theorem shows the needed simulation relation between the counter automata we construct and the programs.

To illustrate the construction of counter automata from ACFGs, we present in Figure 5.9 the counter automaton built from the ACFG from Figure 5.8 of our running example, i.e., the DFT procedure from Figure 5.1(b).

[5] $-1$ corresponds to $x$ being null.

[6] This can be done by computing the Parikh image of a context-free language $\mathcal{L}(I_0)$ corresponding to the regular tree language $I_0$. For each tree $t \in I_0$ there is a word in $\mathcal{L}(I_0)$ consisting of all nodes of $t$. We use special symbols to denote the position of a node in the tree relative to a given variable (under the variable, between it and the root) and the data values of nodes.

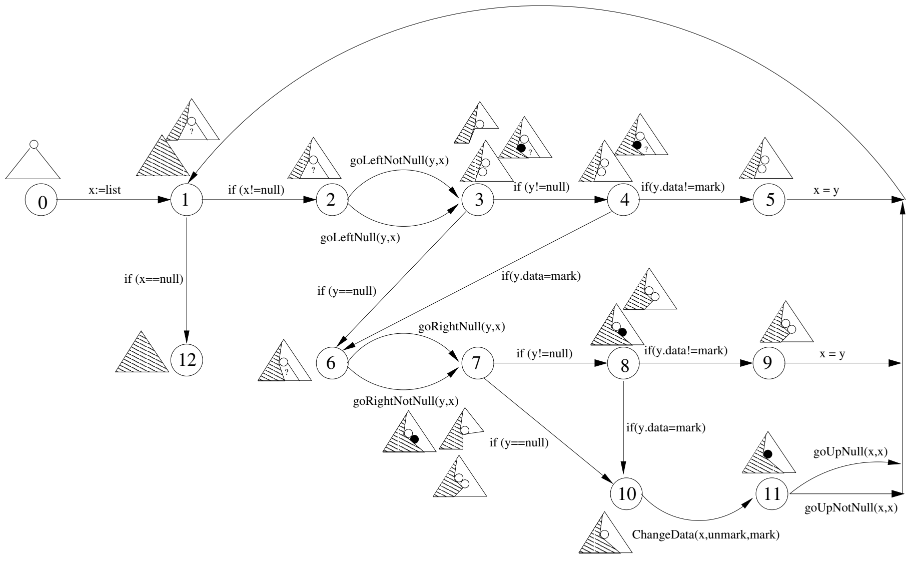


Figure 5.8: The initial ACFG of the DFT procedure from Figure 5.1(b)

**Theorem 3** *Given a program $P$ and an ACFG $G$ of $P$ covering its invariants, the CA $A_{rsc}(G)$ simulates $P$ in the sense of Definition 1 wrt. $\theta_{rsc}$ and $M_{rsc}$.*

The counter automata $A_{rsc}(G)$ that we generate have further the property that each transition $q \xrightarrow{\varphi} q'$ can be mapped back into the program instruction from which it originates. This is because the instructions onto which we decompose each program statement are assigned different formulae, by the translation function $\theta_{rsc}$, and there is at most one statement between each two control locations of the program. Formally, we capture this by a function $\xi : Q \times \Phi \times Q \to Instr$ such that $\forall q_1, q_2 \in Q, \varphi \in \Phi : q \xrightarrow{\varphi} q' \Rightarrow q \overset{\xi(q_1,\varphi,q_2)}{\longmapsto} q'$. We generalize $\theta_{rsc}$ and $\xi$ to sequences of transitions, i.e., for a path $\pi$ in $A_{rsc}$, $\xi(\pi)$ denotes the sequence of program instructions leading to $\pi$, and $\theta_{rsc}(\xi(\pi))$ denotes the sequence of counter operations on $\pi$ obtained by projecting out the control locations from $\pi$.

## 5.7 Checking Spuriousness of Counterexamples

Since the CA $A_{rsc}$ generated from a program $P$ with trees is a simulation of $P$ (cf. Theorem 3), proving termination of $A_{rsc}$ suffices to prove termination of $P$. However, if $A_{rsc}$ is not proved to terminate by the termination checker of choice, there are three possibilities:

1. $A_{rsc}$ terminates, but the chosen termination checker did not find a termination argument.

2. Both $A_{rsc}$ as well as $P$ do not terminate.

Table 5.1: The mapping $\theta_{rsc}$ from program instructions to counter manipulations

| instruction $i$ | counter manipulation $\theta_{rsc}(i)$ |
|---|---|
| $\texttt{if}(x == \texttt{null})$ | $r_x = -1$ |
| $\texttt{if}(x! = \texttt{null})$ | $r_x \geq 0$ |
| $\texttt{if}(x == y)$ | $r_x = r_y \wedge s_x = s_y$ |
| $\texttt{if}(x! = y)$ | $true$ |
| $\texttt{if}(x.\texttt{data} == d)$ | $r_x \geq 0 \wedge c_d \geq 1$ |
| $\texttt{if}(x.\texttt{data}! = d)$ | $r_x \geq 0 \wedge c_d < TreeSize$ |
| $x = \texttt{null}$ | $r'_x = s'_x = -1$ |
| $x = y$ | $r'_x = r_y \wedge s'_x = s_y$ |
| $\texttt{goLeftNull}(x, y)$ | $r_y \geq 0 \wedge s_y \geq 1 \wedge r'_x = s'_x = -1$ |
| $\texttt{goLeftNonNull}(x, y)$ | $r_y \geq 0 \wedge s_y \geq 2 \wedge r'_x = r_y + 1 \wedge s'_x < s_y$ |
| $\texttt{goUpNull}(x, y)$ | $r_y = 0 \wedge s_y \geq 1 \wedge r'_x = s'_x = -1$ |
| $\texttt{goUpNonNull}(x, y)$ | $r_y \geq 1 \wedge s_y \geq 1 \wedge r'_x = r_y - 1 \wedge s'_x > s_y$ |
| $\texttt{changeData}(x, d, d)$ | $r_x \geq 0 \wedge s_x \geq 1 \wedge c_d > 0$ |
| $\texttt{changeData}(x, d_1, d_2), d_1 \neq d_2$ | $r_x \geq 0 \wedge s_x \geq 1 \wedge c_{d_1} > 0 \wedge c'_{d_2} = c_{d_2} + 1 \wedge c'_{d_1} = c_{d_1} - 1$ |
| $\texttt{leftRotate}(x, X, Y, A, B)$ | $gLeftRotate(x, X, Y, A, B) \wedge aLeftRotate(x, X, Y, A, B)$ |

$$\begin{array}{l}
gLeftRotate(x, X, Y, A, B) = \\
\quad r_x \geq 0 \ \wedge \ s_x \geq 2 \ \wedge \\
\quad (\forall v \in X : r_v = r_x \ \wedge \ s_v = s_x) \ \wedge \\
\quad (\forall v, v' \in Y : r_v = r_x + 1 \ \wedge \ s_v < s_x \ \wedge \\
\qquad\qquad r_v = r_{v'} \ \wedge \ s_v = s_{v'}) \ \wedge \\
\quad (\forall v \in A : r_v \geq r_x + 1 \ \wedge \ s_v < s_x) \ \wedge \\
\quad (\forall v \in B : r_v \geq r_x + 2 \ \wedge \ s_v < s_x - 1)
\end{array}
\qquad
\begin{array}{l}
aLeftRotate(x, X, Y, A, B) = \\
\\
\quad (\forall v \in X : r'_v = r_v + 1 \ \wedge \ s'_v < s_v) \ \wedge \\
\quad (\forall v \in Y : r'_v = r_v - 1 \ \wedge \ s'_v > s_v) \ \wedge \\
\\
\quad (\forall v \in A : r'_v = r_v + 1 \ \wedge \ s'_v = s_v) \ \wedge \\
\quad (\forall v \in B : r'_v = r_v - 1 \ \wedge \ s'_v = s_v)
\end{array}$$

3. $P$ terminates, but $A_{rsc}$ does not, as a consequence of the abstraction used in its construction.

In all these three cases, the CA termination checker outputs a counterexample consisting of a finite path (stem) that leads to a cycle, both paths forming a lasso.

Formally, given a CA $A = \langle X, Q, q_0, \varphi_0 \rangle$, a lasso is a path in the control graph of $A$ of the form $S.L$ where $S = q_0 \xrightarrow{\varphi^s_1} q^s_1 \xrightarrow{\varphi^s_2} q^s_2 \xrightarrow{\varphi^s_3} \ldots q^s_{m-1} \xrightarrow{\varphi^s_m} q^l_1$ and $L = q^l_1 \xrightarrow{\varphi^l_1} q^l_2 \xrightarrow{\varphi^l_2} q^l_3 \xrightarrow{\varphi^l_3} \ldots q^l_n \xrightarrow{\varphi^l_n} q^l_1$. The lasso is said to be *spurious* if and only if there exists a non-terminating run $\langle q_0, \nu_0 \rangle \xrightarrow{\varphi^s_1} \ldots \langle q^s_{m-1}, \nu_{m-1} \rangle \xrightarrow{\varphi^s_m} \big( \langle q^l_1, \nu_m \rangle \xrightarrow{\varphi^l_1} \ldots \xrightarrow{\varphi^l_n} \langle q^l_1, \nu_m \rangle \big)^\omega$ in $A$ where $\nu_0 \models \varphi_0$, and $P$ does not have an infinite run along the path $\xi(q_0, \varphi^s_1, q^s_1) \ldots \xi(q^s_{m-1}, \varphi^s_{m-1}, q^l_1) \big( \xi(q^l_1, \varphi^l_1, q^l_2) \ldots \xi(q^l_n, \varphi^l_n, q^l_1) \big)^\omega$ for any $t \in I_0$.

The three above mentioned cases that may arise when we are given a lasso shaped counterexample over the generated CA are dealt with in the upcoming paragraphs.

**Deciding termination of CA lassos.** We first show that termination of a given control loop is decidable in a CA whose transition relations are conjunctions of difference constraints, i.e. formulae of the forms $x - y \leq c$, $x' - y \leq c$, $x - y' \leq c$, or $x' - y' \leq c$ where

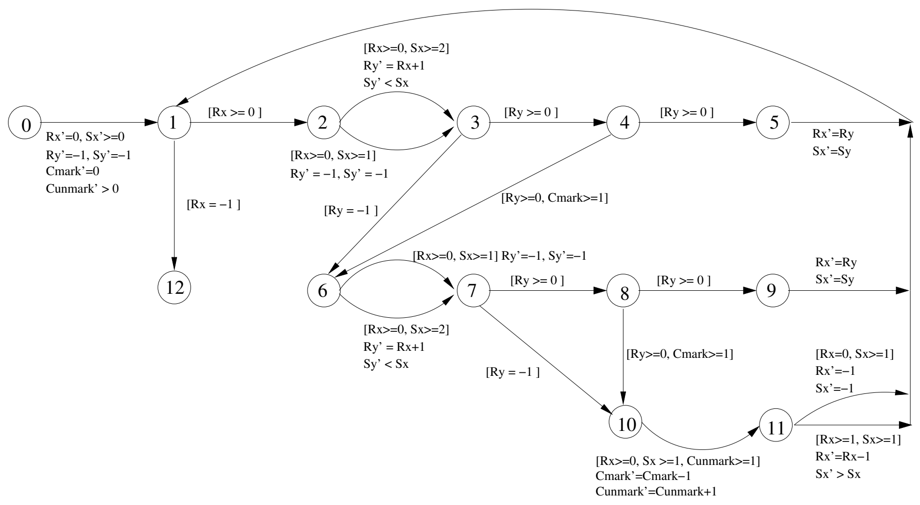


Figure 5.9: The CA built from the ACFG from Figure 5.8 of the DFT program from Figure 5.1(b)

$x'$ denotes the future value of the counter $x$ and $c \in \mathbb{Z}$ is an arbitrary integer constant. For this type of CA, the composed transition relation of the given control loop is also expressible as a conjunction of difference constraints. Then, this relation can be encoded as a constraint graph G such that the control loop terminates iff G contains a negative cycle (for details see Appendix A.3). Using the results of [CJ98, BIL06], this fact can be encoded as a Presburger formula and hence decided. At the same time, it is clear that the CA generated via the translation function $\theta_{rsc}$ fall into the described class of CA. In particular, the constraint that each counter is bounded from below by $-1$ and from above by the $TreeHeight$ or $TreeSize$ parameters is expressible using difference constraints.[7]

**Theorem 4** *Let $A = \langle X, Q, q_0, \varphi_0, \rightarrow \rangle$ be a counter automaton with transition relations given as difference constraints. Then, given a control loop in A, the problem whether there exists an infinite computation along the loop is decidable.*

**Checking termination of program lassos.** Due to the above result, we may henceforth assume that the lasso $S.L$ returned by the termination analyzer has a real non-terminating run in the CA. The lasso is mapped back into a sequence of program instructions $\xi(S).\xi(L)$ forming a program lasso. Two cases may arise: either the lasso is real on the program, or it is spurious.

***Non-spurious program lassos.*** Since we do not consider dynamic allocation, the number of configurations that a program can reach from any input tree is finite. Consequently, if there is a tree $t_\omega$ from which the program will have an infinite run along a given lasso,

[7]To encode conditions of the form $x \leq c$ we add a new variable $z$, initially set to zero, with the condition $z' = z$ appended to each transition, and rewrite the original condition as $x - z \leq c$.

then we can discover it by an exhaustive enumeration of trees. Moreover, as shown in the innermost loop of the analysis procedure in Section 5.4, using the symbolic implementation of the program instructions over tree automata, we can handle the discovery of $t_\omega$ by evaluating the lasso symbolically for all trees of a certain (increasingly growing) height at the same time. As we work with finite sets of trees, we are bound to visit the same set twice after a finite number of iterations if there exists a non-terminating run along the lasso.

```
while ($x \neq root$) do
  leftRotate($x$);
  $x := x$.up;
  $y := y$.up;
  rightRotate($x$);
  $x := x$.up;
  $y := y$.up;
```

Figure 5.10: A possibly nonterminating program

Note that, unfortunately, we cannot replace the above approach of examining gradually trees of increasing size by a single symbolic run starting with $I_0$. The set $I_0$ may contain both trees for which the program loops and also infinitely many trees for which infinite looping is impossible, but at the same time, the number of times the loop of the lasso can be fired for the latter trees depends on their size (and hence is not fixed). Then, if we try to symbolically run the lasso starting from all input trees, we will never converge as in every iteration of the loop, the working set of trees will shrink by some trees for which a further iteration is not possible. This situation happens, e.g., in the program in Figure 5.10 where if we initially have $x == y$, the run will not stop (provided the node pointed by $x$ has a right successor). On the other hand, if $y$ points to a node above the node pointed by $x$, the program will stop, but after a number of steps depending on the initial distance of $y$ from the root.

***Spurious program lassos.*** We handle this case also by a symbolic iteration of a given program lasso $\sigma.\lambda$ starting with the initial set of trees. We compute iteratively the sets $post(\sigma.\lambda^k, I_0), k = 1, 2, \ldots$. In the case of lassos without destructive updates, this computation is shown to reach the empty set after a number of iterations that is bounded by a double exponential in the length of the lasso (cf. Section 5.7.1). In the case of lassos with destructive updates, we can guarantee termination of the iteration with the empty set provided there exists some CA $A_u$ (albeit unknown) keeping track of the particular tree measures we consider here (formalized via the functions $M_{rsc}$ and $\theta_{rsc}$ in Section 5.6.2) that simulates the given program and that terminates[8] (cf. Section 5.7.2). In the latter case, even though we cannot guarantee the discovery of $A_u$, we can at least ensure that the sequence $post(\sigma.\lambda^k, I_0)$, $k = 1, 2, \ldots$ terminates with the empty set. This gives us a basis

[8]We can relax this condition by saying that $A_u$ does not have any infinite run, not corresponding to a run of the program. For the sake of clarity, we have chosen the first stronger condition.

for refining the current ACFG such that we get rid of the spurious lasso encountered, and we can go on in the search for a CA showing the termination of the given program.

### 5.7.1 Deciding Spuriousness of Lassos without Destructive Updates

In this section, we show that the spuriousness problem for a given lasso in a program with trees is decidable, if the lasso does not contain destructive updating instructions, i.e., tree rotations. The argument for decidability is that if there exists a non-terminating run along the loop, then there exists also a non-terminating run starting from a tree of size bounded by a constant depending on the program. Thus, there exists a tree within this bound that will be visited infinitely many often.[9]

Given a loop without destructive updates, we first build an abstraction of it by replacing the $\mathtt{go}\{\mathtt{Left}|\mathtt{Right}|\mathtt{Up}\}\mathtt{Null}(x, y)$ instructions by $x = \mathtt{null}$, and by eliminating all instructions $\mathtt{changeData}(x, d_1, d_2)$ and the tests. Clearly, if the original loop has a non-terminating computation, then its abstraction will also have a non-terminating run starting with the same tree. Hence it is sufficient to reason about lassos composed only of $x = \mathtt{null}$, $x = y$ and $\mathtt{go}\{\mathtt{Left}|\mathtt{Right}|\mathtt{Up}\}\mathtt{NonNull}(x, y)$ instructions[10]. The loop is then encoded as an iterative linear transformation which, for each pointer variable $x \in PVar$, has a counter $p_x$ encoding the binary position of the pointer in the current tree using 0/1 as the left/right directions. Additionally, the most significant bit of the encoding is required to be always one, which allows for differentiating between, e.g., the position $001$ encoded by $9 = (1001)_2$, and $0001$ encoded by $17 = (10001)_2$. Null pointers are encoded by the value $0$. The program instructions are translated into counter operations as follows:

$$x = \mathtt{null} : p_x = 0 \qquad x = y : p_x = p_y \qquad \mathtt{goLeftNonNull}(x, y) : p_x = 2 \star p_y$$
$$\mathtt{goRightNonNull}(x, y) : p_x = 2 \star p_y + 1 \qquad \mathtt{goUpNonNull}(x, y) : p_x = \tfrac{1}{2} p_y$$

where $2\star$ and $\frac{1}{2}$ denote the integer functions of multiplication ($x \mapsto 2x$) and division ($x \mapsto x/2$). Assuming that we have $n$ pointer variables, each program instruction is modeled by a linear transformation of the form $\mathbf{p}' = A\mathbf{p} + B$ where $A$ is an $n \times n$ matrix with at most one non-null element, which is either $1, 2$ or $\frac{1}{2}$, and $B$ is an $n$-column vector with at most one $1$ and the rest $0$[11]. The composition of the instructions on the loop is also a linear transformation, except that $A$ has at most one non-null element on each line, which is either $\mathcal{I}$, or a composition of $2\star$'s and $\frac{1}{2}$'s.

Since $A$ has at most one non-null element on each line, one can extract an $m \times m$ matrix $A_0$ for some $m \leq n$ that has exactly one non-null element on each line and column. Our proof is based on the fact that there exists some constant $k$ bounded by $\mathcal{O}(3^m)$ such that ${A_0}^k$ is a diagonal matrix. Intuitively, this means that the position of each pointer at

[9]Since there is no dynamic allocation, all trees visited starting with a tree of size $k$ will also have size $k$. Hence each run of the program will either stop, or re-visit the same program configuration after a bounded number of steps.

[10]Notice that instructions of the form $\mathtt{go}\{\mathtt{Left}|\mathtt{Right}|\mathtt{Up}\}\mathtt{Err}(x, y)$ can never occur in a lasso that is supposed to have a non-terminating run.

[11]We interpret the matrix operations over the semiring of integer functions $\langle \mathbb{N} \to \mathbb{N}, +, \circ, 0, \mathcal{I} \rangle$ where $\circ$ is functional composition and $\mathcal{I}$ is the identity function.

step $i+k$ is given by a linear function of the position of the pointer at $i$. Then $A^i$ is an exponential function of $i$. As there is no dynamic allocation of nodes in the tree, the non-termination hypothesis implies that the positions of pointers have to stay in-between bounds. But this is only possible if the elements of the main diagonal of ${A_0}^k$ are either $\mathcal{I}$ or compositions of the same number of $2\star$ and $\frac{1}{2}$. Intuitively, this means that all pointers are confined to move inside bounded regions of the working tree.

**Theorem 5** *Let $P$ be a program over trees, $PVar$ and $Data$ be its sets of pointer variables and data elements, $\mathcal{C} = Data \times 2^{PVar}$, $I_0 \subseteq \mathcal{T}(\mathcal{C})$ be an initial set of trees, and $\sigma.\lambda$ be a lasso of $P$. Then, if $P$ has an infinite run along the path $\sigma.\lambda^\omega$ for some $t_0 \in I_0$, then there exists a tree $t_{b0} \in \mathcal{T}(\mathcal{C})$ of height bounded by $(\|PVar\|+1)\cdot\big(|\sigma| + |\lambda|\cdot 3^{\|PVar\|}\big)$ such that $P$, started with $t_{b0}$, has an infinite run along the same path.*

Decidability of spuriousness is an immediate consequence of this theorem. Also, there is a bound on the number of symbolic unfoldings of a spurious lasso starting with the initial set of trees.

**Corollary 1** *Let $P$ be a program over trees, $PVar$ and $Data$ its sets of pointer variables and data elements, $\mathcal{C} = Data \times 2^{PVar}$, and $I_0 \subseteq \mathcal{T}(\mathcal{C})$ an initial set of trees. Given a lasso $S.L$ in the CA $A_{rsc}(G)$ built from an ACFG $G$ of $P$, let $\sigma = \xi(S)$ and $\lambda = \xi(L)$. Then, if $\sigma.\lambda$ does not contain destructive updates, its spuriousness is decidable. Moreover, if the lasso is spurious, for all $k \geq |\lambda| \cdot \mathbf{max}(2, \|Data\|)^{2^{(\|PVar\|+1)\cdot(|\sigma|+|\lambda|\cdot 3^{\|PVar\|})}}$, we have $post(\sigma.\lambda^k, I_0) = \emptyset$.*

Despite the double exponential bound, experimental evidence (see Section 5.9) shows that the number of unfoldings necessary to eliminate a spurious lasso is fairly small.

### 5.7.2 Checking Spuriousness of Lassos with Destructive Updates

We now explore the case when we are given a lasso in a CA $A = \langle X, Q, q_0, \varphi_0, \rightarrow\rangle$ built from an ACFG $G = \langle Instr, L, \alpha, l_0, \longmapsto\rangle$ of a program $P$, and the lasso contains counter operations corresponding to rotations. For this case, we do not give a general guarantee for discovering spuriousness of lassos in a finite number of steps. However, we can give such a guarantee at least for the case when we are dealing with a program for which there exists some (albeit unknown) CA $A_u$ simulating the program wrt. $\theta_{rsc}$ and $M_{rsc}$. That is, if there exists a termination argument for the program based on the tree measures we use, then we can prove spuriousness of the lasso by a symbolic iteration of the initial set. Note that in such a case, we still cannot guarantee we will find $A_u$ nor some other CA showing termination of $P$ (cf. Theorem 2), but we can at least get rid of particular spurious counterexamples and do not loop on the spot in our computation.

**Theorem 6** *Let $P$ be a program with an ACFG $G$ and let $S.L$ be a spurious lasso in $A_{rsc}(G)$. If there exists a CA $A_u$ that simulates $P$ wrt. $\theta_{rsc}$ and $M_{rsc}$ and that terminates on all inputs, then there exists $k \in \mathbb{N}$ such that $post(\xi(S).\xi(L)^k, I_0) = \emptyset$.*

*Proof*: Let $S = q_0^s \xrightarrow{\varphi_1^s} q_1^s \dots q_{m-1}^s \xrightarrow{\varphi_m^s} q_1^l$, $L = q_1^l \xrightarrow{\varphi_1^l} q_2^l \dots q_n^l \xrightarrow{\varphi_n^l} q_1^l$, $\sigma = \xi(S)$, and $\lambda = \xi(L)$.

We prove the theorem by contradiction. Suppose that for any $k \in \mathbb{N}$, $\sigma.\lambda^k$ is fireable for some $t \in I_0$, i.e., $post(\sigma.\lambda^k, I_0) \neq \emptyset$. We know that $\langle l_1, I_1\rangle \xrightarrow[A]{\varphi} \langle l_2, I_2\rangle$ iff $l_1 \xrightarrow[P]{\xi(\langle l_1,I_1\rangle,\varphi,\langle l_2,I_2\rangle)} l_2$, and so $\sigma.\lambda^k$ is a path in the CFG of $P$. Moreover, as $I_0$ is the set of initial trees for $P$, we have that $\sigma.\lambda^k$ is a run of $P$ for any $k \in \mathbb{N}$.

Next, as the CA $A_u$ simulates $P$ wrt. $\theta_{rsc}$, for any $k \in \mathbb{N}$, there must be a path $\theta_{rsc}(\sigma).\theta_{rsc}(\lambda)^k = \varphi_1^s...\varphi_m^s(\varphi_1^l...\varphi_n^l)^k$ in the control graph of $A_u$ over which the simulation of $\sigma.\lambda^k$ may be run. As the set of paths $\{\theta_{rsc}(\sigma).\theta_{rsc}(\lambda)^k \mid k \geq 0\}$ is infinite and the control of $A_u$ is finite, the control graph of $A_u$ must contain one (or several) lassos with a stem $\theta_{rsc}(\sigma') = \theta_{rsc}(\sigma.\lambda^{k_1})$ and a loop $\theta_{rsc}(\lambda') = \theta_{rsc}(\lambda^{k_2})$ (for $k_1 \geq 0, k_2 \geq 1$)—note that $\theta_{rsc}(\lambda)$ may be partly unfolded in such lassos. The existence of a lasso of the form $\theta_{rsc}(\sigma').\theta_{rsc}(\lambda')$ in $A_u$ then implies that there is an infinite path $\theta_{rsc}(\sigma).\theta_{rsc}(\lambda)^\omega = \varphi_1^s...\varphi_m^s(\varphi_1^l...\varphi_m^l)^\omega$ in $A_u$.

Further, the fact that $\theta_{rsc}(\sigma).\theta_{rsc}(\lambda)$ is a spurious lasso over $A$ implies that there is a valuation $\nu_0 : X \to \mathbb{Z}$ such that $\varphi_1^s...\varphi_m^s(\varphi_1^l...\varphi_m^l)^\omega$ is fireable in $A$ from $\nu_0$. Moreover, as the initial constraint $\varphi_0$ of $A$ is such that a valuation $\nu_0'$ satisfies $\varphi_0$ iff there exists $t' \in I_0$ such that $\forall x \in X : M_{rsc}(x, t') = \nu_0'(x)$, the initial constraint $\varphi_0^x$ of $A_u$ must be satisfiable by (at least) the same valuations as $\varphi_0$ in order to satisfy requirement (1) of Definition 1. Hence, $\nu_0 \models \varphi_0^x$ and $\varphi_1^s...\varphi_m^s(\varphi_1^l...\varphi_m^l)^\omega$ is fireable in $A_u$ from $\nu_0$ also. However, this is a contradiction as we know that $A_u$ terminates. □

Theorem 6 implies that if we execute the program along the lasso starting with all the input trees, we will reach the empty set in a finite number of steps (corresponding to the fact that the computation cannot go on beyond some constant number of steps for any of the input trees regardless of their size). In the innermost `while` loop in the procedure presented in Section 5.4, this computation is interleaved with checking whether the lasso is a real one. Under the assumptions of Theorem 6, we now know that the loop terminates.

## 5.8 Abstraction Refinement

**Exclusion of Spurious Lassos.** If we find a spurious lasso $S.L$ in a counter automaton $A_{rsc}(G) = \langle X, Q, q_0, \varphi_0, \to\rangle$ built from an ACFG $G = \langle Instr, LI, \langle l_0, I_0\rangle, \longmapsto\rangle$ of a program $P$, our goal is to refine the ACFG $G$ to $G_{S.L}$ such that the CA $A_{rsc}(G_{S.L})$ will not contain the lasso any more. For this, as we explain below, we can exploit the sets of trees generated within the symbolic execution used to show spuriousness of the lasso.

Let us have a spurious lasso with a stem $S = q_0^s \overset{\varphi_1^s}{\to} q_1^s \dots q_{m-1}^s \overset{\varphi_m^s}{\to} q_1^l$ and a loop $L = q_1^l \overset{\varphi_1^l}{\to} q_2^l \dots q_n^l \overset{\varphi_n^l}{\to} q_1^l$ in the CA $A_{rsc}(G)$. Due to the construction of $A_{rsc}(G)$, we can unambiguously map this lasso back to a lasso $q_0^s \overset{i_1^s}{\longmapsto} q_1^s \dots q_{m-1}^s \overset{i_m^s}{\longmapsto} q_1^l \overset{i_1^l}{\longmapsto} q_2^l \dots q_n^l \overset{i_n^l}{\longmapsto} q_1^l$ in $G$ where $i_1^s = \xi(q_0, \varphi_1^s, q_1)$, ..., $i_m^s = \xi(q_{m-1}, \varphi_m^s, q_1^l)$, $i_1^l = \xi(q_1^l, \varphi_1^l, q_2^l)$, ..., $i_n^l =$

$\xi(q^l_n, \varphi^l_n, q^l_1) \in Instr$ and $q^s_0 = \langle l^s_0, I^s_0 \rangle$, ..., $q^s_{m-1} = \langle l^s_{m-1}, I^s_{m-1} \rangle$, $q^l_1 = \langle l^l_1, I^l_1 \rangle$, ..., $q^l_n = \langle l^l_n, I^l_n \rangle \in LI$ ($l^s_0 = l_0$ and $I^s_0 = I_0$).

We suppose that the lasso $S.L$ was detected to be spurious via the methods described in Section 5.7.1 or 5.7.2. Hence we know that there is a $k \in \mathbb{N}$ such that the sequence of instructions $\sigma.\lambda^k$ where $\sigma = i^s_1 ... i^s_m$ and $\lambda = i^l_1 ... i^l_m$ is not fireable for any input tree $t \in I_0$. Let $I_0, I^s_1, I^s_2, ..., I^s_{m-1}, I^l_{11}, I^l_{21}, \ldots, I^l_{n1}, I^l_{12}, I^l_{22}, ..., I^l_{1k}, I^l_{2k}, ..., I^l_{nk}, \emptyset$ be the sequence of sets of configurations that one obtains when symbolically executing the sequence of instructions $\sigma.\lambda^k$ starting with $I_0$ and using the implementation of the instructions over tree automata that we have.

We can create the new ACFG $G_{S.L}$ from $G$ by splitting the sets of configurations that appear in the locations of $G$ corresponding to the lines through which the lasso $L.S$ goes (i.e., $l^s_0, ..., l^s_{m-1}, l^l_1, ..., l^l_n$), preserving all other locations, and recomputing the transitions according to the definition of ACFG. Note that we split all locations that correspond to the program lines that appear in the lasso $S.L$, not only the locations $q^s_0, ..., q^s_{m-1}, q^l_1, ..., q^l_n$ that appear directly in $S.L$. This allows us to get rid of all copies of the same lasso that could appear in $G$. The splitting of the locations is done in the following two steps starting with $LI_{S.L} = LI$:

1. For $i$ iterating from 1 to $m - 1$, we replace (split) each abstract control location $\langle l^s_i, I \rangle \in LI_{S.L}$ by $\langle l^s_i, I \cap I^s_i \rangle, \langle l^s_i, I \cap \overline{I^s_i} \rangle \in LI_{S.L}$. Note that some of the control locations can be split several times because the stem can go through the appropriate line more then once.

2. For $i$ iterating from 1 to $n$, we split each abstract control location $\langle l^l_i, I \rangle \in LI_{S.L}$ according to the sets of trees $I^l_{ij}$ as follows: $\langle l^l_i, I \rangle$ will be replaced by $k+1$ new abstract control locations $\langle l^l_i, I \cap [I^l_{i0}] \rangle, \langle l^l_i, I \cap [I^l_{i1}] \rangle, ..., \langle l^l_i, I \cap [I^l_{ik}] \rangle \in LI_{S.L}$ where

   - $[I^l_{i0}] = \overline{(I^l_{i1} \cup I^l_{i2} \cup ... \cup I^l_{ik})}$,
   - $[I^l_{ij}] = I^l_{ij} \cap \overline{(I^l_{i(j+1)} \cup ... \cup I^l_{ik})}$ for $1 \leq j < k$ (intuitively, these sets contain trees reachable at line $i$ with the same future in the loop $\lambda$ of the lasso), and
   - $[I^l_{ik}] = I^l_{ik}$.

   Note that $\forall p \neq q : [I^l_{ip}] \cap [I^l_{iq}] = \emptyset \wedge \bigcup_{j=0}^{j \leq k} [I^l_{ij}] = I$. Note also that some of the abstract control locations can be split several times because the loop can go through the appropriate line more then once.

**Theorem 7** *Suppose we have a spurious lasso with a stem $S$ labelled with counter operations $\varphi^s_1 ... \varphi^s_m$ and a loop L labelled with counter operations $\varphi^l_1 ... \varphi^l_n$ in a CA $A_{rsc}(G)$ built from an ACFG G of a program $P$. Then, the CA $A_{rsc}(G_{S.L})$ built from the refined ACFG $G_{S.L}$ simulates $P$ in the sense of Definition 1 wrt. $\theta_{rsc}$ and $M_{rsc}$, and it does not contain any lasso-shaped path with a stem labelled with $\varphi^s_1 ... \varphi^s_m (\varphi^l_1 ... \varphi^l_n)^p$ and a loop labelled with $(\varphi^l_1 ... \varphi^l_n)^q$ for any $p, q \geq 0$.*

*Proof*: The first claim follows immediately from the fact the we only split the sets of program configurations included in the particular abstract control locations (we do not loose any of the configurations), and so we can apply Lemma 1 and Theorem 3.

As for the second claim, we show below that $G_{S.L}$ does not contain a lasso-shaped path with a stem $\sigma.\lambda^p$ and a loop $\lambda$ for any $p \geq 0$. This is enough to prove that in $A_{rsc}(G_{S.L})$ there is no lasso with a stem $\varphi_1^s...\varphi_m^s(\varphi_1^l...\varphi_n^l)^p$ and a loop $\varphi_1^l...\varphi_n^l$ for any $p \geq 0$—otherwise, from some line of the program there would have to lead two different instructions (that must come from the same program statement) with the same counter operation assigned, which is not possible in our framework.

It remains to show that $G_{S.L}$ does really not contain a lasso-shaped path with a stem $\sigma.\lambda^p$ and a loop $\lambda$ for any $p \geq 0$. First, by induction on the length of the stem, it is easy to show that for any $\langle l, I\rangle \in LI_{S.L}$ reachable via a path labelled with $i_1^s...i_{m-1}^s$ from $\langle l_0, I_0\rangle$, $\langle l, I\rangle \subseteq I_{m-1}^s$. Moreover, $post(i_m^s, I_{m-1}^s) = I_{11}^l$, which implies that all paths labelled with $\sigma$ in $G_{S.L}$ lead to $\langle l_1^l, I'\rangle$ where $I' \subseteq [I_{1i}^l]$ for some $i \in \{1, ..., k\}$.

Next, for any $u \in \{1, ..., n\}$, $v \in \{1, ..., k\}$, and any $t \in [I_{uv}^l]$, we know that $t \notin I_{uq}$ for any $q > v$ and either (1) $post(i_u^l, t)$ is not defined, (2) $u < n \ \wedge \ post(i_u^l, t) \in I_{(u+1)v}^l$, or (3) $u = n \ \wedge \ post(i_u^l, t) \in I_{1(v+1)}^l$. Moreover, from the definition of $[I_{xy}^l]$, for $u < n$, we have $I_{(u+1)v}^l \ \cap \ \bigcup_{q=0}^{q<v}[I_{(u+1)q}^l] = \emptyset$ and $I_{(u+1)v}^l \subseteq \bigcup_{q=v}^{q\leq k}[I_{(u+1)q}^l]$. Similarly, if $u = n$, we have $I_{1(v+1)}^l \ \cap \ \bigcup_{q=0}^{q<v+1}[I_{1q}^l] = \emptyset$ and $I_{1(v+1)}^l \subseteq \bigcup_{q=v+1}^{q\leq k}[I_{1q'}^l]$. Together, we then get that either (1) $post(i_u^l, t)$ is not defined, (2) $u < n \ \wedge \ post(i_u^l, t) \in [I_{(u+1)q}^l]$ for some $q \in \{v, ..., k\}$, or (3) $u = n \ \wedge \ post(i_u^l, t) \in [I_{1q}^l]$ for some $q \in \{v+1, ..., k\}$.

However, the above means that for each transition $\langle l, I\rangle \underset{G_{S.L}}{\overset{i}{\longmapsto}} \langle l', I'\rangle$, we have $I \subseteq [I_{uv}^l]$ and $I' \subseteq [I_{u'v'}^l]$ such that $((u' > u \ \wedge \ v' \geq v\ ) \ \vee \ (u' = 1 \ \wedge \ v' > v))$. Consequently, as the number of the sets $[I_{xy}^l]$ is finite, the loop $\lambda$ of the lasso must be broken in $G_{S.L}$. □

**Beyond the Exclusion of Single Spurious Lassos.** One of the reasons why the above exclusion of lassos may not be sufficient by itself is that the concrete number of transitions in a spurious lasso may restrict the sets of trees reachable along the lasso to trees of a certain form and with *some measure on the tree having a certain concrete value* $x$. The lasso is then excluded for these trees, but subsequently a new, slightly longer one is encountered leading to splitting wrt. trees with the measure being $x+1$, then $x+2$, etc. For example, imagine that within the stem of a lasso we go through a loop `while` $(x.$`left` $\neq$ `null`$)$ $x = x.$`left` (we want to move $x$ to the left most node) and we in particular fire the statement $x = x.$`left` twice and then leave the `while` loop and go on in the stem. This restricts the analysis of the rest of the lasso to trees with the left-most branch of length 3, and all the splitting in the rest of the lasso will be with respect to such trees only. In the next lasso, the statement $x = x.$`left` can be fired 3 times, 4 times, etc. If we need to split wrt. trees of some form but with the given measure being unrestricted, we will never converge—note that a similar problem happens, e.g., also in predicate abstraction [LBBO01].

A heuristic solution we propose to the above problem is to *accelerate the splitting* in a suitable way. As shown in Section 5.9, we in particular successfully experimented with an *acceleration based on abstracting the sets* $[I_{ij}^l]$ used in the splitting via the finite-height abstraction from ARTMC (whose height may be iteratively adjusted in the verification loop). When using the abstraction, we split wrt. the original sets (and thus guarantee exclusion of the given spurious lasso), but also wrt. their abstraction, which according to our experience significantly increases chances of the technique to terminate (a proper theoretical examination of guarantees that can be provided for such a case is an interesting future work).

Another situation when the exclusion of spurious lassos will not itself suffice happens when the invariants computed by ARTMC are not precise enough. That is why in our verification loop, we have a certain pre-defined number that says how many times one tries to refine ACFGs and CA by splitting. If it is exhausted, one *re-runs ARTMC with a refined abstraction*—for this, we suggest to use the finite-height abstraction (possibly combined with the predicate-based one) with the abstraction height being gradually increased. Moreover, ARTMC is to be re-run over the the refined CFG of the last obtained ACFG so that the effect of the previous iterations of splitting is preserved, and the new sets of configurations computed by ARTMC are restricted to be subsets of those present in the abstract control locations of the last obtained ACFG.

## 5.9 Implementation and Experimental Results

To demonstrate the applicability of our approach, we tested its prototype implementation on several real procedures manipulating trees. In order to generate the needed line invariants and to check safety properties of the considered programs, we modified the ARTMC tool described in Sections 4.5 and 4.6. We restricted this tool to only binary trees with parent pointers and we replaced general purpose destructive updates supported by the tool by a special implementation of tree rotations. The procedures were fully automatically verified for absence of null/undefined pointer dereferences. In some cases, we also added shape testers (cf. Section 4.2.2) to check some further safety properties. The use of shape testers can also increase the precision of the line invariants, because one may need a more precise abstraction to check some additional shape properties. Using the generated invariants, we built ACFGs of the considered programs and the counter automata based on them. Then, we applied the ARMC tool [Ryb] to prove termination of these counter automata (and hence termination of the programs). We considered the following experiments (plus one addition mentioned in Section 5.9.1):

**Depth-first tree traversal.** This is the first considered procedure – i.e. our running example (cf. Figure 5.1(b)). The initial configuration was the set of all possible binary trees with parent pointers where all nodes have their data value set to *unmarked*. Within the ARTMC phase, we use a tester to check that at the end of the procedure, all nodes have the data value set to *marked*.

**Searching a data value in a red-black tree.** This is our second example. A red-black tree is a binary search tree where nodes are colored as red or black [CLR90]. The tree has the following properties:

- There are no two red successors in the tree.

- All paths from the root to leaves have an equal number of *black* nodes.
- The root of the tree is colored as *black*.
- All leaves are collored as *black*.

In this experiment, the set of input trees used was a regular overapproximation of the set of all possible red-black trees (the red-black property is not regular).

The procedure starts in the root of the tree and search, whether a given value is present in the tree. The actual data values were abstracted away and all the comparisons were done in a random way. In the ARTMC phase, we check just the basic memory consistency.

**Rebalancing red-black trees after inserting a new element** . This procedure presents our third experiment. After the insertion of a new node, the red-black balancedness properties are temporarily broken. Therefore a rebalancing based on tree rotations is performed [CLR90]. As in the previous case, the initial set was a regular overapproximation of the set of all possible red-black trees including the inserted node at some leaf position. Within this experiment, we did two sub-experiments. To obtain the line invariants overapproximating the sets of configurations reachable at each line, we first ran ARTMC while checking both the basic memory consistency as well as shape invariance (namely that there are no two *red* successors), and then we ran ARTMC while checking only the basic memory consistency. In the latter case, we obtained (in a much better running time) less precise line invariants, which were, however, still precise enough for the termination proof.

For these three examples, termination was shown without a need of refinement (unlike for the experiment we present later on). The results of the experiments performed on a 1.4 GHz Xeon with 1GB of memory are summarized in Table 5.2. The table contains the ARTMC running times ($Time_{ARTMC}$), the number of states of the largest invariant generated by ARTMC ($|Q|_{Inv}$), the time spent by the ARMC tool to show termination ($Time_{CA}$), and the number of counters ($N_{cnt}$)[12], locations ($N_{loc}$), and transitions ($N_{tr}$) of the CA. Taking into account the prototype nature of the tool (where, e.g., the ARTMC times could be much better if we were not building on top of the tool described in Section 4.5, which was designed for handling much more general data structures), we consider the obtained times encouraging.

Table 5.2: Results of experimenting with the presented method

| Example | $Time_{ARTMC}$ | $\|Q\|_{Inv}$ | $Time_{CA}$ | $N_{cnt}$ | $N_{loc}$ | $N_{tr}$ |
|---|---|---|---|---|---|---|
| Depth-first tree traversal | 43s | 67 | 10s | 5 | 15 | 20 |
| RB-search | 2s | 22 | 1s | 3 | 8 | 11 |
| RB-rebalance after insert | 1m 9s | 87 | 36s | 7 | 44 | 66 |

[12]We safely skipped the use of the subtree measures in our experiments.

```
bool $odd$ = false;
if ($list$ != null) then
  while (true) do
    $x = list$;
    while ($x$ != null) do
      $x$.data = $odd$;
      $odd$ = not($odd$);
      $x$ = $x$.next;
    if (not($odd$)) then break;
```

Figure 5.11: A procedure marking elements of a list as odd or even from the end of a list

### 5.9.1 Necessity of Refinement

To test our refinement procedure, we applied it on another case study where the initial invariants were not sufficient to prove termination. In particular, we considered the procedure in Figure 5.9.1 that marks the elements of a singly-linked list as even or odd, depending on whether their distance to the end of the list is even or odd. As the procedure does not know the length of the list and cannot use back-pointers, it tries to mark the first element as even, and at the end of the list, it checks whether the last element was marked as odd. If this is true, the marking is correct, otherwise the marking has to be reversed.

In the example, we slightly extend the syntax of our programs (defined in Section 5.3) by a support of boolean program variables with an operation $\texttt{not} : \texttt{bool} \rightarrow \texttt{bool}$ and a test $\texttt{if}\ (\texttt{bool})$ with the straightforward semantics. This does not influence the expressive power of the considered language as instead of a boolean variable, we could use a pointer variable $x$ with the semantics $(x\ ==\ \texttt{null}) \cong \texttt{false}$ and $(x\ !=\ \texttt{null}) \cong \texttt{true}$.

For this procedure, even if one builds the CA starting with the exact line invariants, termination cannot be established. To establish termination, one has to separate configurations where the procedure is marking the list in a correct way from those where the marking is incorrect. Then, the outer loop of the procedure will not appear in the CA at all since, in fact, it can be fired at most twice: once when the initial guess is correct and twice otherwise. The challenge is to recognise this fact automatically.

We managed to verify termination of the procedure on an arbitrary input list after excluding 9 spurious lassos (in 2 cases, the refinement was accelerated by the use of the finite-height abstraction on the $I_{ij}$ sets that resulted from splitting line invariants when excluding certain spurious lassos).

## 5.10 A Support for the Insert and Delete Statements

The approach we have presented so-far can be easily extended by program statements for attaching and detaching leafs. Let us represent the attachement by statements $\texttt{leafInsert}\{\texttt{Left}|\texttt{Right}\}(x)$, and the detachment of by the statement $\texttt{leafDelete}(x)$. First, we need to extend the concrete semantics presented in Figure 5.3 by the semantics of these statements, which is showed in Figure 5.12.

$$\frac{\begin{array}{cc} d \in \mathit{Data} & p \in \mathit{dom}(t) \\ x \in \nu(t(p)) & t(p.0) = \square \end{array}}{\langle l, t\rangle \rightarrow \langle l', t[p.0 \leftarrow \langle d, \emptyset\rangle, p.0.0 \leftarrow \square, p.0.1 \leftarrow \square]\rangle} \qquad \frac{\begin{array}{c} (\forall p \in \mathit{dom}(t) \,.\, x \notin \nu(t(p))) \vee \\ (p \in \mathit{dom}(t) \ \wedge\ x \in \nu(t(p)) \ \wedge\ t(p.0) \neq \square) \end{array}}{\langle l, t\rangle \rightarrow \mathit{Err}}\ \texttt{leafInsertLeft}(x)$$

$$\frac{\begin{array}{cc} d \in \mathit{Data} & p \in \mathit{dom}(t) \\ x \in \nu(t(p)) & t(p.1) = \square \end{array}}{\langle l, t\rangle \rightarrow \langle l', t[p.1 \leftarrow \langle d, \emptyset\rangle, p.1.0 \leftarrow \square, p.1.1 \leftarrow \square]\rangle} \qquad \frac{\begin{array}{c} (\forall p \in \mathit{dom}(t) \,.\, x \notin \nu(t(p))) \vee \\ (p \in \mathit{dom}(t) \ \wedge\ x \in \nu(t(p)) \ \wedge\ t(p.1) \neq \square) \end{array}}{\langle l, t\rangle \rightarrow \mathit{Err}}\ \texttt{leafInsertRight}(x)$$

$$\frac{\begin{array}{c} p \in \mathit{dom}(t) \quad x \in \nu(t(p)) \\ t(p.0) \ = \ t(p.1) \ = \ \square \end{array}}{\langle l, t\rangle \rightarrow \langle l', t[p \leftarrow \square, p.0 \leftarrow \bot, p.1 \leftarrow \bot]\rangle} \qquad \frac{\begin{array}{c} (\forall p \in \mathit{dom}(t) \,.\, x \notin \nu(t(p))) \vee \\ (p \in \mathit{dom}(t) \ \wedge\ x \in \nu(t(p)) \wedge \\ (t(p.0) \neq \square \ \vee\ t(p.1) \neq \square)) \end{array}}{\langle l, t\rangle \rightarrow \mathit{Err}}\ \texttt{leafDelete}(x)$$

Figure 5.12: The concrete semantics of leafInsert and leafDelete

Then we need to extend the preprocessing phase from Section 5.3.1. For each set of variables $A \subseteq PVar$ with $x \in A$ which are aliased with $x$ or point to a node in the path from the root to $x$, each statement of the form $\texttt{leafInsert}\{\texttt{Left}|\texttt{Right}\}(x)$ is split into instructions that we denote as $\texttt{leafInsert}\{\texttt{Left}|\texttt{Right}\}(x, A)$. The semantics of the instructions is the same as the one of the original statement with the precondition $A = \cup_{p' \preceq p} \nu(t(p'))$. Intuitively, the subtree-size counters corresponding to all variables in $A$ increase after adding the new element. The statement $\texttt{leafInsert}\{\texttt{Left}|\texttt{Right}\}\texttt{Err}(x)$ corresponds to cases where (i) the position for the insertion is not defined:
$\forall p \in \mathit{dom}(t) \,.\, x \notin \nu(t(p)))$, or (ii) the position for the insertion is occupied: $p \in \mathit{dom}(t) \wedge x \in \nu(t(p)) \wedge t(p.0) \neq \square$ for $\texttt{leafInsertLeftErr}(x)$, and $p \in \mathit{dom}(t) \wedge x \in \nu(t(p)) \ \wedge\ t(p.1) \neq \square$ for $\texttt{leafInsertRightErr}(x)$ respectively.

Similarly, for each choice of disjoint subsets $X, A \subseteq PVar$ with $x \in X$, we split each statement $\texttt{leafDelete}(x)$ into instructions $\texttt{leafDelete}(x, X, A)$. The semantics of the instructions is the same as the one of the original statement with the preconditions $X = \nu(t(p)) \wedge x \in X$ and $A = \cup_{p' \prec p} \nu(t(p'))$. The variables in $X$ become null whereas the subtree sizes for $A$ decrease. The instruction $\texttt{leafDeleteErr}(x)$ covers the cases where (i) the position for the deletion is not defined: $\forall p \in \mathit{dom}(t) \,.\, x \notin \nu(t(p)))$, or (ii) the position to delete is not a leaf: $p \in \mathit{dom}(t) \ \wedge\ x \in \nu(t(p)) \ \wedge\ (t(p.0) \neq \square \ \vee\ t(p.1) \neq \square)$.

Now, we can extend the function $\theta_{rsc}$ from Table 5.1 by introducing counter operations for the newly added instructions as is showed in Table 5.3. Note that validity of Theorem 3 is preserved, and therefore the constructed CA simulates the original program. Hence CA termination tools like [Ryb] can be applied. If one proves the generated CA to terminate, this will imply a termination of the original program. When the termination check fails, we obtain a lasso-shaped counter example as in the case without the `leafInsert` and `leafDelete` instructions.

Table 5.3: The extension of $\theta_{rsc}$ to leafInsert and leafDelete

| instruction $i$ | counter manipulation $\theta_{rsc}(i)$ |
|---|---|
| `leafInsert`$\{$`Left`$\|$`Right`$\}(x, A)$ | $gLeafInsert(x, A) \wedge aLeafInsert(x, A)$ |
| `leafDelete`$(x, X, A)$ | $gLeafDelete(x, X, A) \wedge aLeafDelete(x, X, A)$ |

$$\begin{array}{ll}
gLeafInsertLeft(x, A) = & aLeafInsertLeft(x, A) = \\
\quad r_x \geq 0 \wedge & \\
\quad (\forall v \in A : r_v \leq r_x \ \wedge \ s_v \geq s_x) & \quad (\forall v \in A : s'_v = s_v + 1) \\
gLeafDelete(x, X, A) = & aLeafDelete(x, X, A) = \\
\quad r_x \geq 0 \ \wedge \ s_x = 0 \wedge & \\
\quad (\forall v \in X : r_v = r_x \ \wedge \ s_v = s_x) \wedge & \quad (\forall v \in X : r'_v = s'_v = -1) \wedge \\
\quad (\forall v \in A : r_v < r_x \ \wedge \ s_v > s_x) & \quad (\forall v \in A : s'_v = s_v - 1)
\end{array}$$

**Checking spurious lassos**

The counter operations associated to `leafInsert` and `leafDelete` can be easily transformed to the conjunctions of formulae of the form $x - y \leq c$, $x' - y \leq c$, $x - y' \leq c$, or $x' - y' \leq c$ where $x'$ denotes the future value of the counter $x$ and $c \in \mathbb{Z}$ is an arbitrary integer constant. Therefore the termination of a given control loop in CA is decidable as was showed in Section 5.7 and so we may assume that the lasso is real in the given CA. We transform the CA lasso back to the program statements and we have the following two possibilities:

- The loop of the program lasso does not contain destructive updates (rotations, insertions, deletes).
- The loop of the program lasso contains destructive updates.

In the first case, we can safely use the methods proposed in Section 5.7.1 to decide whether a given program lasso is a spurious or a real one. In the second case, we can decide the spuriousness of the program lasso only in case when there exists an (unknown) CA simulating the program by means of our measures (see Theorem 6). This theorem can be applied also in cases with the `leafInsert` and `leafDelete` instructions. In case of a spurious program lasso, the refinement can be done in the same fashion, as is described in Section 5.8.

## 5.11 Summary of Termination for Tree Manipulating Programs

In this Chapter we have presented a new automatic approach which allows us to check termination on programs manipulating trees with parrent pointers (lists and DLLs are special cases of these trees) in the considered operating suffice. We consider binary trees with nodes carrying data values from a finite domain. The only considered destructive updates on these trees are tree rotations which allow us to cover a huge family of real procedures manipulating (balanced) binary search trees. The basic idea is to run ARTMC (cf. Chapter 4) to check safety properties and generate invariants for all locations in the control flow graph of the given program. The invariants are then used to restrict the transitions of the control flow graph just to the feasible ones. Then, a counter automaton simulating the original program is created as an abstraction of this annotated control flow graph. This allows us to use some termination checker designed for counter system to prove the termination. The fact that the counter automaton simulates the program means that its termination implies termination of the original program.

In case a nonterminating run in the generated counter automaton is reported by a termination checker, the termination checker outputs a lasso-shaped counter example[13]. We need to check whether the given lasso is a real or spurious counter-example (caused by the abstraction used) in the original program. We showed that this is decidable in case of lassos without destructive updates. In case of lasso with destructive updates, we can just guarantee that we will discover spuriousness of a given lasso in the cases where there exists another counter automaton (unknown for us) simulating the program via the measures defined by our approach (i.e. the termination proof is possible within our verification framework). For an arbitrary spurious program lasso, we provide an automatic way to refine the abstraction. The refinement is based on splitting of the locations in the annotated control flow graph.

The method is not guaranteed to terminate due to the fact that the termination problem for our class of programs is not recursively enumerable. In spite of this, the method was proved to be useful on several examples of tree manipulating programs.

There is still a lot of work in the area of automatic termination checking for programs with dynamic data structures. The existing methods are able to handle just very limited data structures or their efficiency is not completely satisfying (also for very restricted classes of programs). The future research has to go in the following directions: (i) to generalize the existing methods or provide new ones for more complex data structures, (ii) to increase the efficiency of the existing methods and (iii) to provide some guaranties of termination at least for some restricted classes of programs and data structures.

[13]The problem whether a given lasso-shaped conterexample is a real counterexample in the CA is decidable for the class of CA generated by our approach. In case that the reported lasso-shaped conter-example is spurious in the CA, we suggest to use another tool.

# Chapter 6

# Conclusions and Future Directions

In this thesis, we have concentrated on approaches of formal verification of infinite state-space systems based on regular tree model checking—a symbolic verification technique based on the theory of regular tree languages. Our contribution can be basically divided into three parts. Each of these parts has been presented in a separate chapter and has its own summary with a discusion of possible extensions. Let us now just shortly summarize all the results of the thesis as well as future research directions in the given area.

**Summary**

The first part of our contribution has been a generalization of abstract regular model checking into the domain of systems whose states may be viewed as trees. The technique is a combination of regular tree model checking with automated abstraction based on the CEGAR loop. Instead of the precise set of reachable configurations, an overapproximation is computed, and the abstraction is used to accelerate the computation. The used abstraction is automatically refined in cases where the computed overapproximation is to rough to prove the system correct according to the given specification. We call this technique *abstract regular tree model checking* (ARTMC).

In general, the problem considered in regular tree model checking is undecidable, and therefore our method is not guaranteed to terminate. But our experiments with a set of examples from the area of parametric tree networks of processes have showed that the method is quite useful.

In the next part, we have proposed a general method for verification of safety properties of pointer manipulating procedures build on top of the ARTMC framework. We have proposed an original encoding of shape graphs (which are in general unrestricted graphs) into trees using special *extra edges* defined as regular expressions over the directions in trees. A set of shape graphs encoded as trees is then captured by a tree automaton, and pointer manipulating procedures are encoded as tree transducers. The encoding into tree automata and transducers allows us to apply ARTMC. We have implemented the method in a prototype tool based on the Mona tree automata library and we have applied it on a set of pointer manipulating procedures, where some of them have been—for the first time—verified by a fully automated method.

As last part of our contribution presented in this thesis, we have concentrated on automated creation of termination proofs for programs working with tree-shaped dynamic data structures. The only allowed destructive updates are tree rotations. We use ARTMC to check safety properties and to generate invariants for all control locations of a program. Then we use these invariants to construct a counter automaton simulating the original program. This construction allows us to subsequently use some model checker for counter systems. The positive answer of the counter automata model checker implies termination of the original program. In case of a negative answer, we check whether it is caused by a non-terminating run of the program, or due to the construction of the counter automaton. In the second case, we refine the counter automaton and continue the computation. We have ilustrated capabilities of the method on several procedures manipulating trees. To the best of our knowledge, we are not aware of any other generic and fully automated method for termination proofs of the considered type of programs.

## Future Research Directions

Future research in the areas related to the thesis may be divided to several categories.

Research on possibilities of how to increase the existing techniques forms the first category of possible future works. It includes research on possible applications using a more efficient underlaying library for manipulating tree automata. The library could use a more advanced BDD technique than is used in the Mona library on which our prototype is based on. Sharing of BDD nodes should be applied not only within particular automata, but across all the automata that appear in a verification run. Further an interesting idea is to use nondeterministic automata instead of deterministic ones. This allows one to avoid the expensive determinization step as was proposed in [WDHR06]. Moreover, even within Mona, a significant improvement could be achieved by using the concept of *guided tree automata* [BKR97] proved to be capable of bringing significant improvements by the authors of Mona [KM01]. The efficiency of the ARTMC framework could probably be improved by using special-purpose abstraction methods designed to concrete applications. Special-purpose abstractions were proved useful in the case of abstract regular word model checking [BHMV05], so there is a hope that it can help also in the case of trees.

The second category is a specialization of the existing techniques. The goal of this research direction should be to provide a verification technique capable of handling more restricted classes of data structures, but with a better performance, with some termination guarantees, or with a possibility to prove some more complex properties. A similar approach has, indeed, been taken within the last technique that we presented in this thesis, where we use ARTMC for a restricted class of pointer-manipulating procedures in order to prove termination. The restricted classes can also allow us to use some more specialized abstraction methods within ARTMC.

Finally, another possible direction is a development of new methods which will be capable to handle more complex data structures and/or more complex properties. For these purposes, one could try, e.g., to generalize ARTMC into some more general class of automata designed for more complex structures then trees or use some automata with constrains [HIV06]. Another possibility is to develop some more efficient encodings of some types of data structures into existing frameworks. Different verification frameworks can also be combined together (e.g. ARTMC with some technique based on logics, etc.). Such a combination could bring a better performance or allow us to verify some more complex properties.

# Bibliography

[ABH$^+$97] R. Alur, R. Brayton, T. Henzinger, S. Qadeer, and S. Rajamani. Partial-Order Reduction in Symbolic State Space Exploration. In *Proc. of CAV'97*, volume 1254 of *LNCS*. Springer, 1997.

[ACD93] R. Alur, C. Courcoubetis, and D.L. Dill. Model-Checking in Dense Real-Time. *Information and Computation*, 104:2–34, 1993. First appeared in Proc. of LICS'90.

[AD94] R. Alur and D.L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126:183–235, 1994. First appeared in Proc. of ICALP'90.

[AdJN02] P.A. Abdulla, J. d'Orso, B. Jonsson, and M. Nilsson. Regular Model Checking Made Simple and Efficient. In *Proc. of CONCUR'02*, volume 2421 of *LNCS*. Springer, 2002.

[AdJN03] P.A. Abdulla, J. d'Orso, B. Jonsson, and M. Nilsson. Algorithmic Improvements in Regular Model Checking. In *Proc. of CAV'03*, volume 2725 of *LNCS*. Springer, 2003.

[AJ01] P. Abdulla and B. Jonsson. Verifying Programs with Unreliable Channels. *Information and Computation*, 127(2):91–101, 2001.

[AJMd02] P.A. Abdulla, B. Jonsson, P. Mahata, and J. d'Orso. Regular Tree Model Checking. In *Proc. of CAV'02*, volume 2404 of *LNCS*. Springer, 2002.

[ALdA05] P.A. Abdulla, A. Legay, J. d'Orso, and A.Rezine. Simulation-Based Iteration of Tree Transducers. In *Proc. of TACAS'05*, volume 3440 of *LNCS*. Springer, 2005.

[AMN05] R. Alur, P. Madhusudan, and W. Nam. Symbolic Compositional Verification by Learning Assumptions. In *Proc. of CAV'05*, volume 3576 of *LNCS*. Springer, 2005.

[Ang87] D. Angluin. Learning Regular Sets from Queries and Counterexamples. *Information and Computation*, 75(2):87–106, 1987.

[BAN04] S. Bardin, A.Finkel, and D. Nowak. Toward Symbolic Verification of Programs Handling Pointers. In *Proc. of AVIS'04*, 2004.

[BBH$^+$06] A. Bouajjani, M. Bozga, P. Habermehl, R. Iosif, P. Moro, and T. Vojnar. Programs with Lists are Counter Automata. In *Proc. of CAV'06*, volume 4144 of *LNCS*. Springer, 2006.

[BBLS00] K. Baukus, S. Bensalem, Y. Lakhnech, and K. Stahl. Abstracting WS1S Systems to Verify Prameterized Networks. In *Proc. of TACAS 2000*, volume 1785 of *LNCS*. Springer, 2000.

[BCC$^+$07a] Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, Peter O'Hearn, Thomas Wies, and Hongseok Yang. Shape analysis for composite data structures. Technical Report TR-2007-13, Microsoft research, 2007.

[BCC$^+$07b] Josh Berdine, Aziem Chawdhary, Byron Cook, Dino Distefano, and Peter O'Hearn. Variance analyses from invariance analyses. In *Proc. of POPL'07*. ACM Press, 2007.

[BCE$^+$05] P. Baldan, A. Corradini, J. Esparza, T. Heindel, B. König, and V. Kozioura. Verifying Red-Black Trees. In *Proc. of COSMICAH'05*, Technical report RR-05-04. Queen Mary, University of London, 2005.

[BCK01] P. Baldan, A. Corradini, and B. König. A Static Analysis Technique for Graph Transformation Systems. In *Proc. of CONCUR'01*, volume 2154 of *LNCS*. Springer, 2001.

[BCM$^+$92] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: $10^{20}$ States and Beyond. *Information and Computation*, 98(2):142–170, 1992.

[BFL06] S. Bardin, A. Finkel, and E. Lozes. From Pointer Systems to Counter Systems Using Shape Analysis. In *Proc. of AVIS'06*, 2006.

[BHMV05] A. Bouajjani, P. Habermehl, P. Moro, and T. Vojnar. Verifying Programs with Dynamic 1-Selector-Linked Structures in Regular Model Checking. In *Proc. of TACAS'05*, volume 3440 of *LNCS*. Springer, 2005.

[BHRV05] A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. Abstract Regular Tree Model Checking. In *Proc. of Infinity'05*, 2005.

[BHRV06a] A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. Abstract Regular Tree Model Checking. *ENTCS*, 149:37–48, 2006. A preliminary version was presented at Infinity'05.

[BHRV06b] A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. Abstract Regular Tree Model Checking of Complex Dynamic Data Structures. In *Proc. of SAS'06*, volume 4134 of *LNCS*. Springer, 2006.

[BHT05] D. Beyer, T.A. Henzinger, and G. Théoduloz. Lazy Shape Analysis. Technical Report HMTC-REPORT-2005-006, EPFL, Lausanne, Switzerland, 2005.

[BHV04] A. Bouajjani, P. Habermehl, and T. Vojnar. Abstract Regular Model Checking. In *Proc. of CAV'04*, volume 3114 of *LNCS*. Springer, 2004.

[BI05] Marius Bozga and Radu Iosif. Quantitative verification of programs with lists. Technical Report TR-2005-2, Verimag, Centre Équation, 38610 Gières, Jan 2005.

[BIL03] M. Bozga, R. Iosif, and Y. Lakhnech. Storeless Semantics and Alias Logic. In *Proc. of PEPM'03*. ACM Press, 2003.

[BIL06] M. Bozga, R. Iosif, and Y. Lakhnech. Flat Parametric Counter Automata. In *Proc. of ICALP'06*, volume 4052 of *LNCS*. Springer, 2006.

[BJNT00] A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Regular Model Checking. In *Proc. of CAV'00*, volume 1855 of *LNCS*. Springer, 2000.

[BKMW01] A. Bruggemann-Klein, M. Murata, and D. Wood. Regular Tree and Regular Hedge Languages over Unranked Alphabets: Version 1. Technical Report HKUST-TCSC-2001-0, The Hongkong University of Science and Technology, 2001.

[BKR97] M. Biehl, N. Klarlund, and T. Rauhe. Algorithms for Guided Tree Automata. In *Proc. of WIA'96*, volume 1260 of *LNCS*. Springer, 1997.

[BLBS01] K. Baukus, Y. Lakhnech, S. Bensalem, and K. Stahl. Networks of Processes with Parameterized State Space. In *Proc. of VEPAS'01*, volume 50 of *ENTCS*. Elsevier, 2001.

[BLO98] S. Bensalem, Y. Lakhnech, and S. Owre. Computing Abstractions of Infinite State Systems Compositionally and Automatically. In *Proc. of CAV'98*, LNCS. Springer, 1998.

[BLW03] B. Boigelot, A. Legay, and P. Wolper. Iterating Transducers in the Large. In *Proc. of CAV'03*, volume 2725 of *LNCS*. Springer, 2003.

[BLW04] B. Boigelot, A. Legay, and P. Wolper. Omega-Regular Model Checking. In *Proc. of TACAS'04*, volume 2988 of *LNCS*. Springer, 2004.

[BnHS05] A.R. Bradley and Z. Manna nad H.B. Sipma. The Polyranking Principle. In *Proc. of ICALP'05*, volume 3580 of *LNCS*. Springer, 2005.

[BPZ05] I. Balaban, A. Pnueli, and L.D. Zuck. Shape Analysis by Predicate Abstraction. In *Proc. of VMCAI'05*, volume 3385 of *LNCS*. Springer, 2005.

[BPZ07] Ittai Balaban, Amir Pnueli, and Lenore D. Zuck. Shape analysis of single-parent heaps. In *Proc. of VMCAI'07*, LNCS. Springer, 2007.

[BR01] T. Ball and S. K. Rajamani. The SLAM Toolkit. In *Proc. of CAV'01*, LNCS. Springer, 2001.

[Bry86] R.E. Bryant. Graph-based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[BS97] O. Burkart and B. Steffen. Model Checking the Full Modal mu-Calculus for Infinite Sequential Processes. In *Proc. of ICALP'97*, volume 1256 of *LNCS*. Springer, 1997.

[BT02] A. Bouajjani and T. Touili. Extrapolating Tree Transformations. In *Proc. of CAV'02*, volume 2404 of *LNCS*. Springer, 2002.

[CC77] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. of POPL'77*. ACM Press, 1977.

[CC79] P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In *Proc. of POPL'79*. ACM Press, 1979.

[CCG$^+$04] S. Chaki, E. Clarke, A. Groce, J. Ouaknine, O. Strichman, and K. Yorav. Efficient Verification of Sequential and Concurrent C Programs. *Formal Methods in System Design*, 25(2-3):129–166, 2004.

[CCST05] S. Chaki, E. Clarke, N. Sinha, and P. Thati. Automated Assume-Guarantee Reasoning for Simulation Conformance. In *Proc. of CAV'05*, volume 3576 of *LNCS*. Springer, 2005.

[CDG$^+$05] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree Automata Techniques and Applications, 2005. URL: `http://www.grappa.univ-lille3.fr/tata`.

[CDOY06] C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Beyond Reachability: Shape Abstraction in the Presence of Pointer Arithmetic. In *Proc. of SAS'06*, volume 4134 of *LNCS*. Springer, 2006.

[CE81] E.M. Clarke and E.A. Emerson. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In *Logic of Programs, Workshop*, volume 131 of *LNCS*. Springer, 1981.

[ČEV06] M. Češka, P. Erlebach, and T. Vojnar. Pattern-Based Verification of Programs with Extended Linear Linked Data Structures. *ENTCS*, 145:113–130, 2006. A preliminary version was presented at AVOCS'05.

[ČEV07a] M. Češka, P. Erlebach, and T. Vojnar. Generalised multi-pattern-based verification of programs with linear linked structures. *Formal Aspects of Computing*, 2007(19):12, 2007.

[ČEV07b] M. Češka, P. Erlebach, and T. Vojnar. Pattern-based verification for trees. In *Computer Aided Systems Theory*, pages 181–182, 2007.

[CFI96] G. Cécé, A. Finkel, and S.P. Iyer. Unreliable Channels Are Easier to Verify Than Perfect Channels. *Information and Computation*, 141(1), 1996.

[CFJ93] E.M. Clarke, T. Filkorn, and S. Jha. Exploiting Symmetry In Temporal Logic Model Checking. In *Proc. of CAV'93*, volume 697 of *LNCS*. Springer, 1993.

[CGJ$^+$00] E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement. In *Proc. of CAV 2000*, volume 1855 of *LNCS*. Springer, 2000.

[CGL94] E.M. Clarke, O. Grumberg, and D.E. Long. Model Checking and Abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994.

[CGP99] E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[CJ98] H. Comon and Y. Jurski. Multiple Counters Automata, Safety Analysis and Presburger Arithmetic. In *Proc. of CAV'98*, volume 1427 of *LNCS*. Springer, 1998.

[CLM89] E.M. Clarke, D. Long, and K.L. McMillan. Compositional Model Checking. In *Proc. of LICS'89*. IEEE Press, 1989.

[CLR90] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

[Cou81] P. Cousot. Semantic Foundations of Program Analysis. In *Program Flow Analysis: Theory and Applications*. Prentice-Hall, 1981.

[CPR05] B. Cook, A. Podelski, and A. Rybalchenko. Abstraction Refinement for Termination. In *Proc. of SAS'05*, volume 3672 of *LNCS*, 2005.

[CPR06] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Terminator: Beyond Safety. In *Proc. of CAV 2006*, volume 4144 of *LNCS*. Springer, 2006.

[CR00] S.J. Creese and A.W. Roscoe. Data Independent Induction over Structured Networks. In *Proc. of PDPTA 2000*. CSREA Press, 2000.

[CR07] Sigmund Cherem and Radu Rugina. Maintaining Doubly-Linked List Invariants in Shape Analysis with Local Reasoning. In *Proc. of VMCAI'07*, volume 4349 of *LNCS*. Springer, 2007.

[DBCO06] D. Distefano, Josh Berdine, Byron Cook, and P.W. O'Hearn. Automatic Termination Proofs for Programs with Shape-shifting Heaps. In *Proc. of CAV'06*, volume 4144 of *LNCS*. Springer, 2006.

[DEG06] J.V. Deshmukh, E.A. Emerson, and P. Gupta. Automatic Verification of Parameterized Data Structures. In *Proc. of TACAS'06*, volume 3920 of *LNCS*. Springer, 2006.

[Deu92] A. Deutsch. *Operational Models of Programming Languages and Representations of Relations on Regular Languages with Application to the Static Determination of Dynamic Aliasing Properties of Data*. PhD thesis, University Paris VI, Paris, France, 1992.

[Deu94] A. Deutsch. Interprocedural May-Alias Analysis for Pointers: Beyond $k$-Limiting. In *Proc. of PLDI'94*. ACM Press, 1994.

[Dil89] D.L. Dill. Timing Assumptions and Verification of Finite-state Concurrent Systems. In *Proc. of Int. Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*. Springer, 1989.

[DLS01] D. Dams, Y. Lakhnech, and M. Steffen. Iterating Transducers. In *Proc. of CAV'01*, volume 2102 of *LNCS*. Springer, 2001.

[DN03] D. Dams and K.S. Namjoshi. Shape Analysis through Predicate Abstraction and Model Checking. In *Proc. of VMCAI'03*, volume 2575 of *LNCS*. Springer, 2003.

[DOY06] D. Distefano, P.W. O'Hearn, and H. Yang. A Local Shape Analysis Based on Separation Logic. In *Proc. of TACAS'06*, volume 3920 of *LNCS*. Springer, 2006.

[EH86] E.A. Emerson and J.Y. Halpern. 'Sometimes' and 'Not Never' Revisited: On Branching Versus Linear Time Temporal Logic. *Journal of the ACM*, 33(1):151–178, 1986.

[EM04] D. Engler and M. Musuvathi. Static Analysis versus Software Model Checking for Bug Finding. In *Proc. of VMCAI'04*, volume 2937 of *LNCS*. Springer, 2004.

[EN96] E.A. Emerson and K.S. Namjoshi. Automatic Verification of Parameterized Synchronous Systems. In *Proc. of CAV'96*, volume 1102 of *LNCS*. Springer, 1996.

[Eng75] J. Engelfriet. Bottom-up and Top-down Tree Transformations—A Comparison. *Mathematical System Theory*, 9:198–231, 1975.

[Esp02] J. Esparza. Grammars as Processes. In *Formal and Natural Computing*, volume 2300 of *LNCS*. Springer, 2002.

[EV05] P. Erlebach and T. Vojnar. Automated Formal Verification of Programs with Dynamic Data Structures Using State-of-the-Art Tools. In *Proc. of ISIM'05*. MARQ Ostrava, 2005.

[Fin94] A. Finkel. Decidability of the Termination Problem for Completely Specified Protocols. *Distributed Computing*, 7(3):129–135, 1994.

[FO97] L. Fribourg and H. Olsen. Reachability Sets of Parametrized Rings as Regular Languages. *ENTCS*, 9, 1997. A preliminary version was presented at Infinity'97.

[FP01] D. Fisman and A. Pnueli. Beyond Regular Model Checking. In *Proc. of FSTTCS'01*, volume 2245 of *LNCS*. Springer, 2001.

[Gen05] T. Genet. Timbuk: A Tree Automata Library, 2005. URL: `http://www.irisa.fr/lande/genet/timbuk`.

[God91] P. Godefroid. Using Partial Orders for the Efficient Verification of Deadlock Freedom and Safety Properties. In *Proc. of CAV'91*, volume 575 of *LNCS*. Springer, 1991.

[GS92] S.M. German and A.P. Sistla. Reasoning about Systems with Many Processes. *Journal of the ACM*, 39(3):675–735, 1992.

[GS97] S. Graf and H. Saidi. Construction of Abstract State Graphs with PVS. In *Proc. of CAV'97*, volume 1254 of *LNCS*. Springer, 1997.

[HIRV07a] P. Habermehl, R. Iosif, A. Rogalewicz, and T. Vojnar. Proving Termination of Tree Manipulating Programs. In *Proc. of ATVA'07 (to appear)*. Springer, 2007.

[HIRV07b] P. Habermehl, R. Iosif, A. Rogalewicz, and T. Vojnar. Proving Termination of Tree Manipulating Programs. Technical Report TR-2007-1, Verimag, 2007.

[HIV06] P. Habermehl, R. Iosif, and T. Vojnar. Automata-Based Verification of Programs with Tree Updates. In *Proc. of TACAS'06*, volume 3920 of *LNCS*. Springer, 2006.

[HJMS02] T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy Abstraction. In *Proc. of POPL'02*. ACM Press, 2002.

[HJMS03] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software Verification with Blast. In *Proc. of 10th SPIN Workshop*, volume 2648 of *LNCS*. Springer, 2003.

[HNSY94] T.A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic Model Checking for Real-time Systems. *Information and Computation*, 111:193–244, 1994. First appeared in Proc. of LICS'92.

[HV04] P. Habermehl and T. Vojnar. Regular Model Checking Using Inference of Regular Languages. In *Proc. of Infinity'04*, 2004.

[HV05] P. Habermehl and T. Vojnar. Regular Model Checking Using Inference of Regular Languages. *ENTCS*, 138:21–36, 2005. A preliminary version was presented at Infinity'04.

[ID96] C.N. Ip and D.L. Dill. Better Verification Through Symmetry. *Journal of Formal Methods in System Design*, 9(1/2):41–76, 1996.

[IMO] International mathematical olympiad 1976.
URL: `http://www.kalva.demon.co.uk/imo/imo76.html`.

[JN00] B. Jonsson and M. Nilsson. Transitive Closures of Regular Relations for Verifying Infinite-State Systems. In *Proc. of TACAS'00*, volume 1785 of *LNCS*. Springer, 2000.

[Jon81] H.B.M. Jonkers. Abstract Storage Structures. In *Algorithmic Languages*. IFIP, 1981.

[KK06] B. König and V. Kozioura. Counterexample-Guided Abstraction Refinement for the Analysis of Graph Transformation Systems. In *Proc. of TACAS'06*, volume 3920 of *LNCS*. Springer, 2006.

[Kle87] S. Kleene. *Introduction to Mathematics*. North-Holland, 1987.

[KM95] R.P. Kurshan and K.L. McMillan. A Structural Induction Theorem for Processes. *Information and Computation*, 117(1), 1995.

[KM01] N. Klarlund and A. Møller. MONA Version 1.4 User Manual, 2001. BRICS, Department of Computer Science, University of Aarhus, Denmark.

[KMe00a] M. Kaufmann, P. Manolios, and J. Strother Moore (eds.). *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Publishers, 2000.

[KMe00b] M. Kaufmann, P. Manolios, and J. Strother Moore (eds.). *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.

[KMM$^+$01] Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic Model Checking with Rich Assertional Languages. *Theoretical Computer Science*, 256(1–2), 2001.

[Koz83] D. Kozen. Results on the Propositional $\mu$-Calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[KP88] S. Katz and D. Peled. An Efficient Verification Method for Parallel and Distributed Programs. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, Workshop*, volume 354 of *LNCS*. Springer, 1988.

[KS93] N. Klarlund and M.I. Schwartzbach. Graph Types. In *Proc. of POPL'93*. ACM Press, 1993.

[KvS04] M. Křetínský, V. Řehák, and J. Strejček. Extended Process Rewrite Systems: Expressiveness and Reachability. In *Proc. of Concur'04*, volume 3170 of *LNCS*. Springer, 2004.

[LAIS06] T. Lev-Ami, N. Immerman, and M. Sagiv. Abstraction for Shape Analysis with Fast and Precise Transformers. In *Proc. of CAV'06*, volume 4144 of *LNCS*. Springer, 2006.

[Lan92] K. J. Lang. Random DFA's can be Approximately Learned from Sparse Uniform Examples. In *Proc. of the 5th ACM Workshop on Computational Learning Theory*. ACM Press, 1992.

[LBBO01] Y. Lakhnech, S. Bensalem, S. Berezin, and S. Owre. Incremental Verification by Abstraction. In *Proc. of TACAS'01*, volume 2031 of *LNCS*. Springer, 2001.

[LHR97] D. Lesens, N. Halbwachs, and P. Raymond. Automatic Verification of Parameterized Linear Networks of Processes. In *Proc. of POPL'97*. ACM Press, 1997.

[LRS05] A. Loginov, T. Reps, and M. Sagiv. Abstraction Refinement via Inductive Learning. In *Proc. of CAV'05*, volume 3576 of *LNCS*. Springer, 2005.

[LRS06] A. Loginov, T.W. Reps, and M. Sagiv. Automated Verification of the Deutsch-Schorr-Waite Tree-Traversal Algorithm. In *Proc. of SAS'06*, volume 4134 of *LNCS*. Springer, 2006.

[LYY05] O. Lee, H. Yang, and K. Yi. Automatic Verification of Pointer Programs Using Grammar-Based Shape Analysis. In *Proc. of ESOP'05*, volume 3444 of *LNCS*. Springer, 2005.

[Min67] M.L. Minsky. *Computation—Finite and Infinite Machines*. Prentice Hall, 1967.

[MQS00] K.L. McMillan, S. Qadeer, and J.B. Saxe. Induction in Compositional Model Checking. In *Proc. of CAV 2000*, volume 1855 of *LNCS*. Springer, 2000.

[MS01] A. Møller and M.I. Schwartzbach. The Pointer Assertion Logic Engine. In *Proc. of PLDI'01*. ACM Press, 2001. Also in SIGPLAN Notices 36(5), 2001.

[MS02] B. Masson and P. Schnoebelen. On Verifying Fair Lossy Channel Systems. In *Proc. of MFCS'02*, volume 2420 of *LNCS*. Springer, 2002.

[MYRS05] R. Manevich, E. Yahav, G. Ramalingam, and M. Sagiv. Predicate Abstraction and Canonical Abstraction for Singly-Linked Lists. In *Proc. of VMCAI'05*, volume 3385 of *LNCS*. Springer, 2005.

[NDQC07] Huu Hai Nguyen, Cristina David, Shengchao Qin, and Wei-Ngan Chin. Automated Verification of Shape and Size Properties via Separation Logic. In *Proc. of VMCAI'07*, volume 4349 of *LNCS*. Springer, 2007.

[Nil00] M. Nilsson. Regular Model Checking. Licentiate Thesis, Uppsala University, Sweden, 2000.

[Nil05] M. Nilsson. *Regular Model Checking*. PhD thesis, Uppsala University, 2005.

[NPW05] T. Nipkow, L.C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer, 2005.

[OG92] J. Oncina and P. Garcia. Inferring Regular Languages in Polynomial Update Time. In *Pattern Recognition and Image Analysis*, 1992.

[OSRSC01] S. Owre, N. Shankar, J.M. Rushby, and D.W.J. Stringer-Calvert. *PVS System Guide*. Computer Science Laboratory, SRI International, Menlo Park, California, USA, 2001. URL: `http://pvs.csl.sri.com/`.

[Pnu77] A. Pnueli. The Temporal Logic of Programs. In *Proc. of the 18th IEEE Symposium on Foundation of Computer Science*, 1977.

[Pnu89] A. Pnueli. In Transition from Global to Modular Temporal Reasoning about Programs. In *Logics and Models of Concurrent Systems*, NATO Asi Series F: Computer And Systems Sciences. Springer, 1989.

[PRZ01] A. Pnueli, S. Ruah, and L. Zuck. Automatic Deductive Verification with Invisible Invariants. In *Proc. of TACAS'01*, volume 2031 of *LNCS*. Springer, 2001.

[PS00] A. Pnueli and E. Shahar. Liveness and Acceleration in Parameterized Verification. In *Proc. of CAV 2000*, volume 1855 of *LNCS*. Springer, 2000.

[QS82] J.P. Queille and J. Sifakis. Specification and Verification of Concurrent Systems in CESAR. In *Proc. of ISP'82*, volume 137 of *LNCS*. Springer, 1982.

[Rey02] J.C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proc. of LICS'02*. IEEE CS Press, 2002.

[Rog06] Adam Rogalewicz. Abstract regular model checking of complex dynamic data structures - implementation details. In *Second Doctoral Workshop on Mathematical and Engineering Methods in Computer Science*, pages 198–205. Faculty of Information Technology BUT, 2006.

[RSL03] T. Reps, M. Sagiv, and A. Loginov. Finite Differencing of Logical Formluas with Applications to Program Analysis. In *Proc. of ESOP'03*, volume 2618 of *LNCS*. Springer, 2003.

[Ryb] A. Rybalchenko. ARMC: Abstraction Refinement Model Checker. URL: `http://www.mpi-inf.mpg.de/~rybal/armc/`.

[Sha01] E. Shahar. *Tools and Techniques for Verifying Parameterized Systems*. PhD thesis, Faculty of Mathematics and Computer Science, The Weizmann Institute of Science, Rehovot, Israel, 2001.

[Sha02] E. Shahar. *TLV[P] User Manual*. The Weizmann Institute of Science, Rehovot, Israel, 2002.

[SMG97] A.P. Sistla, L. Miliades, and V. Gyuris. SMC: A Symmetry Based Model Checker for Verification of Liveness Properties. In *Proc. of CAV'97*, volume 1254 of *LNCS*. Springer, 1997.

[SP02] E. Shahar and A. Pnueli. Acceleration in Verification of Parameterized Tree Networks. Technical Report MCS02-12, Faculty of Mathematics and Computer Science, The Weizmann Institute of Science, Rehovot, Israel, 2002.

[SRW02] S. Sagiv, T.W. Reps, and R. Wilhelm. Parametric Shape Analysis via 3-valued Logic. *TOPLAS*, 24(3), 2002.

[TB73] B. A. Trakhtenbrot and Y. A. Barzdin. *Finite Automata: Behavior and Synthesis*. North-Holland, 1973.

[Tou01] T. Touili. Regular Model Checking Using Widening Techniques. *ENTCS*, 50, 2001.

[URM] The Uppsala regular model checking tool. URL: `http://www.regularmodelchecking.com/`.

[Val88] A. Valmari. *State Space Generation: Efficiency and Practicality*. PhD thesis, Tampere University of Technology, Tampere, Finland, 1988.

[Val98] A. Valmari. The State Explosion Problem. In *Lectures on Petri Nets I: Basic Models*, volume 1491 of *LNCS*. Springer, 1998.

[Ven99] A. Venet. Automatic Analysis of Pointer Aliasing for Untyped Programs. *Science of Computer Programming*, 35(2), 1999.

[VSVA04a] A. Vardhan, K. Sen, M. Viswanathan, and G. Agha. Actively Learning to Verify Safety for FIFO Automata. In *Proc. of FSTTCS'04*, volume 3328 of *LNCS*. Springer, 2004.

[VSVA04b] A. Vardhan, K. Sen, M. Viswanathan, and G. Agha. Learning to Verify Safety Properties. In *Proc. of ICFEM'04*, volume 3308 of *LNCS*. Springer, 2004.

[Wal96] I. Walukiewicz. Pushdown Processes: Games and Model Checking. In *Proc. of CAV'96*, volume 1102 of *LNCS*. Springer, 1996.

[WB98] P. Wolper and B. Boigelot. Verifying Systems with Infinite but Regular State Spaces. In *Proc. of CAV'98*, volume 1427 of *LNCS*. Springer, 1998.

[WDHR06] M. De Wulf, L. Doyen, T.A. Henzinger, and J.-F. Raskin. Antichains: A New Algorithm for Checking Universality of Finite Automata. In *Proc. of CAV'06*, volume 4144 of *LNCS*. Springer, 2006.

[WKZ$^+$06] T. Wies, V. Kuncak, K. Zee, A. Podelski, and M. Rinard. On Verifying Complex Properties Using Symbolic Shape Analysis. Technical Report MPI-I-2006-2-1, Max-Planck-Institut für Informatik, Saarbrücken, Germany, 2006.

[WL89] P. Wolper and V. Lovinfosse. Verifying Properties of Large Sets of Processes with Network Invariants. In *Automatic Verification Methods for Finite State Systems, Proc. Int. Workshop, Grenoble*, volume 407 of *LNCS*, Grenoble, 1989. Springer.

[YKB02] T. Yavuz-Kahveci and T. Bultan. Automated Verification of Concurrent Linked Lists with Counters. In *Proc. of SAS'02*, volume 2477 of *LNCS*. Springer, 2002.

[YRSW03] E. Yahav, T. Reps, M. Sagiv, and R. Wilhelm. Verifying Temporal Heap Properties Specified via Evolution Logic. In *Proc. of ESOP'03*, volume 2618 of *LNCS*. Springer, 2003.

# Appendix A

# Proofs

## A.1 Proof of Lemma 1

We define a relation $\preceq \subseteq \mathcal{R}_P \times ((Lab \times 2^{\mathcal{T}(\mathcal{C})}) \times \mathcal{T}(\mathcal{C}))$ between reachable configurations of the program and configurations of the ACFG as follows. $(l, t) \preceq (\langle l_1, I\rangle, t_1)$ iff $l = l_1$, $t = t_1$ and $t_1 \in I_1$. We show that $\preceq$ is a simulation relation. Let $(l, t) \preceq (\langle l, I\rangle, t)$ and $(l, t) \xrightarrow[P]{i} (l', t')$. Then, since $(l', t') \in \mathcal{R}_P$ and $G$ covers $P$, there exists $\langle l', I'\rangle \in LI$ with $t' \in I'$ and furthermore $\langle l, I\rangle \xmapsto{i} \langle l', I'\rangle$ as $post(i, I) \cap I' \neq \emptyset$. We have $(l', t') \preceq (\langle l', I'\rangle, t')$. □

## A.2 Proof of Theorem 2

A 2-counter machine $M$ with non-negative counters $c_1, c_2$ is a sequential program:

$$0: \texttt{ins}_1; 1: \texttt{ins}_2; \cdots ; n-1: \texttt{ins}_n;$$

where $\texttt{ins}_n$ is a `halt` instruction and $\texttt{ins}_i$ with $i = 1, 2, \cdots, n-1$ are instructions of the following two types, for $0 \leq k, k_1, k_2 < n$, and $1 \leq j \leq 2$:

1. $c_j = c_j + 1; \texttt{goto}\ k;$

2. $\texttt{if}\ \ c_j = 0$ then goto $k_1$ else $(c_j = c_j - 1;$ goto $k_2)$;

We use the fact that the complement of the halting problem for 2-counter machines is not recursively enumerable, and reduce this problem to the universal halting problem for programs with trees. Let $M$ be a 2-counter machine. We built a program $P$ working with a singly-linked list[1] $l$ of length $n$ that simulates $M$ and terminates iff $M$, starting with both counters at 0, can perform at least $n$ steps. $P$ will have five variables $I_{1,2}$, $D_{1,2}$ and $E$, initially pointing to the head of the list. The control structure of $P$ is almost the same as the one of $M$, and the statements of $P$ are as follows:

[1]In fact trees are not even needed for the proof.

1. $c'_j = c_j + 1;\texttt{goto}\ \ k$; where $j = 1, 2$ is simulated by:

   ```
   $I_j$ = $I_j$.next;
   $E$ = $E$.next;
   if ($E = $ null) goto STOP; else goto $k$;
   ```

2. $\texttt{if}\ \ c_j = 0$ then goto $k_1$ else $(c_j = c_j - 1;$ goto $k_2)$; where $j = 1, 2$ is simulated by:

   ```
   if ($I_j = D_j$) $E$ = $E$.next; else goto $l$;
   if ($E = $ null) goto STOP; else goto $k_1$;
   $l$: $D_i$ = $D_i$.next;
   $E$ = $E$.next;
   if ($E = $ null) goto STOP; else goto $k_2$;
   ```

3. halt is simulated by:

   ```
   goto LOOP;
   ```

where STOP is a halting location, and LOOP is a location with a self-loop in $P$. $P$ first tests if $M$ can perform the next action, and in case the test fails, it goes into a self loop. Otherwise, when $M$ has performed a number of actions equal to the length of the list, it stops. Then $P$ stops on any input list iff $M$ runs forever. But the latter problem is the complement of the halting problem, known not to be r.e. This concludes our proof. $\square$

## A.3 Proof of Theorem 4

It is well-known that the class of relations described by conjunctions of difference constraints is closed under composition, defined as $(R_1 \circ R_2)(\mathbf{x}, \mathbf{x}') = \exists \mathbf{y} . R_1(\mathbf{x}, \mathbf{y}) \wedge R_2(\mathbf{y}, \mathbf{x}')$. In other words, the existential quantifiers can be eliminated[2], the result being written as another conjunction of difference constrains. As a consequence, we can assume w.l.o.g. that the lasso consists of only one transition described by difference constraints, together with some initial condition that summarizes the stem. In general, a relation $R(\mathbf{x}, \mathbf{x}')$ (based on conjunction of difference constraints) can be represented as a directed weighted graph whose set of vertices is the set of variables $\mathbf{x} \cup \mathbf{x}'$, and there is an edge with weight $c \in \mathbb{Z}$ from $x$ to $y$ if and only if there is an explicit constraint $x - y \leq c$ in $\varphi$. An $n$-step iteration of the loop is represented by a *constraint graph* $G_n$, defined as the minimal graph whose set of vertices is $\bigcup_{i=0}^{n} \mathbf{x}^i$, where $\mathbf{x}^i = \{y^i \mid y \in \mathbf{x}\}$ and, for all $0 \leq i < n$, there is an edge labeled $c$:

- from $x^i$ to $y^i$, if there is a constraint $x - y \leq c$ in $R$.
- from $x^{i+1}$ to $y^{i+1}$, if there is a constraint $x' - y' \leq c$ in $R$.

[2]By, e.g., the Fourier-Motzkin procedure.

- from $x^i$ to $y^{i+1}$, if there is a constraint $x - y' \leq c$ in $R$.
- from $x^{i+1}$ to $y^i$, if there is a constraint $x' - y \leq c$ in $R$.

Intuitively, the nodes $\mathbf{x^i}$ in $G_n$ represent the possible values of the counters after $i$ iterations. If $\pi : x^i \xrightarrow{c_1} \ldots \xrightarrow{c_m} y^j$, $0 \leq i, j, \leq n$ is a path in $G_n$, let $\omega(\pi)$ denote the sum of all labels along the path, i.e. $\omega(\pi) = \sum_{k=1}^{m} c_k$. Clearly, we have $x^i - y^j \leq \omega(\pi)$, and moreover, this is the strongest relation involving the values of $x$ and $y$ at the instants $i$ and $j$, respectively. Therefore, for any $n > 0$, there exists a run along the loop, consisting of at least $n$ transitions, if and only if all constraints in $G_n$ are satisfiable. The last condition is equivalent to the absence of negative cycles in $G_n$. One of the results from [CJ98, BIL06] is that the existence of a negative cycle in $G_n$, for any $n$ can be encoded by a formula in Presburger arithmetic. Consequently, the existence of an infinite run along the loop is also Presburger definable, and thus decidable. $\square$

## A.4 Proof of Theorem 5

We assume that the loop $\lambda$ of the lasso consists of at least one instruction of the form go$\{$`Left`$|$`Right`$|$`Up`$\}$ `NonNull`$(x, y)$, otherwise the set of positions of all pointer values will remain the same throughout the run, and non-termination can be trivially witnessed by a tree of height (and also size) bounded by $\|PVar\|$. This is because all guards of $\lambda$ are of the form $(x = y)$, $(x = \text{null})$, $(x.\text{data} = d)$, and boolean combinations of the above. These guards partition the set of tree positions into at most $\|PVar\| + 1$ equivalence classes[3]. If $\lambda$ has an infinite run, then all of them have to be satisfied by a given tree, whose size is bounded by $\|PVar\| + 1$.

First, we abstract the lasso $\sigma.\lambda$ into a lasso $\sigma.\lambda_a$, such that the loop $\lambda_a$ consists only of $x =$ `null`, $x = y$ and go$\{$`Left`$|$`Right`$|$`Up`$\}$`NonNull`$(x, y)$ instructions. This is done by replacing all guards with `if(true)`, go$\{$`Left`$|$`Right`$|$`Up`$\}$`Null`$(x, y)$ instructions by $x =$ `null`, and eliminating all `changeData`$(x, d_1, d_2)$ instructions. It can be shown that, if there exists a non-terminating run following the path $\sigma.\lambda^i.\left(\lambda^j\right)^\omega$, for some $i, j > 0$, then there exists also a non-terminating run following the path $\sigma.\lambda^i.\left(\lambda_a^j\right)^\omega$. Note that $|\lambda_a| \leq |\lambda|$.

Second, we encode the loop $\lambda_a$ as a linear transformation $\mathbf{p}' = A\mathbf{p} + B$ where $\mathbf{p}, \mathbf{p}'$ and $B$ are column vectors of size $\|PVar\|$, and $A$ is a $\|PVar\| \times \|PVar\|$ matrix. As previously discussed, each component of $\mathbf{p}$ encodes the current position of a pointer variable, $A$ has at most one non-zero element on each line, and this is a composition of $2\star$ and $\frac{1}{2}$, and $B$ is composed only of 0 and positive integer values.

The condition of a non-terminating run along the path $\sigma.\lambda^i.\left(\lambda_a^j\right)^\omega$ implies that all pointer variables occurring in an instruction of the form go$\{$`Left`$|$`Right`$|$`Up`$\}$`NonNull`$(x, y)$ have to be non-null infinitely often, otherwise one of the guards would be violated and the instruction would not be executable. This induces the following condition on the counter

[3]Note that guards of the form $(x.\text{data} = d)$ may not introduce more classes in this partition due to the relation between pointer and data equality, namely $x = y \Rightarrow x.\text{data} = y.\text{data}$.

variables from $\mathbf{p}$: for each $x \in PVar$ that occurs in an go{Left|Right|Up}NonNull$(x, y)$ instruction, the value of $p_x$ (1) has to be infinitely often non-zero and (2) may not increase indefinitely.

The property of $A$ of having at most one non-zero element on each line can be formalized as a partial mapping $nz : \{1, \ldots, \|PVar\|\} \rightarrow \{1, \ldots, \|PVar\|, \perp\}$ where for a line number $1 \leq i \leq \|PVar\|$, $nz(i)$ is the position of the non-zero element, if any, or $\perp$ otherwise. Let $A_0$ be the submatrix of $A$, and $B_0$ be a subvector of $B$, obtained in the following way:

- for all $1 \leq i \leq \|PVar\|$ if line $i$ of $A$ has only zero elements, then cross out line $i$, as well as column $i$ of $A$, and line $i$ of $B$.
- for each line $i = 1, 2, \ldots, \|PVar\|$ cross out all lines and columns of $A$ $j > i$, with $nz(j) = nz(i)$, as well as each line $j$ of $B$.

The resulting matrix $A_0$ has exactly one non-zero element on each line and each column. The intuition behind this is to transform the initial system $\mathbf{p}' = A\mathbf{p} + B$ into a system $\mathbf{p_0'} = A_0\mathbf{p_0} + B_0$ in which the next value of each variable is computed from the value of another variable, no variable has a constant value, and no two variables are computed based on the same variable. Note that, for all counters $p_x \in \mathbf{p_0}$ that occur in the new system, the corresponding pointer variables $x \in PVar$ have to occur within either go{Left|Right|Up}NonNull$(x, y)$ or $x = y$ instructions.

In analogy with a permutation matrix, there exists a constant $k$ such that ${A_0}^k$ is a diagonal matrix[4]. This constant is bounded by $3^m \leq 3^{\|PVar\|}$ where $m$ is the size of $A_0$[5]. Moreover, the elements on the main diagonal of ${A_0}^k$ are (at most $k$ times) compositions of $2\star$'s and $\frac{1}{2}$'s.

Consequently, we have $A_0^n = \left(A_0^k\right)^{n/k} \cdot A_0^{n \mod k}$, for any $n \geq 0$, i.e. $\left(A_0^n\right)_{ij} = \left(\left({A_0}^k\right)_{ii}\right)^{n/k} \cdot \left(A_0^{n \mod k}\right)_{ij}$ for all $1 \leq i, j \leq m$. Having obtained a closed-form expression for $A_0^n$, allows us to express the values of each $p_x$, $x \in PVar$ as functions of the form:

$$\sum_{i=1}^{m} \left(\left({A_0}^k\right)_{ii}\right)^{n/k} \alpha_i(n \mod k)$$

where $\alpha_i$ are positive coefficients depending on $n \mod k$.

In order to meet conditions (1) and (2) it must be that $\left({A_0}^k\right)_{ii}$ is a compositions of as many $2\star$'s as $\frac{1}{2}$'s. For assume that the number of $2\star$'s is bigger than the number of $\frac{1}{2}$'s in some $\left({A_0}^k\right)_{ii}$, then the value of $\mathbf{p}_i$ increases indefinitely. Else, if the number of $\frac{1}{2}$'s is

[4]If $A_0$ had only 0 and 1 elements, with exactly only 1 on each line and column, it would represent a permutation, and the constant $k$ would be the permutation order. The behavior of $A_0$ is similar to the one of a permutation matrix, except the fact that, at each iteration the non-zero values of $A_0^i$ are products of its non-zero elements.

[5]The permutation order is the least common multiple of the sizes of all cyclic permutations, which sum up to the size of the matrix. The problem is to bound the maximum value of $\Pi_{i=1}^{l} m_i$ where $m_1, m_2, \ldots, m_l$ are such that $m = \sum_{i=1}^{l} m_i$. A bound of $3^{\frac{m}{3}}$ can be shown. A slight variation of this problem has been proposed at the 18th International Mathematical Olympiad in 1976 [IMO].

bigger than the number of $2^s$'s, the value of $\mathbf{p}_i$ will eventually become zero and stay zero from that moment on.

Thus, $A_0^n$ is a periodic function of period $k$. As a consequence, the positions of all pointer variables are given by periodic functions as well, of period $k \leq 3^{\|PVar\|}$. After the execution of the stem, the variables can move at most $|\sigma|$ levels from their initial positions. Analogously, after each iteration of the loop, the difference in the level may be of at most $|\lambda|$, for each variable. Hence we need only consider trees of height at most $(\|PVar\| + 1) \cdot (|\sigma| + |\lambda| \cdot k) \leq (\|PVar\| + 1) \cdot \big(|\sigma| + |\lambda| \cdot 3^{\|PVar\|}\big)$. □

## A.5 Proof of Corollary 1

If $\chi$ is a composition of the $x \mapsto 2 \star x$ and $x \mapsto \frac{1}{2}x$ functions, let $\#_2(\chi)$ denote the number of occurrences of 2, and $\#_{\frac{1}{2}}(\chi)$ denote the number of occurrences of $\frac{1}{2}$ in $\chi$.

(a) To test whether $S.L$ is spurious or not, proceed as in the proof of Theorem 5, first abstracting $\lambda$ into $\lambda_a$ by eliminating instructions relative to the data and guards, and then build the linear system $\mathbf{p}' = A\mathbf{p} + B$ encoding the operations of $\lambda_a$, then compute $A_0$ from $A$, and iterate it to the diagonal form $A_0^k$. There are two cases:

1. $\#_2\Big(\big(A_0^k\big)_{ii}\Big) \neq \#_{\frac{1}{2}}\Big(\big(A_0^k\big)_{ii}\Big)$. Then either $\#_2\Big(\big(A_0^k\big)_{ii}\Big) > \#_{\frac{1}{2}}\Big(\big(A_0^k\big)_{ii}\Big)$, in which case the pointer variable corresponding to $\mathbf{p_0}_i$ will move down in the tree every $k$ iterations, otherwise $\#_2\Big(\big(A_0^k\big)_{ii}\Big) < \#_{\frac{1}{2}}\Big(\big(A_0^k\big)_{ii}\Big)$ and it will move up. In both cases, the iteration of $\lambda_a$ is bound to terminate for all input values, which implies the spuriousness of the original lasso.

2. $\#_2\Big(\big(A_0^k\big)_{ii}\Big) = \#_{\frac{1}{2}}\Big(\big(A_0^k\big)_{ii}\Big)$. Then we cannot infer termination of the lasso directly from the form of $A_0^k$. By Theorem 5 we have that, if the lasso is non-spurious, there is a tree of height bounded by $(\|PVar\| + 1) \cdot \big(|\sigma| + |\lambda| \cdot 3^{\|PVar\|}\big)$ that witnesses non-termination. In case $\|Data\| = 1$, the number of such trees is bounded by $2^{(\|PVar\|+1)\cdot\left(|\sigma|+|\lambda|\cdot 3^{\|PVar\|}\right)}$, otherwise by $\|Data\|^{2^{(\|PVar\|+1)\cdot\left(|\sigma|+|\lambda|\cdot 3^{\|PVar\|}\right)}}$. In general, the number of trees is bounded by $\mathbf{max}(2, \|Data\|)^{2^{(\|PVar\|+1)\cdot\left(|\sigma|+|\lambda|\cdot 3^{\|PVar\|}\right)}}$.
   As the loop has no destructive updates, at most
   $|\lambda| \cdot \mathbf{max}(2, \|Data\|)^{2^{(\|PVar\|+1)\cdot\left(|\sigma|+|\lambda|\cdot 3^{\|PVar\|}\right)}}$ program configurations of the form $\langle l, t\rangle \in Lab \times \mathcal{T}(\mathcal{C})$ are reachable by the loop. Hence a non-terminating run along $\lambda$ can be detected after at most as many steps, starting with some tree $t_{0b} \in I_0$ of bounded size.

(b) For the second part of the proof, let us suppose that $S.L$ is a spurious lasso, meaning that there is a non-terminating computation along this lasso in $A_{rsc}(G)$, but there is no such computation along $\sigma.\lambda$ in $P$. Then, there are again two cases:

1. $\#_2\Big(\big(A_0^k\big)_{ii}\Big) \neq \#_{\frac{1}{2}}\Big(\big(A_0^k\big)_{ii}\Big)$ for some $1 \leq i \leq m$. Let $x \in PVar$ be the pointer variable corresponding to $\mathbf{p_0}_i$, and $r_x$ be the counter of $A$, keeping track of the

distance of $x$ from the root of the tree. It can be shown, by induction of the structure of $\lambda_a$ that:

(a) if $\#_2\Big(\big(A_0^k\big)_{ii}\Big) > \#_{\frac{1}{2}}\Big(\big(A_0^k\big)_{ii}\Big)$, then the increment of $r_x$ in $L$ is greater or equal to 1, and

(b) if $\#_2\Big(\big(A_0^k\big)_{ii}\Big) < \#_{\frac{1}{2}}\Big(\big(A_0^k\big)_{ii}\Big)$, then the increment of $r_x$ in $L$ is smaller or equal to $-1$.

In other words, the fact that a pointer is moving up or down in the tree is witnessed in both encodings, the one from the proof of Theorem 5 and the one from Table 5.1. As a consequence $S.L$ will terminate in $A_{rsc}(G)$ since there will be at least one $r_x$ counter whose value will increase or decrease indefinitely. But this contradicts the hypothesis that the lasso is spurious.

2. $\#_2\Big(\big(A_0^k\big)_{ii}\Big) = \#_{\frac{1}{2}}\Big(\big(A_0^k\big)_{ii}\Big)$. In this case, all pointer variables are confined to move up or down at most $|\sigma| + |\lambda| \cdot 3^{\|PVar\|}$ levels from the initial position. Therefore, the sizes of all subtrees that are visited during any run from some initial tree $t_0 \in I_0$ sum up to at most $2^{|\sigma|+|\lambda|\cdot 3^{\|PVar\|}}$. Then, the maximum number of steps taken by any finite run starting with $t_0$ is bounded by $|\lambda| \cdot \mathbf{max}(2, \|Data\|)^{2^{(\|PVar\|+1)\cdot\left(|\sigma|+|\lambda|\cdot 3^{\|PVar\|}\right)}}$, which is the maximum number of program configurations that can be visited at most once. Note that this is the maximum run on a spurious lasso, because any run longer than this must visit the same configuration twice, being in fact a non-terminating run. □