

RHEINISCH-WESTFÄLISCHE TECHNISCHE HOCHSCHULE AACHEN  
Chair for Software Modeling and Verification  
Prof. Dr. Ir. Dr. h. c. Joost-Pieter Katoen

---

Master Thesis

---

# Comparing Hierarchical and On-The-Fly Model Checking for Java Pointer Programs

---

**Sally Chau**

August 23, 2019

First Reviewer: apl. Prof. Dr. Thomas Noll  
Second Reviewer: Prof. Dr. Ir. Dr. h. c. Joost-Pieter Katoen  
Supervisor: Christoph Matheja



# Acknowledgement



# Eidesstattliche Erklärung

Hiermit versichere ich an Eides statt und durch meine Unterschrift, dass die vorliegende Arbeit von mir selbstständig, ohne fremde Hilfe angefertigt worden ist. Inhalte und Passagen, die aus fremden Quellen stammen und direkt oder indirekt übernommen worden sind, wurden als solche kenntlich gemacht. Ferner versichere ich, dass ich keine andere, außer der im Literaturverzeichnis angegebenen Literatur verwendet habe. Die Arbeit wurde bisher keiner Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

---

Aachen, den 19. Juli 2019, Sally Chau



# Abstract





# Contents

<b>1</b>	<b>Introduction</b>	<b>12</b>
1.1	Related Work . . . . .	16
<b>2</b>	<b>Preliminaries</b>	<b>18</b>
2.1	ATTESTOR . . . . .	18
2.2	Heap Representation . . . . .	22
2.3	Transition Systems . . . . .	26
2.4	Recursive State Machines . . . . .	30
2.5	Linear Temporal Logic . . . . .	34
<b>3</b>	<b>Hierarchical Model Checking</b>	<b>41</b>
3.1	Tableaux Construction . . . . .	41
3.2	Model Checking Contracts . . . . .	50
<b>4</b>	<b>On-The-Fly Hierarchical Model Checking</b>	<b>52</b>
4.1	Algorithm . . . . .	52
4.2	Implementation . . . . .	55
4.3	Discussion . . . . .	57
<b>5</b>	<b>RSM-based Hierarchical Model Checking</b>	<b>59</b>

5.1	Algorithm . . . . .	60
5.2	Implementation . . . . .	62
5.3	Discussion . . . . .	63
<b>6</b>	<b>Benchmarks</b>	<b>66</b>
6.1	Experimental Setup . . . . .	66
6.2	Instances . . . . .	67
6.3	Results . . . . .	69
6.3.1	Effectiveness . . . . .	69
6.3.2	Efficiency . . . . .	70
<b>7</b>	<b>Conclusion and Future Work</b>	<b>75</b>
<b>A</b>	<b>Benchmark results</b>	<b>78</b>



# Chapter 1

## Introduction

Nowadays, software systems are found in every corner in everyday life. Smart devices, automated processes, and digital support in economy and industry are indispensable. The more important is a system's reliability including bug-free behavior. Think of an autonomous car that processes a number of inputs in order to drive safely from A to B. The aftermath of a faulty velocity value communicated between the steering computer and the car's mechanics could cause substantial damage. Software verification approaches the problem of assuring error-free behavior of a system by checking whether the system under consideration satisfies a specified property. In contrast to testing, software verification does not only discover errors, but ensures their absence. Examples for properties checked during verification are reachability properties, memory leaks, and cycle detection. A formal verification technique in software verification is *model checking*. The analysis yields three possible outcomes:

1. The program *satisfies* the property.
2. The program *violates* the property. The analysis returns a counterexample indicating at which state of the program the property is violated.
3. It is *unknown* whether the program satisfies the property. This answer is returned if the analysis terminates as computational resources are exceeded, e.g., the computation runs out of memory.

In order to apply model checking for analyzing a system, two ingredients are required: a suitable *model* of the system and a formal description of the *properties* that are checked.

In this thesis, we focus on effectively and efficiently model checking JAVA pointer-programs with recursive method calls. We refer to this problem as *hierarchical model checking* as the model induced by recursive programs is a hierarchical one due to different levels of method calls. Hierarchical model checking includes three main challenges. One challenge is that method calls induce sub-models that need

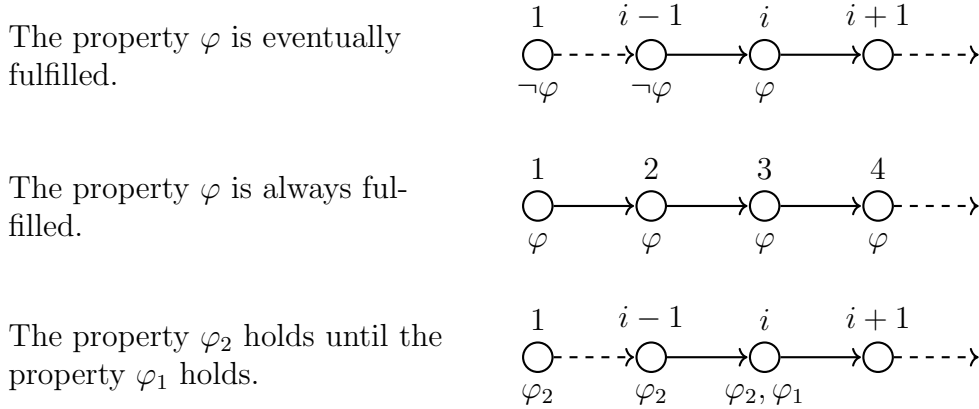


Figure 1.1: Examples for modalities of time in linear temporal logic.

to be computed and checked which leads to an increase in time and space demand compared to model checking single level models. Moreover, reoccurring procedure calls need to be handled such that already generated models or checked verification goals are not computed repeatedly. Another aspect to consider in hierarchical model checking is that the recursion depth of a procedure can be unbounded.

We base our work on the ATTESTOR framework by Heinen et al. [8], a model checking tool for JAVA pointer programs. We model the system (or program) under consideration as a graph, called *state space* or *transition system*, where vertices represent states of the system and edges between two states model a transition from one state to another. A transition represents a change in a system's state. Each state models a point during program execution by representing the current program heap as a graph. Pointer programs contain dynamic data structures that induce potentially infinite state spaces as objects are created at runtime. Also arbitrary sizes of input to a program have to be considered. In order to finitely represent possibly infinite states, Heinen et al. employ *hypergraphs* that summarize data structures by replacing subgraphs with placeholders (also called non-terminals) [8]. Placeholders represent several subgraphs and offer a compact representation of possibly infinite data structures. A hypergraph containing placeholders is called abstract while a hypergraph without any placeholders is called concrete. The process of replacing subgraphs by placeholders is called abstraction while the reverse is called concretisation. The rules that define how abstraction and concretisation are performed are summarized in graph grammars.

Model checking is performed for the generated state space and a property specified by formulae in *linear temporal logics* (LTL). LTL is a logic that formulates properties for paths with respect to modalities of time. Here, time is understood as a linear sequence of moments where each moment is followed by a well-defined succes-

sor. Figure 1.1 illustrates examples for modalities of time in LTL, e.g., a property is eventually fulfilled, always satisfied, or holds until another one becomes true. In order to verify LTL formulae for a state space, ATTESTOR implements the tableaux construction algorithm by Grumberg et al. [4] that constructs a proof structure based on a set of rules in order to infer whether a state satisfies a property. The approach thus successively divides a verification goal into subgoals until a statement can be made about the result of the initial verification goal.

Programs with procedure calls induce (procedure) state spaces. In ATTESTOR, effects of procedure calls are applied to the return state in the state space of the main method of an input program. Therefore, the generated state space reflects the behavior of the main method. Hence, model checking is executed for the top level state space only while procedure state spaces are not directly included. The question arises whether solely verifying a program's top level state space is sufficient to conclude about the validity of the entire system with respect to a specified property.

Consider the JAVA class `SLList` specified in Listing 1.1 that defines a singly-linked list with a pointer `next` to the next element in a list. A method `traverse` that traverses a singly-linked list is given in Listing 1.2. We added the procedure `noop` that takes a list as an input and dereferences the list's `next` pointer, but references the original value again upon return. Thus, the original list is modified during execution of the procedure `noop`, but not in the main procedure `traverse`. We verify whether the method `traverse` satisfies the LTL formula

$$\Box \{\text{identicNeighbours}\}$$

which expresses that for every state all neighbors of a list element remain identical during execution of the program, e.g., the list is never modified.

For the given input program in Listing 1.2 and the formula  $\Box \{\text{identicNeighbours}\}$ , ATTESTOR's model checking procedure returns that the property, that the list is never mutated, is satisfied. However, the list is indeed modified within the procedure `noop`. As this modification has no effect on the top level state space, the failure cannot be detected by the currently implemented top level model checking algorithm. Therefore, in this thesis, we introduce two hierarchical model checking algorithms that do not only check the top level state space but also verify procedure state spaces. The first algorithm is an on-the-fly approach that combines an on-the-fly state space generation with the tableaux construction for model checking. This approach allows for efficient detection of errors in a program that does not require an entirely generated (procedure) state space. The second algorithm captures the hierarchical structure of programs with (recursive) procedure calls within recursive state machines (RSM) as introduced in [1]. The tableaux construction is then performed for the RSM in order to verify given LTL specifications.

```

1 public class SLList {
2
3 public SLList next;
4
5 public SLList(SLList next) {
6 this.next = next;
7 }
8 }

```

Listing 1.1: JAVA class definition for singly-linked lists.

```

1 public static void traverse(SLList head) {
2
3 SLList current = head;
4
5 while (current.next != null) {
6 current = current.next;
7 head = noop(head);
8 }
9 }
10
11 public static SLList noop(SLList head) {
12
13 SLList list = head.next;
14
15 head.next = null;
16 head.next = list;
17
18 return head;
19 }

```

Listing 1.2: JAVA code for traversing a singly-linked list that contains erroneous behaviour in a called procedure.

The theoretical foundations of this thesis are described in Chapter 2 including a detailed description of the ATTESTOR state space generation, the concepts of state spaces by Heinen et al. [8], recursive state machines by Alur et al. [1], and linear temporal logics. Chapter 3 presents the tableaux construction for model checking LTL formulae by Grumberg et al. [4]. Thereupon, Chapters 4 and 5 introduce the on-the-fly hierarchical and RSM-based model checking algorithms, respectively. The thesis concludes with experimental results on the effectiveness and efficiency of the introduced algorithms compared to top level model checking in Chapters 6 and 7.

## 1.1 Related Work

Several algorithms on model checking recursive software programs have been developed in the research of LTL model checking. In [5], Esparza et al. propose an automata-theoretic approach that reduces the model checking problem to the accepting runs problem of a Büchi pushdown system. A Büchi pushdown system is the product of a pushdown system that represents the underlying input program and a Büchi automaton that corresponds to the negation of the LTL formula to be verified. Thus, an empty set of configurations that have an accepting run in the Büchi pushdown system denotes that no run in the pushdown system satisfies the negated input formula, i.e., the formula is satisfied. The algorithm runs in  $\mathcal{O}(|\mathcal{P}^3||\mathcal{B}^3|)$  time and  $\mathcal{O}(|\mathcal{P}^2||\mathcal{B}^2|)$  space, where  $|\mathcal{P}|$  and  $|\mathcal{B}|$  denote the size of the pushdown system and the size of a Büchi automaton for the negation of the LTL formula, respectively. Hence, Esparza et al. show that the model checking problem for linear temporal logics can be solved in polynomial time.

In [6], Esparza et al. extend the automata-theoretic approach for Boolean programs with possibly recursive procedures by representing the underlying input program as a symbolic pushdown system. A symbolic pushdown system is a compact representation of a pushdown system where a symbolic rule denotes a set of transition rules in the corresponding pushdown system. Thus, symbolic pushdown systems finitely capture possibly infinite states. Furthermore, they use binary decision diagrams to encode the values of Boolean variables.

Alur et al. [1] address the problem of model checking recursive input programs that induce possibly infinite hierarchies in state spaces by introducing recursive state machines. Closely related to pushdown systems, recursive state machines model the control flow of input programs with recursive procedure calls. Every procedure is represented by a component within the state machine. A state in a recursive state machine is either a state in a component or a box that represents a procedure call. Alur et al. adapt the automata-theoretic approach for model checking recursive state machines.

We base our work on the previously introduced results and employ the notion of recursive state machines for model checking LTL formula. However, we will refrain from an automata-theoretic approach and focus more on an on-the-fly model checking algorithm as introduced in [4] that constructs a proof structure for the underlying state space in order to verify a linear-time property. We introduce an on-the-fly and an RSM-based hierarchical model checking algorithm. The first algorithm works completely on-the-fly, i.e., state space generation is executed on demand if the tableaux construction requires so. This approach is promising in detecting property violations at an early stage avoiding the need to generate an entire state



---

space. The second algorithm executes the tableaux construction on recursive state machines. By combining the compact representation of recursive programs in recursive state machines and the efficient on-the-fly model checking algorithm, the approach offers an efficient way to verify a set of linear-time properties for state spaces once computed. We evaluate the introduced algorithms by comparing them to ATTESTOR's top level model checking algorithm.

# Chapter 2

## Preliminaries

This chapter presents the foundations of this thesis starting with a description of the ATTESTOR framework in Section 2.1. Thereupon, Sections 2.2 and 2.3 describe how an input program is modeled by a state space including hypergraphs that represent heaps for each state according to Heinen et al. [8]. In order to describe hierarchical structures relevant to modeling programs with procedure calls, we focus on recursive state machines based on the work of Alur et al. [1] in Section 2.4. The chapter concludes by presenting linear temporal logic that formalizes properties with temporal modalities in order to verify an input program.

### 2.1 Attestor

ATTESTOR is a verification tool that checks properties given as LTL formulae for a JAVA program. The tool generates a state space for an input program, where each state represents the current heap and transitions between states model changes in a state. Procedures of the input program are summarized by procedure contracts that specify the heap prior to and after their execution. Thus, procedure state spaces do not need to be computed repeatedly for the same input heap. After the state space has been generated, model checking is performed by checking the resulting state space against LTL specifications. Possible outcomes for the model checking procedure are that the input program either fulfills the property, violates the property, or the result is unknown. In case of property violation, a counterexample is provided.

The ATTESTOR architecture is depicted in Figure 2.1. The tool offers several options to specify a verification task including a JAVA program to be analyzed and LTL formulae that describe the properties verified during model checking. Moreover, ATTESTOR requires graph grammars that define data structures present in the input program. Next to the pre-defined grammars in the tool (singly-linked lists, doubly-linked lists, trees), the user has the option to define custom ones. Further options include the specification of initial heaps or properties for state space gener-

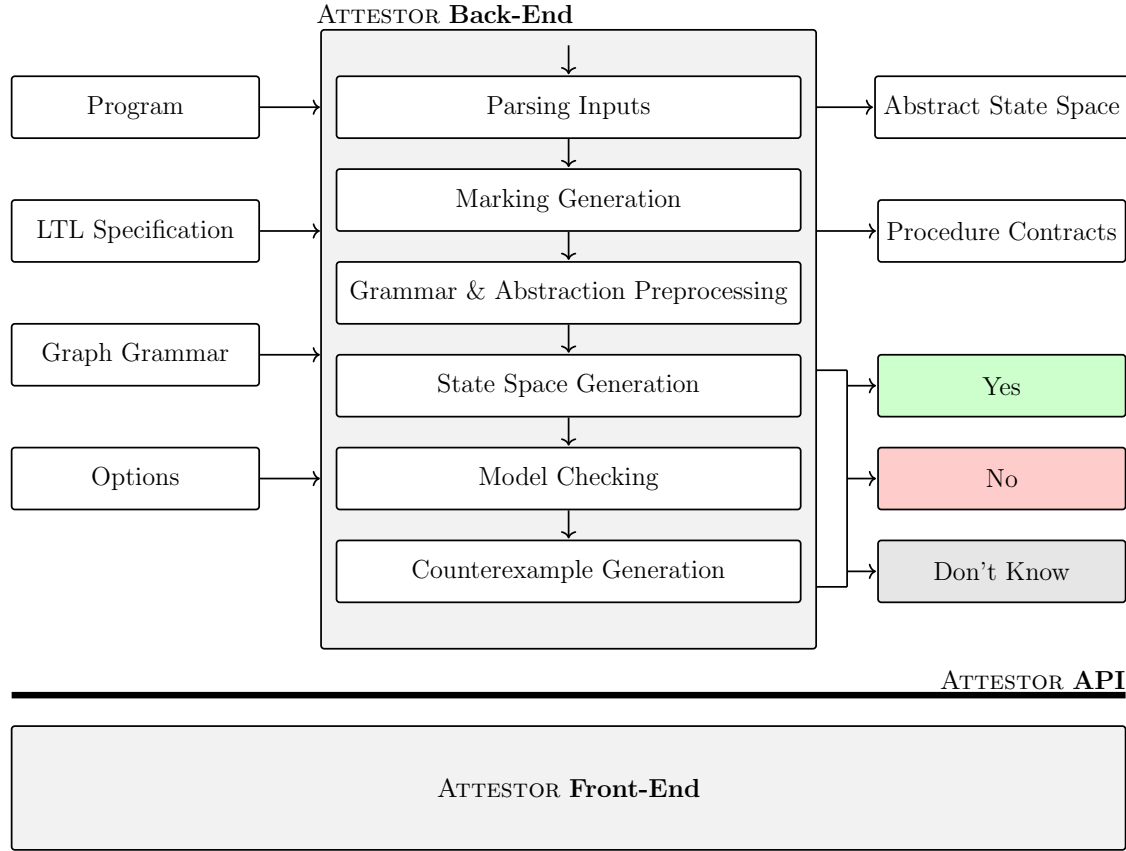


Figure 2.1: ATTESTOR architecture. [2]

ation. The ATTESTOR back-end constitutes the program analysis which is divided into six phases. The first three phases comprise preprocessing of the verification task which are followed by the state space generation and the model checking phase.

**Phase 1: Parsing Inputs** In the first phase, supplied input options passed to ATTESTOR, including the input program and optional parameters, are parsed.

**Phase 2: Marking Generation** After parsing the input, markings are added to the initial heap if required by the specified LTL properties [7]. The markings track object identities during program execution along sequences of states so that properties such as neighborhood preservation can be checked.

**Phase 3: Grammar and Abstraction Preprocessing** In this phase, state space generation is prepared by refining the input grammar such that properties can be decided more efficiently, e.g., by only considering hypergraphs that satisfy a specified property. Furthermore, abstraction preprocessing computes the trans-

formers required for state space generation itself, e.g., garbage collection.

**Phase 4: State Space Generation** After the stages of preprocessing, the state space of the input program is generated by executing program statements on the initial heap such that new states are added. The procedure is illustrated in Figure 2.2. The abstract execution loop is executed until either there are no more states left to process, a fixpoint has been reached or computational resources are exceeded. The loop starts with picking a state, which has not been processed yet, in a DFS manner, i.e., unprocessed states are added to and removed from the end of a list of unprocessed states. The abstract semantics of the next statement are applied to the state. As hypergraphs potentially contain abstracted parts representing a number of concrete subgraphs, the heap potentially needs to be concretised so that the statement is executed on a concrete heap. Concretisation is achieved by applying grammar rules in a forward manner. Thereafter, the heap is cleared, e.g., dead variables are removed and the garbage collector performs its actions. In order to obtain a compact heap representation, heap abstraction is performed by applying grammar rules in a backward manner introducing placeholders for fragments of the graph. Finally, the resulting state is labeled with atomic propositions that are satisfied by the heap. Labels provide information about structural properties about the heap, e.g., the shape of the heap, reachability information or the state is a terminal state. The labeling is implemented by heap automata [2]. Before adding the resulting state to the state space, it is checked whether a more abstract state already covers the current one. If not, the state is added to the state space and the algorithm continues with the next state. Otherwise, ATTESTOR checks if a fixpoint has been reached and terminates the procedure if found `true`. The generated state space represents the transformation of the initial heap during program execution for the main method of the input program.

**Phase 5: Model Checking** ATTESTOR implements the tableaux method described in Section 3.1 which verifies specified LTL formulae for the main state space returned by the previous state space generation phase. In case a formula is violated, a failure trace is returned that constitutes a counterexample generated in the next phase. If all properties are satisfied, ATTESTOR outputs accordingly. A drawback of the current model checking procedure is that state spaces induced by procedural programs with (recursive) methods are not directly checked. Rather, effects of procedure calls are woven into the main state space by considering their influence on the heap only after execution of the called procedure. However, properties should also be checked for procedure state spaces themselves as they might introduce violations not visible in the main state space. We approach this gap by considering hierarchical model checking in Chapters 4 and 5 that also verifies procedure state spaces for specified LTL properties.

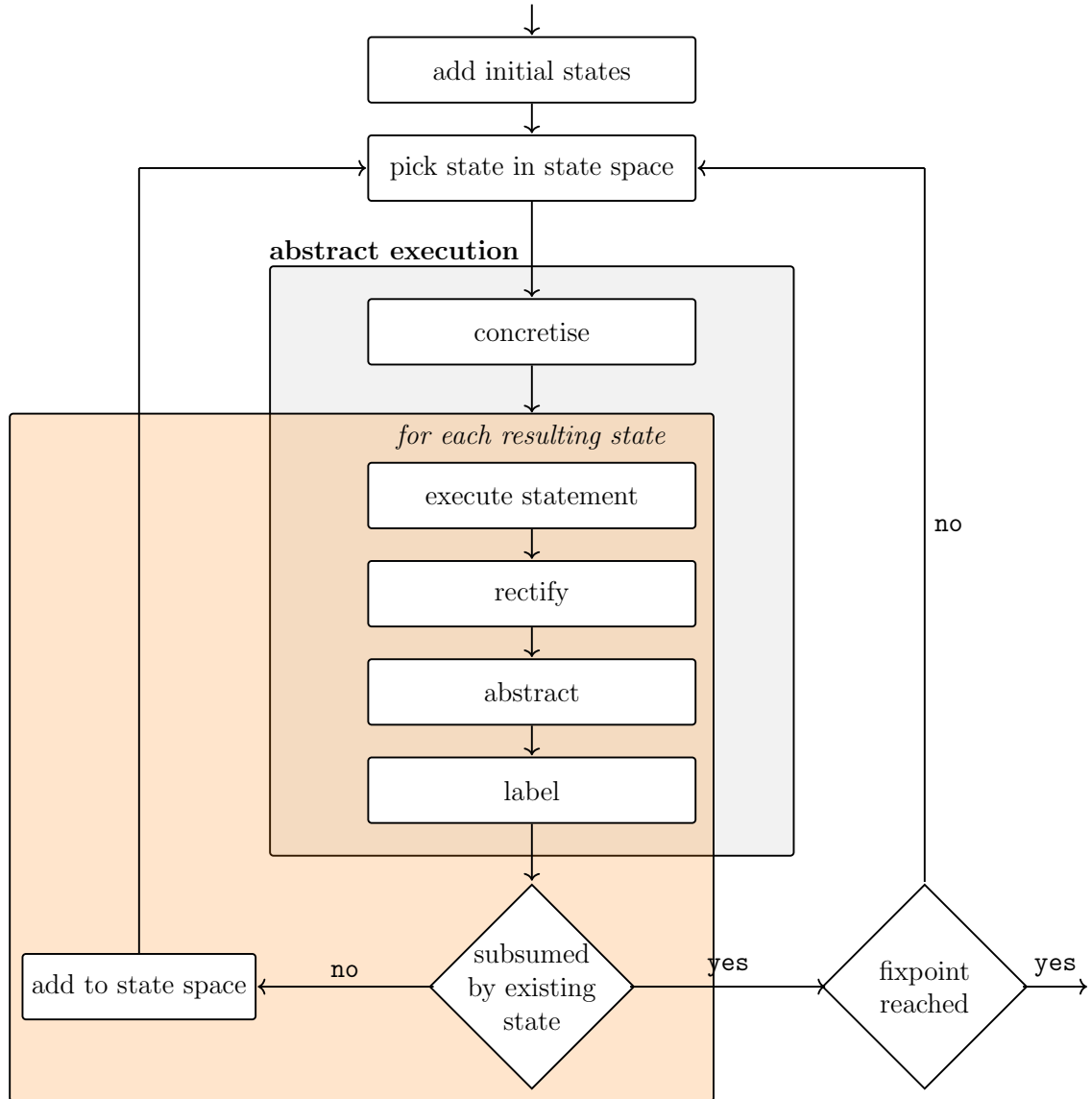


Figure 2.2: ATTESTOR's state space generation phase [2].

**Phase 6: Counterexample Generation** In case an LTL formula is found to be violated, the model checking phase returns a failure trace. Together with the violated formula a counterexample is generated in this phase in order to provide an instance for debugging purposes.

## 2.2 Heap Representation

Our focus in model checking lies on pointer-manipulating programs. Therefore, we consider the heap of the program under consideration at every state. These are generated during ATTESTOR’s state space generation phase (see Figure 2.1). The heap encompasses a set of locations with variables and pointers, i.e., references to locations. It is represented by a graph where vertices represent heap objects, directed edges depict selectors, and labeled edges depict the mapping of program variables to heap objects. Selectors are pointer variables that are connected to two vertices, i.e., a selector points from its source to its target node.

Consider the class definition of a singly-linked list in JAVA code in Listing 1.1 in Chapter 1. It defines a singly-linked list with a selector `next` that points to an element’s successor node. Figure 2.3 illustrates a heap for a singly-linked list. The list consists of four elements represented by circles. The grayed circle at the end of the list represents the `null`-value. The `next`-selectors are represented by directed edges. Furthermore, the program variable `head` is attached to the first vertex of the list via an edge labeled with its rank. It indicates the number of vertices an edge connects, i.e., the rank of variable edges is one. Pointer-manipulating operations are represented by graph transformations. For instance, executing the statement `head := head.next` on the heap given in Figure 2.3 moves the variable `head` to the second vertex of the list.

Take into account a program that contains a loop in which a new element is added to a list with every iteration. Thus, the heap size increases with every iteration. If the loop is visited infinitely often, the number of objects in a heap can become unboundedly large. In order to obtain a finite representation, parts of the graph are abstracted, i.e., a subgraph is replaced by a placeholder. An example for an abstracted graph is depicted in Figure 2.4. The graph shows a singly-linked list with a placeholder labeled *SL* which represents a singly-linked list of arbitrary length. The example illustrates the concept of *hypergraphs*. Hypergraphs are graphs in which an edge is connected to an arbitrary number of nodes. These edges are called *hyperedges*. The number of vertices a hyperedge connects is captured in its *rank*. Hypergraphs represent heaps that are partially concrete and partially abstract. In our example from Figure 2.4, the hyperedge labeled *SL* is an abstracted part of the hypergraph in the otherwise concrete graph. In order to express the abstracted parts of the heap, we require a *ranked alphabet*  $\Sigma = \Sigma_N \uplus \Sigma_T$ , where  $\Sigma_N$  denotes a

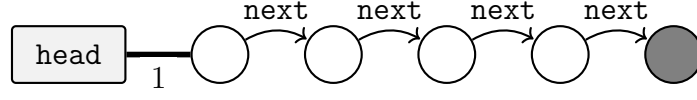


Figure 2.3: Heap for a singly-linked list.

finite set of *nonterminal symbols* and  $\Sigma_T = Var \uplus Sel$  denotes the terminal symbols including the set  $Var$  of variables and the set  $Sel$  of selectors. Program variables are of rank one, while selectors are of rank two. Hypergraphs over the alphabet  $\Sigma_T$  describe concrete heaps that do not contain abstract parts such as the heap shown in Figure 2.3.

**Definition 2.1: Hypergraph [7]**

Given a finite ranked alphabet  $\Sigma = \Sigma_N \uplus \Sigma_T$  with associated ranking function  $\text{rk} : \Sigma \rightarrow \mathbb{N}$ . A (labeled) hypergraph over  $\Sigma$  is a tuple

$$H = (V, E, \text{att}, \text{lab}, \text{ext})$$

where

- $V$  is a finite set of vertices,
- $E$  is a finite set of hyperedges,
- $\text{att} : E \rightarrow V^*$  is the attachment function that maps each hyperedge to a sequence of incident vertices,
- $\text{lab} : E \rightarrow \Sigma$  is the hyperedge-labeling function that maps a label to an edge, and
- $\text{ext} \in V^*$  is the (possibly empty) sequence of pairwise distinct external vertices.

For every  $e \in E$ , we let  $\text{rk}(e) = |\text{att}(e)|$  and we require  $\text{rk}(e) = \text{rk}(\text{lab}(e))$ . The set of all hypergraphs over  $\Sigma$  is denoted by  $HG_\Sigma$ .

In order to obtain all possible heaps represented by an abstract subgraph, we require the concept of *graph grammars*. Graph grammars define a set of production rules that define how nonterminals can be replaced by hypergraphs. Nonterminals represent abstract parts of the graph such as  $SL$  in Figure 2.4. Exhaustively applying production rules to a graph gradually replaces nonterminals by graphs so that concrete graphs without nonterminals can be reached eventually. Graph grammars are therefore comparable to string grammars that define rules to manipulate strings.

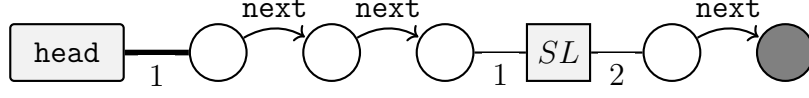


Figure 2.4: A singly-linked list with an abstracted subgraph represented as a hypergraph.

As we consider hypergraphs in our analysis, we focus on hyperedge replacement grammars that are graph grammars working on hypergraphs, where nonterminals represent hyperedges.

**Definition 2.2: Hyperedge Replacement Grammar [7]**

A *hyperedge replacement grammar*  $G$  over a ranked alphabet  $\Sigma$  is a set of production rules of the form  $X \rightarrow R$ , where  $X \in \Sigma_N$  is a nonterminal and  $R \in HG_\Sigma$  is a rule graph, i.e., a hypergraph with  $|ext_R| = \text{rk}(X)$ .

The language of a hyperedge replacement grammar contains all concrete hypergraphs that are obtained by repeatedly applying production rules to a given hypergraph. In this context, a nonterminal in a hypergraph is replaced by the rule graph of a production rule. Therefore, the external nodes of the replacing hypergraph are mapped to the hyperedges of the nonterminal to be replaced. To illustrate an hyperedge replacement, consider the hyperedge replacement grammar given in Figure 2.5. The grammar describes the language of all singly-linked lists. The first production rule recursively adds an element to an existing list introducing a new nonterminal  $SL$  in order to allow for insertion of elements by further hyperedge replacements. The second rule terminates the production by replacing the nonterminal  $SL$  with a concrete graph. Let us consider the hypergraph from Figure 2.3. We have two options to apply the hyperedge replacement grammar for singly-linked lists to the hypergraph under consideration. Applying the first rule yields the (abstract) hypergraph depicted in Figure 2.6, while applying the second rule yields a concrete hypergraph as shown in Figure 2.7.

Figures 2.6 and 2.7 display the forward application of production rules on a hypergraph, also called *concretisation*, as an abstract fragment is replaced by a (more) concrete subgraph. A concretisation step can yield more than one concrete hypergraph if several production rules are applicable. Therefore, concretisation needs to consider all possible hypergraphs. In fact, the application of the production rules of the hyperedge replacement grammar for singly-linked lists in Figures 2.6 and 2.7 is an example for a case where more than one rule is applicable.

In contrast to concretisation, *abstraction* describes the backward application of production rules such that a subgraph is replaced by a nonterminal. Abstraction of



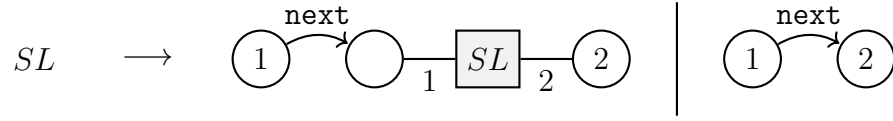


Figure 2.5: A hyperedge replacement grammar for singly-linked lists.

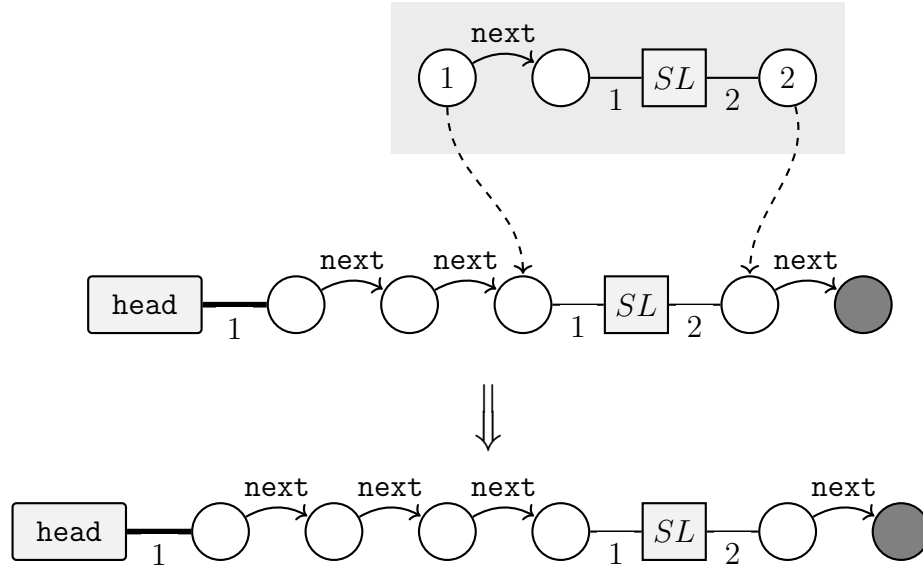


Figure 2.6: Applying the first production rule from Figure 2.5 adds an element to the list. An abstract hypergraph is obtained.

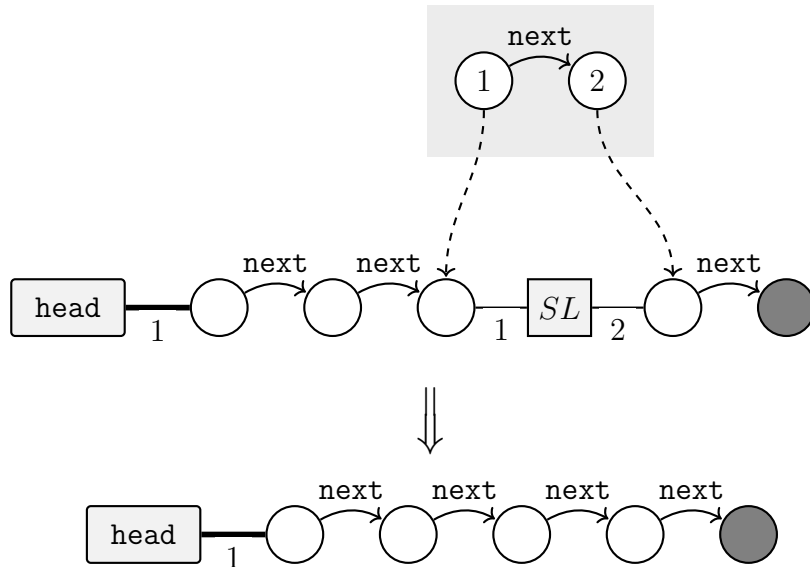


Figure 2.7: Applying the second production rule from Figure 2.5 yields a concrete hypergraph.

subgraphs yield an over-approximation of the current set of concrete hypergraphs, since information on what exactly has been abstracted is lost during abstraction. Abstraction allows us to represent possibly unboundedly large graphs in a finite manner. Examples for abstraction are given in Figures 2.6 and 2.7 if we consider a reversed application of the production rules. Thus, starting in the resulting hypergraphs in Figures 2.6 and 2.7 and following the reversed direction of the arrow, we obtain abstracted hypergraphs by applying the first and second rule in a backward fashion, respectively.

## 2.3 Transition Systems

Hypergraphs model the heap of a program at a certain moment., i.e., a single state during program execution. In order to describe the overall behavior of a program, we use the notion of a *transition system*. A transition system is a graph where nodes represent *states* reachable by a program and edges indicate *transitions* between states. A transition reflects how a state of the system changes, e.g., due to a statement execution. Therefore, the state space resulting from ATTESTOR's state space generation phase (see Figure 2.1) is a transition system, where the states compose of hypergraphs.

### Definition 2.3: Transition System [3]

A *transition system* is a tuple

$$T = (S, I, \delta, AP, L)$$

where

- $S$  is a set of states,
- $I \subseteq S$  is a set of initial states,
- $\delta \subseteq S \times S$  is a transition relation,
- $AP$  is a set of atomic propositions, and
- $L : S \rightarrow 2^{AP}$  is a labeling function.

$T$  is called *finite* if  $S$  and  $AP$  are finite.

We consider transition systems where a state is a hypergraph that encompasses the current heap of the program under consideration. The set  $AP$  of atomic propositions consists of the properties under consideration. The labeling function  $L$  maps a state  $s$  to a set  $L(s) \in 2^{AP}$  of atomic propositions satisfied by  $s$ . For example, the atomic

proposition " $i < 0$ " holds for a state where the variable  $i$  is set to the value  $-2$ . Based on the set  $L(s)$ , we can specify that  $s$  satisfies a propositional logic formula  $\phi$  if the evaluation induced by  $L(s)$  fulfills the formula  $\phi$ . Therefore,

$$s \models \phi \quad \text{iff} \quad L(s) \models \phi.$$

The transition relation  $\delta$  formally describes how the transition system  $T$  evolves starting in a state  $s \in S$ . Thus, the transition  $(s, s') \in \delta$  defines that state  $s$  evolves to state  $s'$ . If a state has more than one outgoing transition, the next transition is chosen nondeterministically. A state without any outgoing transitions is called a *terminal state*.

**Definition 2.4: Terminal State [3]**

A state  $s \in S$  in a transition system  $T = (S, I, \delta, AP, L)$  is called *terminal* if and only if

$$\nexists s' \in S. (s, s') \in \delta.$$

Figures 2.8 and 2.9 depict transition systems for the methods **traverse** and **noop** given in Listing 1.2, respectively. Let us exemplify the transition system  $T = (S, I, \delta, AP, L)$  for the method **traverse**. Every state encompasses a hypergraph that represents the heap at the current position in the analyzed program. The initial state  $s_0 \in I$  is marked with a short incoming arrow. It consists of a singly-linked list with two elements. The variable **head** is attached to the first node of the list.  $s_0$  is succeeded by state  $s_1$  which contains a further variable **current** that points to the same node as the variable **head**. The succeeding state  $s_2$  of  $s_1$  contains a self-loop, i.e.,  $s_2$  is its own successor. It also marks the final state of the transition system. The transition relation  $\delta$  is represented by arrows connecting two states. For example,  $(s_0, s_1) \in \delta$  and  $(s_1, s_2) \in \delta$ . The self-loop at state  $s_2$  depicts the transition  $(s_2, s_2) \in \delta$ . The transition system does not have any terminal states as all states have outgoing transitions. The states in  $T$  are labeled with atomic propositions that they satisfy, i.e., all states satisfy the proposition **identicNeighbours**, abbreviated as **idN**, that indicates that the order of the elements in the list is not modified. Therefore,  $L(s_0) = L(s_1) = L(s_2) = \{\text{identicNeighbours}\}$ . From the stated propositions, the set  $AP$  of atomic propositions is implicitly given as  $AP = \{\text{identicNeighbours}\}$ .

A sequence of states starting in an initial state  $s_0 \in I$  and either ending in a terminal state  $s \in S$  or infinitely prolonging, is called a *path* of transition system  $T$ .

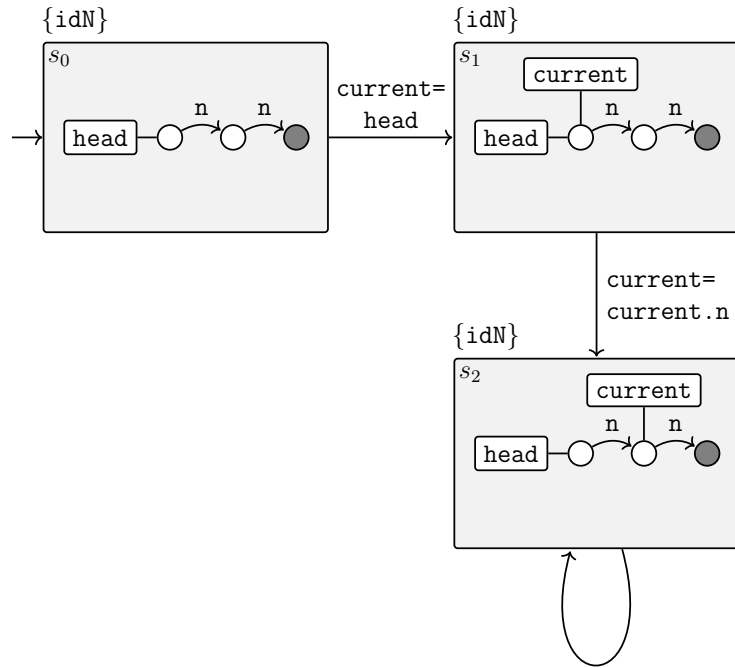


Figure 2.8: State space for method `SLList.traverse`.

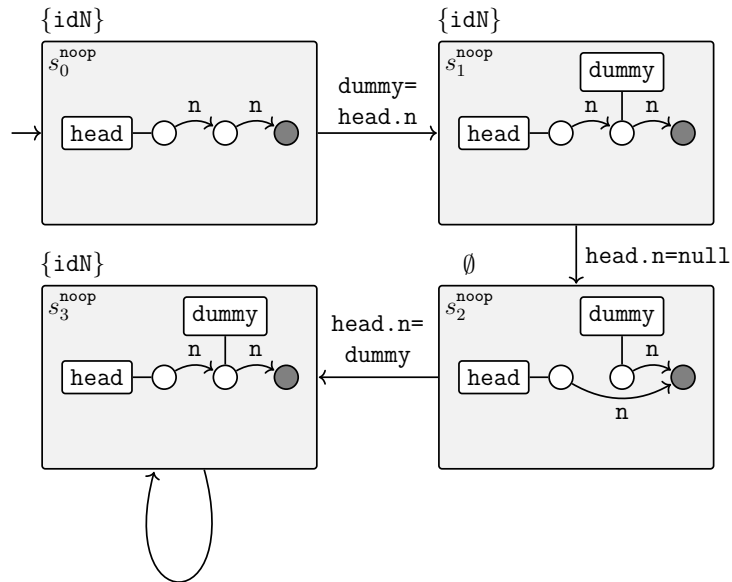


Figure 2.9: State space for method `SLList.noop`.

**Definition 2.5: Paths [3]**

A *finite path*  $\pi$  of a transition system  $T = (S, I, \delta, AP, L)$  is a finite sequence  $s_0 s_1 \dots s_n$  such that

- $s_0 \in I$  is an initial state,
- $s_i \in \{s \in S \mid (s_{i-1}, s) \in \delta\}$  for all  $0 < i \leq n$ , where  $n \geq 0$ , and
- $s_n$  is a terminal state.

An *infinite path*  $\pi'$  is an infinite sequence  $s_0 s_1 s_2 \dots$  such that

- $s_0 \in I$  is an initial state,
- $s_i \in \{s \in S \mid (s_{i-1}, s) \in \delta\}$  for all  $i > 0$ , and
- $s_i$  is not a terminal state for all  $i > 0$ .

$\text{Paths}(T)$  denotes the set of all paths in  $T$ .

For a path  $\pi$ ,  $\pi[i]$  denotes the  $i$ th state of  $\pi$ , while  $\pi[i..]$  denotes the  $i$ th suffix of  $\pi$ . Paths display the order of states that are traversed throughout a sequence of transitions. However, the related sets of atomic propositions of the traversed states, which are relevant for model checking, are not observable in the path itself. Therefore, we consider the notion of *traces* which are sequences of sets of atomic propositions that are satisfied along a path  $\pi$ . We regard transition systems without terminal states. This condition is not a restriction, since for every transition system an equivalent one without terminal states can be constructed by adding a state  $s_{\text{stop}}$  with a self-loop to the transition system to which all terminal states have an outgoing edge.

**Definition 2.6: Trace [3]**

Let  $T = (S, I, \delta, AP, L)$  be a transition system without terminal states. The *trace* of a finite path  $\pi = s_0 s_1 \dots s_n$  is defined as

$$\text{trace}(\pi) = L(s_0)L(s_1) \dots L(s_n).$$

Similarly, the trace of an infinite path  $\pi' = s_0 s_1 \dots$  is defined as

$$\text{trace}(\pi') = L(s_0)L(s_1) \dots$$

$\text{Traces}(s)$  denotes the set of traces of paths starting in state  $s$  and  $\text{Traces}(T)$  denotes the set of traces of the initial states of  $T$ .

Consider the sample transition system  $T$  in Figure 2.8. A path  $\pi$  in  $T$  is the finite sequence  $s_0s_1s_2$ . The corresponding trace of  $\pi$  is given by

$$\text{trace}(\pi) = L(s_0)L(s_1)L(s_2) = \{\text{idN}\}\{\text{idN}\}\{\text{idN}\}.$$

An infinite path  $\pi'$  in  $T$  is the sequence  $s_0s_1s_2s_2\dots$  that infinitely loops around the state  $s_2$ .  $\pi'$  implies the trace

$$\text{trace}(\pi') = L(s_0)L(s_1)L(s_2)L(s_2)\dots = \{\text{idN}\}\{\text{idN}\}\{\text{idN}\}\{\text{idN}\}\dots$$

## 2.4 Recursive State Machines

Often software programs contain method calls that induce call- and return-statements to different sections of the program. In order to capture the hierarchical structure of method invocation, we introduce *recursive state machines* defined by Alur et al. in [1]. Each method is encapsulated in its own *component* which consists of a set of *nodes* that are the states of the model and *boxes* that are each mapped to a component. A box represents an interface with entry and exit nodes that models the transition into another method environment, i.e., entering a box resembles a method invocation while exiting a box represents the return from a method execution. Edges between states and boxes identify transitions. Thus, a component state machine is a transition system for a single procedure while a recursive state machine composes a global transition system in which states can invoke procedures and hence enter different transition systems. We assume that the set  $AP$  of atomic propositions is implicitly defined by the context of the program under consideration.

### Definition 2.7: Recursive State Machine [1]

A *recursive state machine* (RSM)  $\mathcal{A}$  over a finite alphabet  $\Sigma$  is given by a tuple  $(A_1, \dots, A_k)$ , where each *component state machine* is a tuple

$$A_i = (S_i, En_i, Ex_i, \delta_i, L_i),$$

$1 \leq i \leq k$ , where

- $S_i = N_i \cup B_i$  is a set of states composing of a set  $N_i$  of *nodes* and a (disjoint) set  $B_i$  of *boxes*,
- $En_i \subseteq N_i$  is a set of *entry nodes*,
- $Ex_i \subseteq N_i$  is a set of *exit nodes*,
- $\delta_i$  is a *transition relation* with transitions of the form  $(u, \sigma, v)$ , where

- the source  $u$  is either a node of  $N_i$  or a pair  $(b, x)$ , where  $b$  is a box in  $B_i$  and  $x$  is an exit node in  $Ex_j$  for  $j = L_i(b)$ ,
  - the label  $\sigma$  is in  $\Sigma$ , and
  - the destination  $v$  is either a node in  $N_i$  or a pair  $(b, e)$ , where  $b$  is a box in  $B_i$  and  $e$  is an entry node in  $En_j$  for  $j = L_i(b)$ , and
- $L_i : B_i \mapsto \{1, \dots, k\}$  is a *labeling* that assigns to every box an index  $j \in \{1, \dots, k\}$  referring to one of the component state machines  $A_1, \dots, A_k$ .

A sample RSM with three components  $A_1, A_2, A_3$  is depicted in Figure 2.10. The nodes drawn at the border of the components represent the entry and exit nodes. Directed edges depict the transitions between states of a component as well as between states and boxes. Each box is mapped to a component, e.g. box  $b_1$  in component  $A_1$  is mapped to component  $A_2$ . Box  $c_2$  in component  $A_2$  is mapped to component  $A_3$ . Thus, entering box  $b_1$  or  $c_2$  changes the current component under control from component  $A_1$  to  $A_2$  and from  $A_2$  to  $A_3$ , respectively. The change in components represents an invocation of a program method where entry nodes represent input arguments to the called method and exit nodes model return values.

In order to define the execution of an RSM  $\mathcal{A} = (A_1, \dots, A_k)$ , we first describe the global relation between its components  $A_i, 1 \leq i \leq k$ . A *global state* of an RSM is a sequence of boxes ending in a node of a component.

**Definition 2.8: Global State [1]**

A *global state* of an RSM  $\mathcal{A} = (A_1, \dots, A_k)$  is a tuple  $(b_1, \dots, b_r, u)$ , where  $b_1 \in B_1, \dots, b_r \in B_r$  are boxes and  $u$  is a node. The set  $S$  of global states of  $\mathcal{A}$  is  $B^*N$ , where  $B = \bigcup_i B_i$  and  $N = \bigcup_i N_i$ . A state  $(b_1, \dots, b_r, u)$  with  $b_i \in B_{j_i}$  for  $1 \leq i \leq r$  and  $u \in N_j$  is *well-formed* if  $L_{j_i}(b_i) = j_{i+1}$  for  $1 \leq i < r$  and  $L_{j_r}(b_r) = j$ .

A well-formed state  $(b_1, \dots, b_r, u)$  of an RSM  $\mathcal{A}$  corresponds to a path through the components  $A_j$  of  $\mathcal{A}$ , where we enter component  $A_j$  via box  $b_r$  of component  $A_{j_r}$ .

Consider the sample RSM given in Figure 2.10. A global state in the sample RSM is given by

$$(b_1, c_1, c_1, c_2, d, b_2, d, u_3),$$

where the sequence of boxes reflects which components are visited before reaching the current state  $u_3$ . The state is well-formed as for every box  $b \in \{b_1, b_2, c_1, c_2, d\}$  the labeling function  $L_i$  coincides with the component referred to by the next box in the sequence, e.g.,  $L_1(b_1) = A_2$ , which is exactly the component in which the

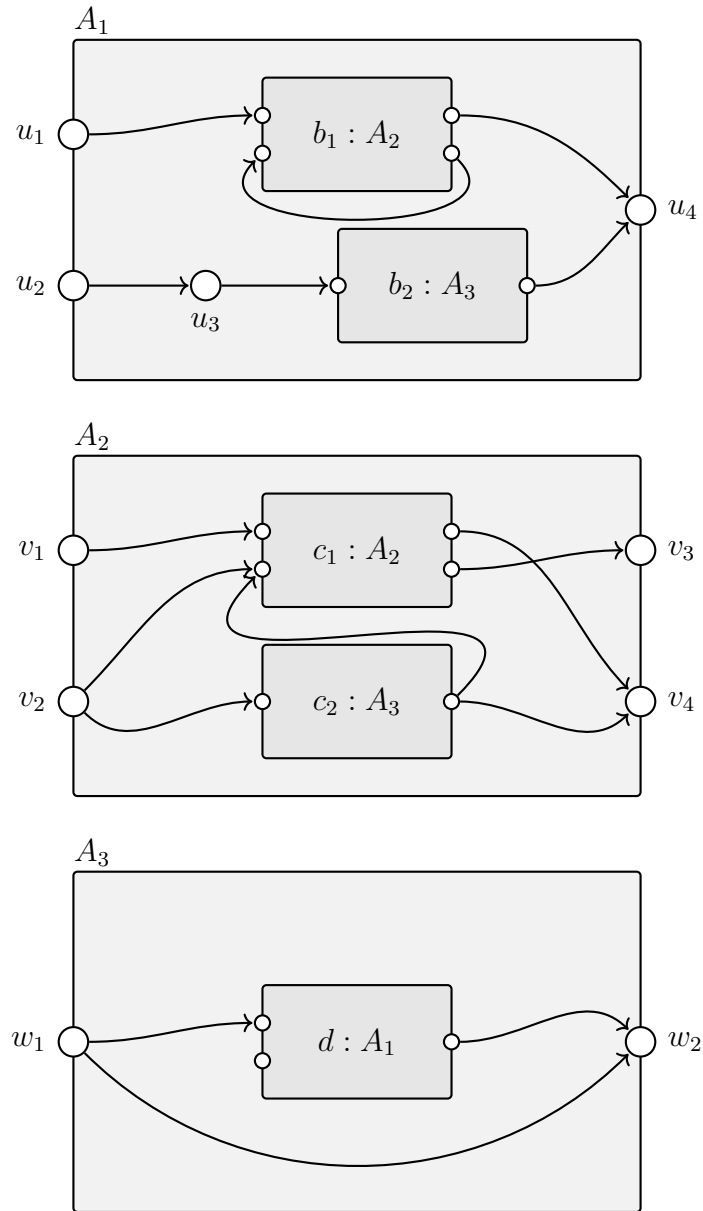


Figure 2.10: A sample recursive state machine [1].



following box  $c_1$  is defined.

In order to move between global states of an RSM  $\mathcal{A}$ , we require the notion of the *global transition relation*  $\delta$  which enables us to not only transition between states within a component  $A_j$  as defined by its transition relation  $\delta_j$ , but also between pairs of components.

**Definition 2.9: Global Transition Relation [1]**

Let  $s = (b_1, \dots, b_r, u) \in S$  be a state with  $u \in N_j$  and  $b_r \in B_m$  for an RSM  $\mathcal{A} = (A_1, \dots, A_k)$ . The *global transition relation*  $\delta$  for  $\mathcal{A}$  defines  $(s, \sigma, s') \in \delta$  if and only if one of the following holds:

1.  $(u, \sigma, u') \in \delta_j$  for a node  $u'$  of  $A_j$  and  $s' = (b_1, \dots, b_r, u')$ .
2.  $(u, \sigma, (b', e)) \in \delta_j$  for a box  $b'$  of  $A_j$  and  $s' = (b_1, \dots, b_r, b', e)$ .
3.  $u$  is an exit-node of  $A_j$ ,  $((b_r, u), \sigma, u') \in \delta_m$  for a node  $u'$  of  $A_m$ , and  $s' = (b_1, \dots, b_{r-1}, u')$ .
4.  $u$  is an exit-node of  $A_j$ ,  $((b_r, u), \sigma, (b', e)) \in \delta_m$  for a box  $b'$  of  $A_m$ , and  $s' = (b_1, \dots, b_{r-1}, b', e)$ .

Definition 2.9 specifies the possible kinds of transitions between global states  $s, s' \in S$  of an RSM  $\mathcal{A}$ . Again, consider the RSM given in Figure 2.10. For each case depicted in Definition 2.9, we illustrate the global transition relation:

Case 1 describes the scenario where the source and the destination nodes are both within the same component  $A_j$ . For instance, the component  $A_1$  defines  $(u_2, \sigma, u_3) \in \delta_1$ , thus, in terms of the global transition relation  $\delta$ , a valid global transition is  $((b_2, d, u_2), \sigma, (b_2, d, u_3)) \in \delta$ .

Case 2 depicts that a new component is entered via box  $b'$  of  $A_j$ . Thus, the current node of the destination state  $s'$  is the entry-node  $e$ . An example for this case is given by the global state  $(b_1, c_1, c_2, d, u_3)$  which is located in component  $A_1$ . The local transition relation  $\delta_1$  contains the transition  $(u_3, \sigma, (b_2, v_1))$ . Therefore, globally  $((b_1, c_1, c_2, d, u_3), \sigma, (b_1, c_1, c_2, d, b_2, v_1)) \in \delta$  which corresponds to entering component  $A_2$  via box  $b_2$  and transitioning to state  $(b_1, c_1, c_2, d, b_2, v_1)$ .

Case 3 and 4 are both exiting component  $A_j$  via the exit-node  $u$ . While case 3 returns to component  $A_m$ , from where we entered  $A_j$  before, case 4 directly enters a new component via box  $b'$  of component  $A_m$ . An example for case 3 is given by the transition  $((b_1, c_1, c_2, d, u_4), \sigma, (b_1, c_1, c_2, w_2)) \in \delta$ , where we return from component  $A_1$  via box  $d$  to component  $A_3$ . If we continue the return action for state

$(b_1, c_1, c_2, w_2)$ , we get  $((b_1, c_1, c_2, w_2), \sigma, (b_1, c_1, c_1, v_2)) \in \delta$  as a sample transition for case 4. The transition describes that we exit component  $A_3$  entered via box  $c_2$  and directly enter component  $A_2$  via box  $c_1$  as  $((c_2, w_2), \sigma, (c_1, v_2))$  is a valid transition according to  $\delta_2$ .

After defining the terms of global states and the global transition relation for an RSM  $\mathcal{A}$ , we can summarize these components together with the finite alphabet  $\Sigma$  within the concept of a *labeled transition system* (LTS)  $T_{\mathcal{A}}$  induced by  $\mathcal{A}$ . The LTS encodes the execution of  $\mathcal{A}$ .

**Definition 2.10: Labeled Transition System induced by an RSM [1]**

For an RSM  $\mathcal{A} = (A_1, \dots, A_k)$ , the induced *labeled transition system* (LTS) is a tuple

$$T_{\mathcal{A}} = (S, \Sigma, \delta)$$

where

- $S$  is the set of global states,
- $\Sigma$  is the finite alphabet, and
- $\delta$  is the global transition relation.

The LTS of an RSM is basically the flattening of the hierarchical structure induced by the components and boxes of an RSM. Therefore, an LTS corresponds to our initial definition of a transition system, where the set  $I$  of initial states, the set  $AP$  of atomic propositions, and the labeling function  $L$  are implicitly specified by the underlying RSM (cf. Definition 2.3). The notion of paths and traces of transition systems also carry over to LTS. Thus, traces of an RSM are the traces of the corresponding LTS.

We specified the relevant framework for model checking pointer-manipulating programs. The program execution is modeled by recursive state machines capturing the hierarchical nature of method calls, while hypergraphs offer a finite representation of heaps that constitute the states of the model under consideration. Another ingredient to model checking is the formal definition of the properties the model is to be validated for. Therefore, we focus on *linear temporal logic* described in the following section.

## 2.5 Linear Temporal Logic

First proposed by Pnueli in 1977, *linear temporal logic* (LTL) is a logic that describes *linear-time properties*. Linear-time properties specify requirements on paths (or

rather their traces) and can be understood as a set of (infinite) words over a set  $AP$  of atomic propositions.

**Definition 2.11: Linear-Time Property [3]**

A *linear-time property* over the set of atomic propositions  $AP$  is a subset of the set  $(2^{AP})^\omega$ , i.e., all infinite words defined over the alphabet  $2^{AP}$ .

The satisfaction relation  $\models$  for linear-time properties defines that a transition system  $T$  satisfies a linear-time property  $P \subseteq (2^{AP})^\omega$  if and only if all traces of  $T$  are included in the set  $P$ , i.e., every trace of  $T$  is a word in the language induced by  $P$ .

**Definition 2.12: Satisfaction Relation for Linear-Time Properties [3]**

Let  $P$  be a linear-time property over  $AP$  and  $T = (S, I, \delta, AP, L)$  a transition system. Then,  $T$  *satisfies*  $P$ , denoted  $T \models P$ , if and only if  $\text{Traces}(T) \subseteq P$ . A state  $s \in S$  satisfies  $P$ , denoted  $s \models P$ , if and only if  $\text{Traces}(s) \subseteq P$ .

Linear-time properties can be specified by LTL formulae that encode temporal specifications for paths. LTL formulae are composed of three components: the Boolean operators *negation* ( $\neg$ ) and *conjunction* ( $\wedge$ ), the temporal operators *next* ( $\bigcirc$ ) and *until* ( $\mathbf{U}$ ), and a set of atomic propositions  $AP$ . Atomic propositions are state labels of a transition system, which express properties that hold for a single state, e.g., " $i = 1$ ". Formally, the syntax of LTL formulae is defined as follows:

**Definition 2.13: Syntax of LTL [3]**

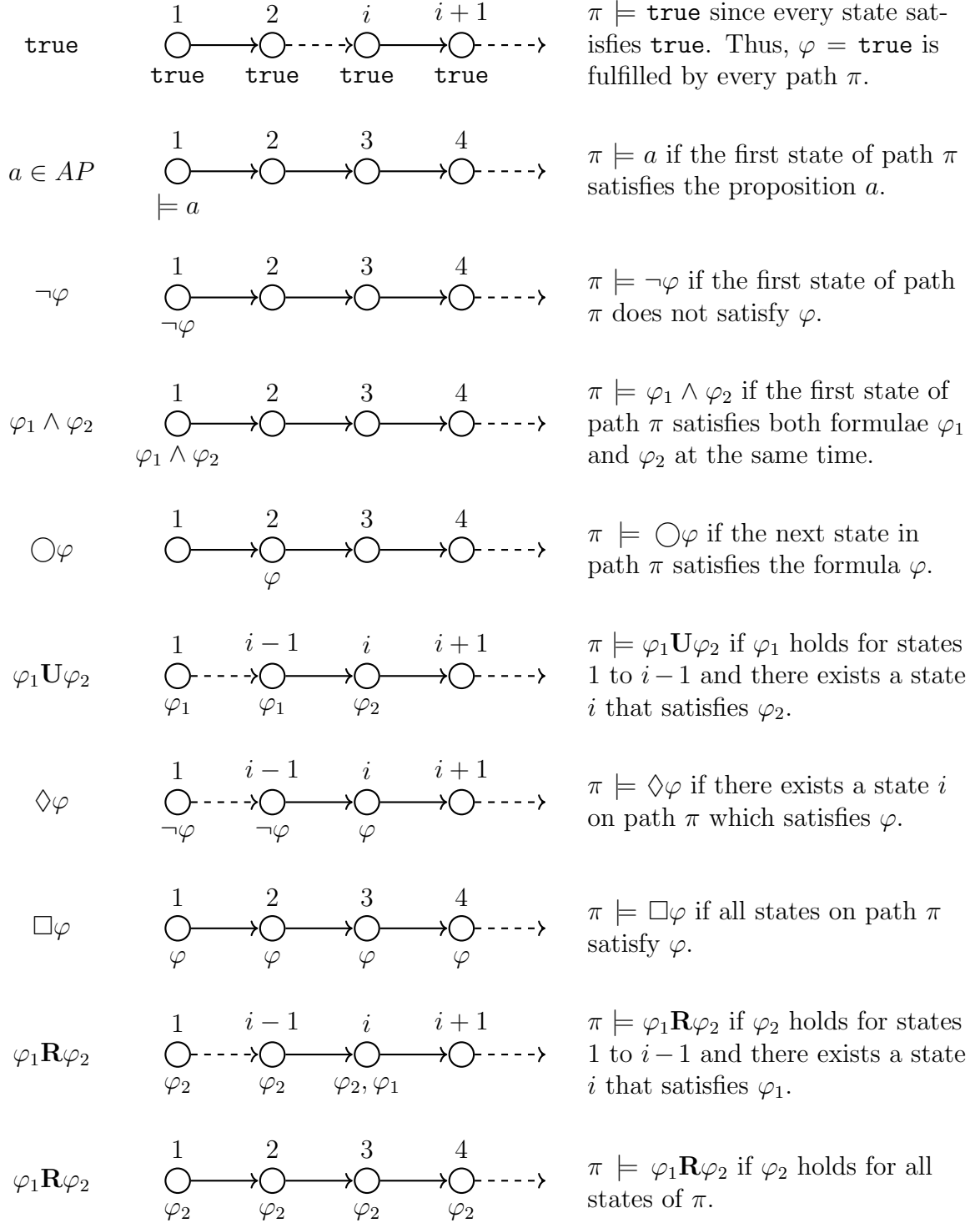
Given a set  $AP$  of atomic propositions with  $a \in AP$ , *LTL formulae* are given by the context-free grammar below:

$$\varphi := \text{true} \mid a \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \bigcirc\varphi \mid \varphi_1 \mathbf{U} \varphi_2.$$

Further temporal operators that are commonly used, but are not included in the definition of LTL formulae, are the temporal modalities *eventually* ( $\Diamond$ ), *globally* ( $\Box$ ), and *release* ( $\mathbf{R}$ ). They are derived by the operators given in Definition 2.13 by

$$\begin{aligned} \Diamond\varphi &:= \text{true} \mathbf{U} \varphi \\ \Box\varphi &:= \neg\Diamond\neg\varphi \\ \varphi_1 \mathbf{R} \varphi_2 &:= \neg(\neg\varphi_1 \mathbf{U} \neg\varphi_2). \end{aligned}$$

We constitute an intuitive understanding of temporal operators by visualizing their semantics in Figure 2.11.

Figure 2.11: Intuitive semantics of temporal operators for a path  $\pi$ .

The following definition captures the relation between linear-time properties and LTL formulae as the latter can be interpreted as words over the alphabet  $2^{AP}$ .

**Definition 2.14: Semantics of LTL (Interpretation over Words) [3]**

Let  $\varphi$  be an LTL formula over  $AP$ . The linear-time property induced by  $\varphi$  is

$$\text{Words}(\varphi) = \{\sigma \in (2^{AP})^\omega \mid \sigma \models \varphi\}$$

where the satisfaction relation  $\models \subseteq (2^{AP})^\omega \times LTL$  is the smallest relation with the following properties:

$$\begin{aligned} \sigma &\models \text{true} \\ \sigma &\models a && \text{iff } a \in A_0, \text{ where } \sigma = A_0 A_1 A_2 \dots \\ \sigma &\models \varphi_1 \wedge \varphi_2 && \text{iff } \sigma \models \varphi_1 \text{ and } \sigma \models \varphi_2 \\ \sigma &\models \neg \varphi && \text{iff } \sigma \not\models \varphi \\ \sigma &\models \bigcirc \varphi && \text{iff } \sigma[1 \dots] = A_1 A_2 A_3 \dots \models \varphi \\ \sigma &\models \varphi_1 \mathbf{U} \varphi_2 && \text{iff } \exists j \geq 0. \sigma[j \dots] \models \varphi_2 \text{ and } \sigma[i \dots] \models \varphi_1, \forall 0 \leq i < j. \end{aligned}$$

The interpretation of LTL formulae over words describes the semantics of LTL formulae over paths and states of a transition system  $T$ .

**Definition 2.15: Semantics of LTL over Paths and States [3]**

Let  $T = (S, I, \delta, AP, L)$  be a transition system without terminal states and let  $\varphi$  be an LTL-formula over  $AP$ . For an infinite path  $\pi$  of  $T$ , the satisfaction relation is defined by

$$\pi \models \varphi \quad \text{iff} \quad \text{trace}(\pi) \models \varphi.$$

For a state  $s \in S$ , the satisfaction relation  $\models$  is defined by

$$s \models \varphi \quad \text{iff} \quad (\forall \pi \in \text{Paths}(T). \pi \models \varphi).$$

$T$  satisfies  $\varphi$ , denoted  $T \models \varphi$ , if  $\text{Traces}(T) \subseteq \text{Words}(\varphi)$ .

From Definition 2.15, it follows that

$$T \models \varphi \quad \text{iff} \quad \forall s_0 \in I. s_0 \models \varphi.$$

Based on the satisfaction relation of LTL formulae over paths and states, we specify the semantics of LTL for a transition system  $T$ .

**Definition 2.16: Semantics of LTL [3]**

Given an LTL formula  $\varphi$ , a concrete transition system  $T$ , and a path  $\pi \in \text{Paths}(T)$ , the model relation  $\models$  for LTL formulae is defined by

$$\begin{aligned}
 \pi &\models \text{true} \\
 \pi &\models a && \Leftrightarrow \pi[1] \models a \\
 \pi &\models \neg\varphi && \Leftrightarrow \text{not } \pi[1] \models \varphi \\
 \pi &\models \varphi_1 \wedge \varphi_2 && \Leftrightarrow (\pi \models \varphi_1) \text{ and } (\pi \models \varphi_2) \\
 \pi &\models \bigcirc\varphi && \Leftrightarrow \pi[2..] \models \varphi \\
 \pi &\models \varphi_1 \mathbf{U} \varphi_2 && \Leftrightarrow \exists i \geq 1. (\pi[i..] \models \varphi_2 \wedge (\forall 1 \leq k < i. \pi[k..] \models \varphi_1)).
 \end{aligned}$$

Given a state  $s \in S$ ,  $s \models \varphi$  if for all  $\pi \in \text{Paths}(T)$  it holds that  $\pi \models \varphi$ . For a transition system  $T$ ,  $T \models \varphi$  if for all  $\pi \in \text{Paths}(T)$  it holds that  $\pi \models \varphi$ .

For the operators *eventually* ( $\Diamond$ ), *globally* ( $\Box$ ), and *release* ( $\mathbf{R}$ ), the semantics are defined similarly:

$$\begin{aligned}
 \pi &\models \Diamond\varphi && \Leftrightarrow \exists i \geq 1. \pi[i..] \models \varphi \\
 \pi &\models \Box\varphi && \Leftrightarrow \forall i \geq 1. \pi[i..] \models \varphi \\
 \pi &\models \varphi_1 \mathbf{R} \varphi_2 && \Leftrightarrow \forall i \geq 1. \pi[i..] \models \varphi_2 \text{ or } \\
 &&& \exists i \geq 1. (\pi[i..] \models \varphi_1 \wedge (\forall 1 \leq k < i. \pi[k..] \models \varphi_2)).
 \end{aligned}$$

The following LTL formulae are examples for specifying properties for model checking pointer-manipulating programs. The formula

$$\bigcirc\{\text{SLList}\}$$

employs the  $\bigcirc$  operator and the atomic proposition **SLList**, describing that the heap is a singly-linked list, to express that the shape of the heap of the next state is a singly-linked list. Another example is the formula

$$\Box\{\text{SLList}\}$$

which requires the heap of every state to be a singly-linked list. Thus, any state not satisfying the atomic proposition **SLList** falsifies the formula  $\Box\{\text{SLList}\}$ . The formula

$$\Box\Diamond\{\text{terminated}\} \rightarrow \Box\Diamond\{\text{SLList}\}$$

includes another atomic proposition, **terminated**, that describes that a state is a terminating state. Thus, the above formula states that the heap is a singly-linked list upon termination of the analyzed program.

Two LTL formulae are semantically equivalent if they evaluate to the same results under all interpretations. For every LTL formula, there exists an equivalent formula in *positive normal form*, where negations are only allowed on the level of literals [3].

**Definition 2.17: Positive Normal Form [3]**

Given a set  $AP$  of atomic propositions with  $a \in AP$ , LTL formulae in *positive normal form* (PNF) are defined by

$$\varphi := \text{true} \mid \text{false} \mid a \mid \neg a \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \bigcirc \varphi \mid \varphi_1 \mathbf{U} \varphi_2 \mid \varphi_1 \mathbf{R} \varphi_2.$$

The following theorem states the existence of an equivalent formula in PNF for every LTL formula.

**Theorem 2.18: Existence of Equivalent PNF Formulae [3]**

For each LTL formula there exists an equivalent LTL formula in PNF.

The existence of an equivalent PNF formula for every LTL formula is due to the following equivalences that allow to push negations inside [7]:

$$\begin{aligned} \neg \neg \varphi &= \varphi \\ \neg \text{false} &= \text{true} \\ \neg(\varphi_1 \wedge \varphi_2) &= \neg \varphi_1 \vee \neg \varphi_2 \\ \neg \bigcirc \varphi &= \bigcirc \neg \varphi \\ \neg(\varphi_1 \mathbf{U} \varphi_2) &= \neg \varphi_1 \mathbf{R} \neg \varphi_2 \end{aligned}$$

As an example, consider the LTL formula

$$\Box \Diamond \{\text{terminated}\} \rightarrow \Box \Diamond \{\text{SLList}\}$$

where **terminated** and **SLList** are atomic propositions as described above. An equivalent formula in PNF is achieved by the following equivalences:

$$\begin{aligned}
& \Box\Diamond\{\text{terminated}\} \rightarrow \Box\Diamond\{\text{SLList}\} \\
\equiv & \neg(\Box\Diamond\{\text{terminated}\}) \vee (\Box\Diamond\{\text{SLList}\}) && \text{(definition of } \rightarrow \text{)} \\
\equiv & \neg(\Box(\text{true } \mathbf{U} \{\text{terminated}\})) && \\
& \vee (\Box(\text{true } \mathbf{U} \{\text{SLList}\})) && \text{(definition of } \Diamond \text{)} \\
\equiv & \neg(\neg\Diamond\neg(\text{true } \mathbf{U} \{\text{terminated}\})) && \\
& \vee (\neg\Diamond\neg(\text{true } \mathbf{U} \{\text{SLList}\})) && \text{(definition of } \Box \text{)} \\
\equiv & \neg(\neg(\text{true } \mathbf{U} \neg(\text{true } \mathbf{U} \{\text{terminated}\}))) && \\
& \vee (\neg(\text{true } \mathbf{U} \neg(\text{true } \mathbf{U} \{\text{SLList}\}))) && \text{(definition of } \Diamond \text{)} \\
\equiv & \neg(\neg\text{true } \mathbf{R} (\text{true } \mathbf{U} \{\text{terminated}\})) && \\
& \vee (\neg\text{true } \mathbf{R} (\text{true } \mathbf{U} \{\text{SLList}\})) && \text{(duality of } \mathbf{U} \text{ and } \mathbf{R} \text{)} \\
\equiv & \text{true } \mathbf{U} (\neg\text{true } \mathbf{R} \neg\{\text{terminated}\}) && \\
& \vee (\neg\text{true } \mathbf{R} (\text{true } \mathbf{U} \{\text{SLList}\})). && \text{(duality of } \mathbf{U} \text{ and } \mathbf{R} \text{)}
\end{aligned}$$



## Chapter 3

# Hierarchical Model Checking

A challenge in model checking programs with method calls is that the underlying state space of procedural programs is a hierarchical one, where each procedure contributes an own state space, denoted as a *procedure state space*. In contrast to top level model checking, several state spaces need to be generated and model checked. Therefore, it is desirable to verify procedure state spaces in a hierarchical setting as efficient as possible such that repetitive computations are avoided. One solution to hierarchical model checking is to flatten the hierarchical state space by removing the hierarchy. This corresponds to an edge connecting the calling state of a procedure with the "entry" state of the procedure state space. For the flat setting, we introduce the tableaux construction for model checking LTL properties by Grumberg et al. [4]. Since flattening a hierarchical state space is not very efficient and might result in unboundedly large state spaces, we introduce modified approaches of the tableaux construction to hierarchical model checking in Chapters 4 and 5. Both algorithms avoid checking a procedure state space multiple times for the same set of LTL formulae by implementing *model checking contracts* that store previously computed results. We present this concept in the second part of this chapter.

### 3.1 Tableaux Construction

The on-the-fly model checking approach by Grumberg et al. [4] constructs a *proof structure* based on a set of *tableaux rules* in order to verify whether a transition system  $T$  satisfies an LTL formula  $\varphi$  in PNF. The condition that LTL formulae are in PNF is not a restriction due to Theorem 2.18. A proof structure is a directed graph  $(V, E)$ , where the set of vertices  $V$  are composed of *assertions* and the set  $E$  of edges contains the edge  $(\lambda_1, \lambda_2)$  between two assertions  $\lambda_1$  and  $\lambda_2$  if the underlying tableaux contains the inference rule  $\frac{\lambda_1}{\lambda_2}$ .

**Definition 3.1: Proof Structure [4]**

A *proof structure* for  $\lambda \in \Lambda$  is a tuple  $(V, E)$  with  $V \subseteq (\Lambda \cup \mathbf{true})$  and  $E \subseteq V \times V$ , such that for any  $\lambda'$  it holds that  $\lambda'$  is reachable from  $\lambda$  and that the successors of  $\lambda'$  are the ones that result from applying some of the rules, i.e.

$$(\lambda_1, \lambda_2) \in E \quad \text{iff} \quad \frac{\lambda_1}{\lambda_2 \quad s \dots}.$$

The underlying tableaux rules for checking LTL formulae are specified in Figure 3.1. They model the semantics of LTL. The rules for the operators **U** and **R** follow from the expansion law for LTL formulae [3]. Accordingly,

$$\varphi_1 \mathbf{U} \varphi_2 \equiv \varphi_2 \vee (\varphi_1 \wedge \bigcirc(\varphi_1 \mathbf{U} \varphi_2))$$

and

$$\varphi_1 \mathbf{R} \varphi_2 \equiv \varphi_2 \wedge (\varphi_1 \vee \bigcirc(\varphi_1 \mathbf{R} \varphi_2)).$$

A vertex in the set  $V$  of a proof structure  $(V, E)$  is an assertion of the form  $s \vdash \Phi$ , where  $s$  is a state in  $T$  and  $\Phi$  is a set of LTL formulae. An assertion  $s \vdash \Phi$  holds if at least one formula  $\varphi \in \Phi$  is satisfied by the state  $s$ . Thus, an assertion is a verification goal that aims at proving that  $s \models \bigvee_{\varphi \in \Phi} \varphi$ . Therefore, an assertion is split into subgoals according to the tableaux rules. By proving a sequence of subgoals the validity of the initial assertion  $\lambda$  can be concluded from the validity of the subgoals. Hence, the proof structure of an assertion  $\lambda$  contains all subgoals of  $\lambda$ .

The rules  $(R^{\mathbf{U}})$  and  $(R^{\mathbf{R}})$  can introduce cycles into a proof structure if  $\varphi_1$  is fulfilled for every state in the underlying transition system for formulae of the form  $\varphi_1 \mathbf{U} \varphi_2$  or  $\varphi_1 \mathbf{R} \varphi_2$ , while no state fulfills  $\varphi_2$ . Therefore, a cycle in a proof structure represents an *infinite path* in the underlying state space. In an infinite path,  $\varphi_1 \mathbf{U} \varphi_2$  can never be fulfilled, whereas  $\varphi_1 \mathbf{R} \varphi_2$  is fulfilled according to the definition of **R**. Consequently, a cycle in the proof structure originating from successively applying rule  $(R^{\mathbf{U}})$  evaluates to a violated assertion, while a cycle arising from applying rule  $(R^{\mathbf{R}})$  fulfills the subgoal. The other tableaux rules cannot introduce infinite paths as their application reduces the size of the formulae.

For a proof structure  $(V, E)$  for an assertion  $\lambda$ , we define the following relations according to [4]:

- If  $s \vdash \emptyset \in V$ , then  $(V, E)$  is *unsuccessful* as an empty assertion can never be fulfilled.
- $\lambda \in V$  is a leaf of the proof structure if there is no  $\lambda' \in V$  with  $(\lambda, \lambda') \in E$ . A leaf  $\lambda$  is called *successful* if  $\lambda = \mathbf{true}$ .

$$\begin{aligned}
(R^{\models}) \quad & \frac{s \vdash \Phi \cup \{a\}}{\mathbf{true}} \quad \text{if } s \models a \\
(R^{\not\models}) \quad & \frac{s \vdash \Phi \cup \{a\}}{s \vdash \Phi} \quad \text{if } s \not\models a \\
(R^{\vee}) \quad & \frac{s \vdash \Phi \cup \{\varphi_1 \vee \varphi_2\}}{s \vdash \Phi \cup \{\varphi_1\} \cup \{\varphi_2\}} \\
(R^{\wedge}) \quad & \frac{s \vdash \Phi \cup \{\varphi_1 \wedge \varphi_2\}}{s \vdash \Phi \cup \{\varphi_1\} \quad s \vdash \Phi \cup \{\varphi_2\}} \\
(R^{\mathbf{U}}) \quad & \frac{s \vdash \Phi \cup \{\varphi_1 \mathbf{U} \varphi_2\}}{s \vdash \Phi \cup \{\varphi_2, \varphi_1\} \quad s \vdash \Phi \cup \{\varphi_2, \bigcirc(\varphi_1 \mathbf{U} \varphi_2)\}} \\
(R^{\mathbf{R}}) \quad & \frac{s \vdash \Phi \cup \{\varphi_1 \mathbf{R} \varphi_2\}}{s \vdash \Phi \cup \{\varphi_2\} \quad s \vdash \Phi \cup \{\varphi_1, \bigcirc(\varphi_1 \mathbf{R} \varphi_2)\}} \\
(R^{\bigcirc}) \quad & \frac{s \vdash \{\bigcirc\varphi_1, \dots, \bigcirc\varphi_n\}}{s_1 \vdash \{\varphi_1, \dots, \varphi_n\} \quad \dots \quad s_m \vdash \{\varphi_1, \dots, \varphi_n\}}
\end{aligned}$$

Figure 3.1: Tableaux rules for LTL model checking.

- An infinite path  $\lambda_1 \lambda_2 \dots$  in  $(V, E)$  is called successful if and only if there exists a position  $i \in \mathbb{N}$  with  $\varphi_1 \mathbf{R} \varphi_2 \in \lambda_i$  and for all  $j \geq i$  it holds that  $\varphi \notin \lambda_j$ .
- The proof structure  $(V, E)$  is called successful if every leaf as well as every of its infinite paths is successful.

Theorem 3.2 states that the tableaux construction is indeed a suitable procedure to model check a transition system  $T$  for an LTL formula  $\varphi$  as the success of the proof structure  $s \vdash \{\varphi\}$  for a state  $s$  in  $T$  coincides with the validity of  $T \models \varphi$ .

**Theorem 3.2: Correctness of the Tableaux Construction [4]**

For a concrete transition system  $T = (S, I, \delta, AP, L)$  with  $s \in S$  and an LTL formula  $\varphi$ , let  $(V, E)$  be the proof structure for  $s \vdash \{\varphi\}$ . Then it holds that  $s \models \varphi$  if and only if  $(V, E)$  is successful.

In order to illustrate the tableaux construction algorithm, consider the method **reverse** for reversing a singly-linked list given in Listing 3.1.

```

1 public static SLList reverse(SLList head) {
2
3     SLList revList = null;
4     SLList current = head;
5
6     while (current != null) {
7         SLList next = current.next;
8         current.next = revList;
9         revList = current;
10        current = next;
11    }
12    return revList;
13 }

```

Listing 3.1: JAVA method for reversing a singly-linked list.

Figure 3.2 shows the state space of reversing a singly-linked list with two elements according to the method `reverse`. For this state space, we employ tableaux construction to check whether the formula  $\varphi = \Box\Diamond\{\text{SLList}\}$  is satisfied.  $\varphi$  states that the heap will always be a singly-linked list. An equivalent formula to  $\varphi$  in PNF is

$$(\neg \text{true } \mathbf{R} (\text{true } \mathbf{U} \{\text{SLList}\})).$$

The proof structure is depicted in Figure 3.3. It starts with the initial formula  $\varphi$  in PNF and the initial state  $s_0$  of the state space, i.e.,  $\lambda_0 = (s_0 \vdash \{\varphi\})$  is the first assertion of the proof structure. Applying the rule  $(R^{\mathbf{R}})$  to  $\lambda_0$ , we split  $\lambda_0$  into two assertions that are connected to the root assertion via directed edges. Thereupon, the rules  $(R^{\mathbf{U}})$  and  $(R^{\neq})$  are applied to the resulting assertions  $s_0 \vdash \{\text{true } \mathbf{U} \{\text{SLList}\}\}$  and  $s_0 \vdash \{\neg \text{true}, \bigcirc\varphi\}$ , respectively. While the first branch terminates in a leaf with the Boolean value `true`, the second branch expands the proof structure for successor states of  $s_0$ , i.e.,  $s_1$ , by applying the rule  $(R^{\bigcirc})$ . Thus,  $\varphi$  is validated for state  $s_1$ . For the sake of readability, the rule applications for states  $s_2, \dots, s_9$  are omitted as they are analogous to the rule application for state  $s_0$ . All paths ending in the leaf `true` are successful. For state  $s_{10}$ , the proof structure induces two infinite paths: the infinite path  $\pi_1$  looping around the sets of LTL formulae  $\{\varphi\}$ ,  $\{\neg \text{true}, \bigcirc\varphi\}$ , and  $\{\bigcirc\varphi\}$ , and the infinite path  $\pi_2$  looping around the sets  $\{\text{true } \mathbf{U} \{\text{SLList}\}\}$ ,  $\{\text{SLList}, \bigcirc(\text{true } \mathbf{U} \{\text{SLList}\})\}$ , and  $\bigcirc(\text{true } \mathbf{U} \{\text{SLList}\})$ . The loop in  $\pi_2$  is marked red. As the sets of formulae in the loop on path  $\pi_1$  contain an  $\mathbf{R}$ -operator,  $\pi_1$  is considered as successful. Opposed to this, the sets of formulae in the loop on path  $\pi_2$  do not contain an  $\mathbf{R}$ -operator, but originate from applying the rule  $(R^{\mathbf{U}})$ . Thus,  $\pi_2$  is not successful. Therefore, the proof structure for  $\lambda_0$  is unsuccessful concluding that the state space, i.e., the method `reverse`, does not satisfy the LTL formula  $\neg \text{true } \mathbf{R} (\text{true } \mathbf{U} \{\text{SLList}\})$ .

The tableaux construction can be combined with an on-the-fly state space generation such that not all states need to be computed if not required during the run of the tableaux construction. Consider the LTL formula  $\bigcirc\{\text{SLList}\}$  which describes that the heap of the next states of the system under consideration is a singly-linked list. In order to check this formula only the successors of the current state are required. For the state space given in Figure 3.2, starting in state  $s_0$ , the only successor state is  $s_1$ . Thus, checking the heap of  $s_1$  is sufficient to decide whether  $\bigcirc\{\text{SLList}\}$  is satisfied. An on-the-fly approach thus circumvents the generation of the complete state space.

Considering our example introduced in Chapter 1 in Listing 1.2, we construct the proof structure for the method `traverse` based on the state space given in Figure 2.8 for the property  $\Box\{\text{identicNeighbours}\}$  according to ATTESTOR's top level model checking algorithm. Figure 3.4 shows the resulting proof structure, which is indeed successful as no violating path has been encountered. However, including the state space of the method `noop` from Figure 2.9 into the proof structure construction identifies a violating point. Figure 3.5 shows the according proof structure, where the violated assertion is marked as red. Therefore, top level model checking is not sufficient in order to detect property violations within procedure state spaces. We introduce two algorithms that aim at model checking procedure state spaces efficiently.

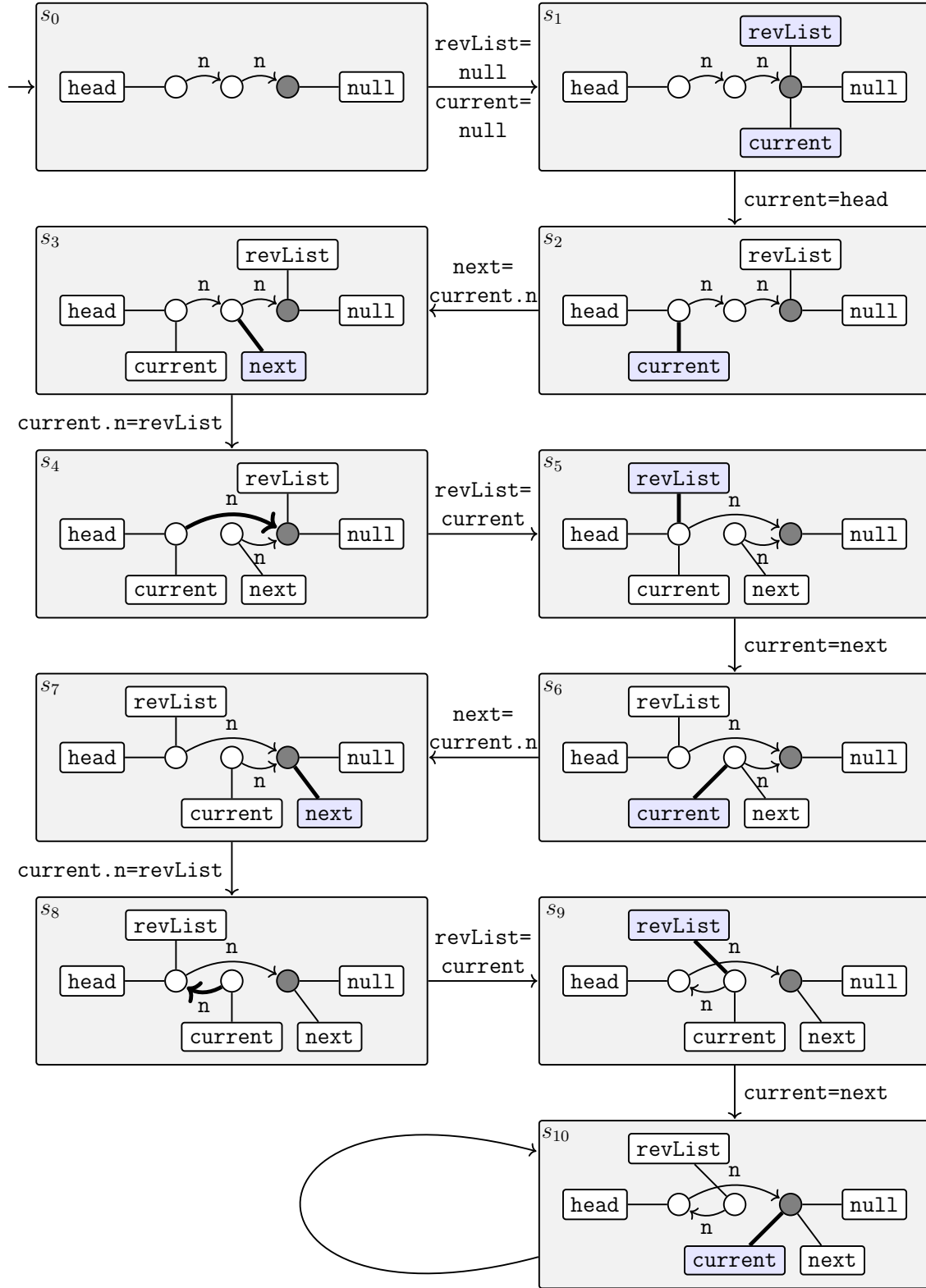


Figure 3.2: State space for reversing a singly-linked list [7].

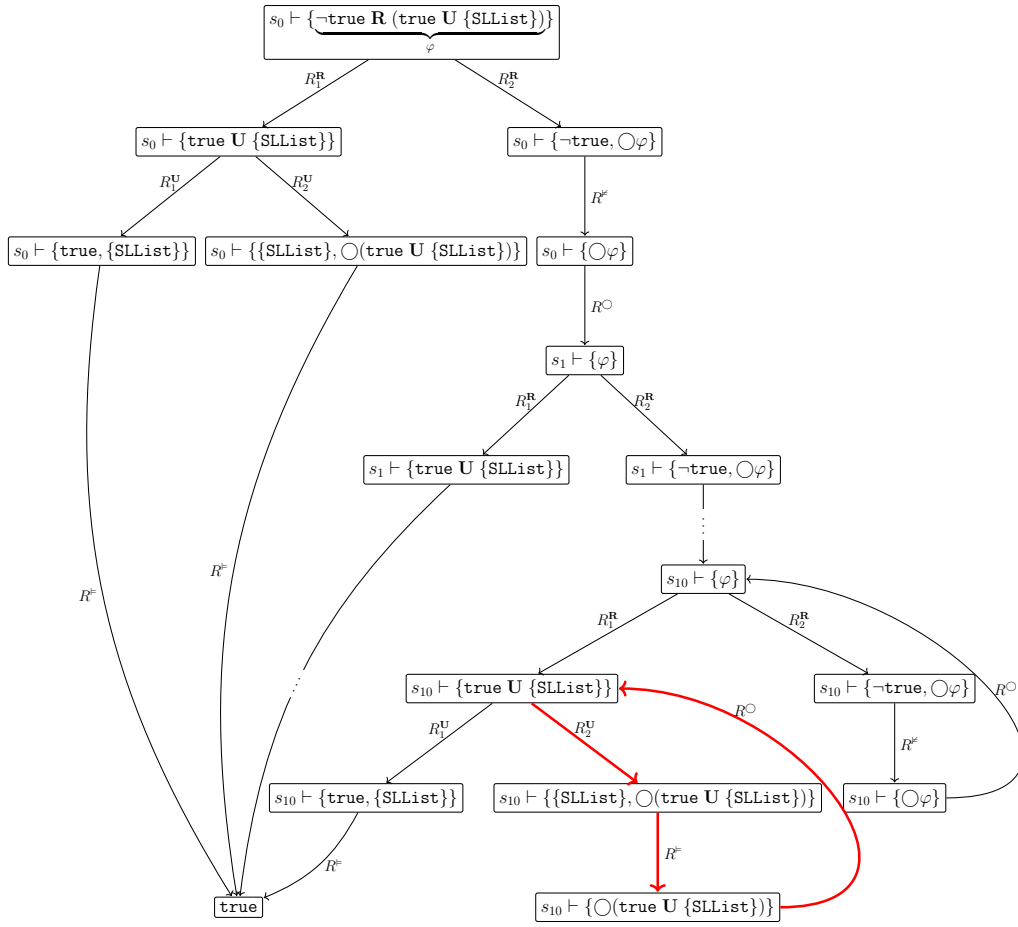


Figure 3.3: Proof structure for model checking method **reverse** for formula  $\varphi = (\neg \text{true } R (\text{true } U \{SList\}))$ . The red loop indicates a violating path.

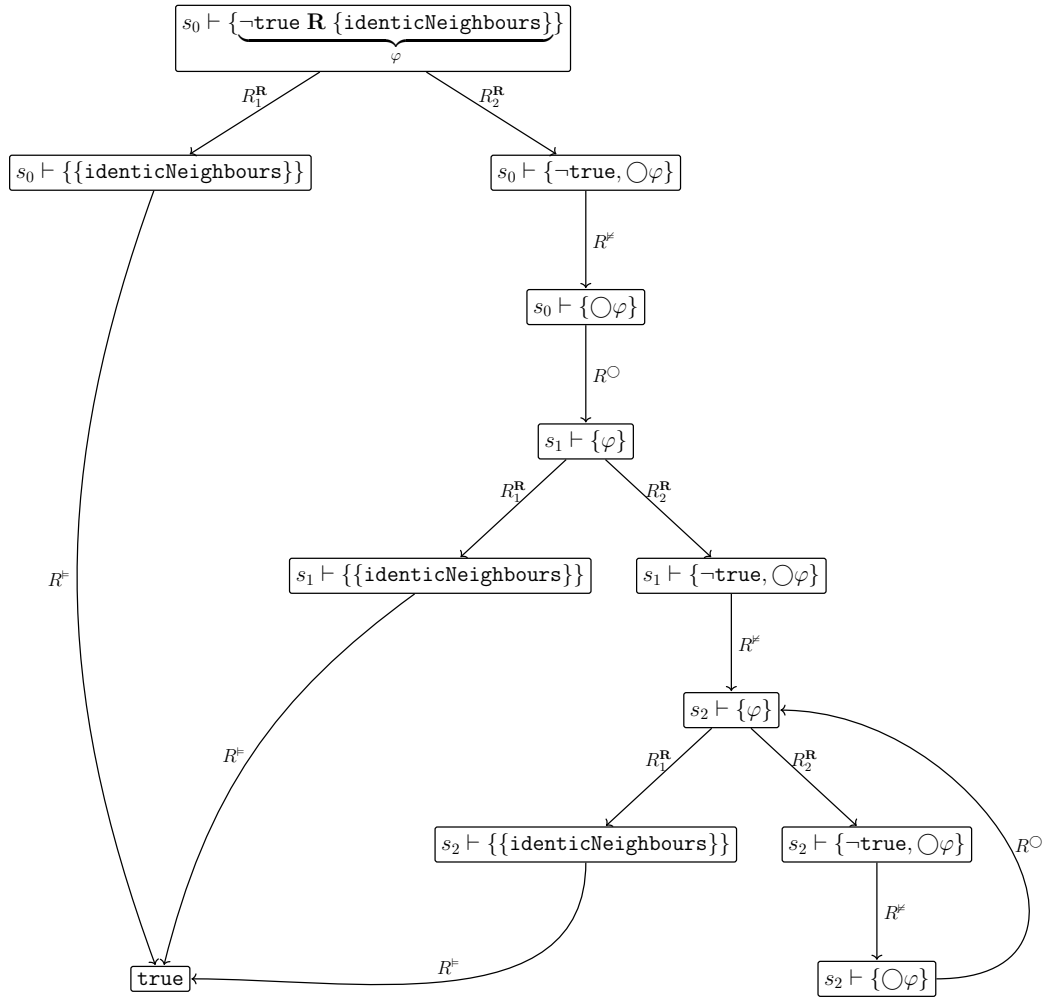


Figure 3.4: Proof structure for model checking method `SLList.traverse` for formula  $\varphi = \Box\{\text{identicNeighbours}\}$ .



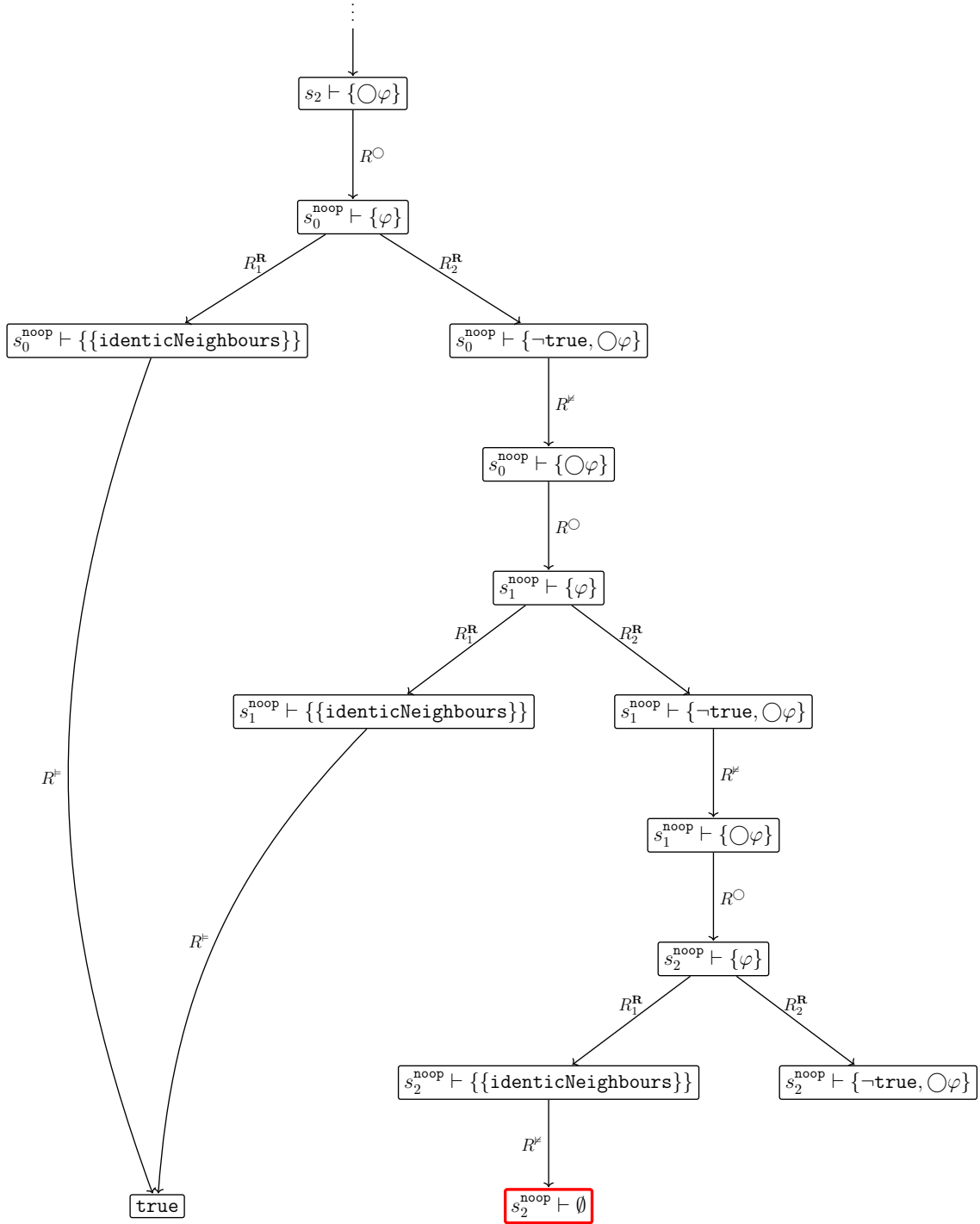


Figure 3.5: Proof structure for model checking method `SLList.noop` for formula  $\varphi = \Box\{\text{identicNeighbours}\}$ .

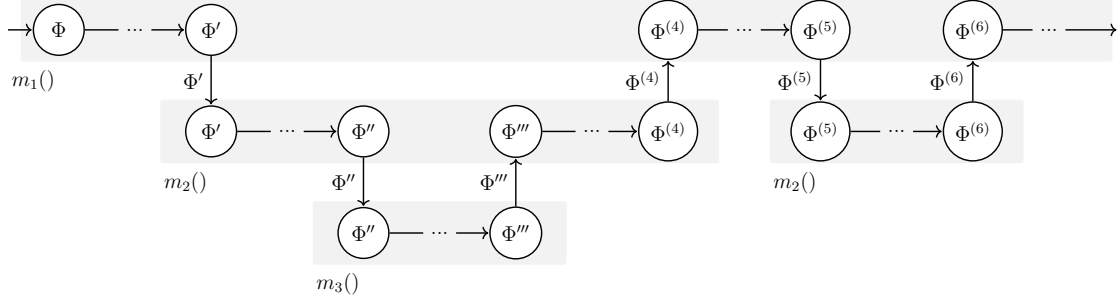


Figure 3.6: Conceptual flow of model checking information in a state space with hierarchical method invocations.

## 3.2 Model Checking Contracts

The tableaux construction by Grumberg et al. [4] is an efficient approach at verifying LTL formulae for software programs. A solution to model checking hierarchical systems is to flatten the underlying state space such that the hierarchy is removed from the model. Then, the state space can be verified by performing tableaux construction on the flattened system. However, this solution yields a number of problems. These include repetitive verification of states in case procedures are called multiple times. Also the size of the state space resulting from flattening a hierarchical model can become unboundedly large due to possibly unbounded recursion depth. Therefore, we propose the concept of *model checking contracts* that allow for model checking hierarchical systems without flattening the underlying state space. Model checking contracts avoid re-computation of previously encountered verification tasks for a state space.

As the hierarchical structure of the system under consideration is kept, it follows that a hierarchy of model checking procedures is constructed where each level represents the verification of a procedure state space. Once model checking is completed for a method execution, model checking results are returned to the above lying level. Figure 3.6 visualizes the flow of model checking information throughout a hierarchy of method calls  $m_i()$ ,  $1 \leq i \leq 3$ . Every executed method  $i$  contributes a state space that is checked. The labels  $\Phi^{(k)}$ ,  $k \in \mathbb{N}$ , denote a set of checked LTL formulae for the respective state.

In order to avoid repetitive verification of procedure state spaces for a same set of LTL formulae, model checking contracts store verification results for procedure state spaces with respect to a verified set of LTL formulae. Hence, results that have been computed beforehand are reused if the same verification task is encountered at a later stage. Model checking results contain information on whether a formula is satisfied as well as the updated set of formulae for which successor states are

checked. They are stored in contracts of the form

$$(HC_{in}, \Phi) \mapsto (\Phi', \delta_{0/1}, \pi),$$

where  $HC_{in}$  denotes the input heap for which the set  $\Phi$  of formulae is validated. The tuple  $(HC_{in}, \Phi)$  is mapped to a tuple  $(\Phi', \delta_{0/1}, \pi)$  of model checking results containing the resulting set  $\Phi'$  of formulae checked for possible successor states, a Boolean value  $\delta_{0/1}$  that indicates whether model checking was successful, and a failure trace  $\pi$  in case a formula is violated. Model checking contracts are stored within *procedure contracts* which unambiguously define the executed procedure. Procedure contracts capture the overall effect of a procedure by defining pairs of pre- and post-conditions that specify the heap prior to and after execution of a procedure, respectively. Details on procedure contracts are described in [9]. Thus, model checking contracts avoid repetitive model checking of a procedure state space.

Another important point in model checking procedure state spaces is the labeling of the contained states with atomic propositions. An atomic proposition  $a \in AP$  is added to a state  $s$  if  $s \models a$ . In ATTESTOR, only the states of the top-level state space are labeled, as procedure state spaces are not considered in the tableaux construction. However, in order to model check procedure state spaces, the corresponding states need to be labeled as well. When a method is invoked at a state  $s$ , the heap of  $s$  is adapted to the scope of the invoked method such that local variables are excluded. Thus, we obtain a *scoped heap*. This might produce faulty results when labeling states within procedure state spaces, as parts of the heap are disguised. For example, consider the property of the heap being a singly-linked list. Assume a heap, where this is in fact true. Now, if we enter a method `foo()` that does nothing, the heap will be scoped to an empty heap. Therefore, the property, that the heap is a singly-linked list, is violated and the state will not be labeled with the corresponding atomic proposition. Consequently, model checking returns that the property is violated despite the fact that the heap is still a singly-linked list outside of the current scope. In order to account for procedure state labeling, we introduce the notion of a *scope hierarchy* which tracks which parts of the heap have been excluded for state space generation. As there might be a hierarchy of method calls, the heap under consideration might have been scoped multiple times. Thus, the collection of scopes is summarized in a scope hierarchy. Now, before computing the labels for a state under consideration, the scope hierarchy is applied to the state in reversed order, such that the original heap is restored. Based on the resulting state, the corresponding atomic propositions are determined.

## Chapter 4

# On-The-Fly Hierarchical Model Checking

The tableaux construction by Grumberg et al. [4] is an on-the-fly approach to model checking transition systems against an LTL formula  $\varphi$ . In this chapter, we describe how we adapt the algorithm to account for model checking of hierarchical structures including procedure state spaces such that not only model checking is executed on-the-fly but also state space generation is performed in an on-the-fly manner. We further present the implementation of the algorithm within the ATTESTOR framework and conclusively discuss our proceedings.

### 4.1 Algorithm

Given an input program and an LTL formula  $\varphi$ , our goal is to verify whether the program under consideration satisfies the temporal property specified by  $\varphi$ . Currently, ATTESTOR solves this problem by first generating a state space for the input program and model checking the resulting state space in a second step. Therefore, the tool implements the tableaux construction by Grumberg et al. (see Figure 2.1). As the state space generation phase only returns the state space of the main method of the input program, procedure state spaces induced by procedure executions are not directly verified during model checking. However, as seen in the example given in Listing 1.2 procedure state spaces might contain erroneous behavior that are not detectable from top level analysis. Hence, we present an on-the-fly hierarchical model checking algorithm that includes procedure state spaces in model checking. As a set of state spaces is included in the process of hierarchical model checking, we aim at providing an approach that does not require to generate all state spaces entirely. This is especially beneficial if verification can be terminated without checking every single state in a state space in case a verification decision can already be made after the analysis of a subset of states. Therefore, we extend the tableaux construction approach by the following functionalities:

- Model checking procedure state spaces, and
- On-the-fly state space generation, therefore interweaving state space generation and model checking.

The on-the-fly hierarchical model checking algorithm is an automated model checking procedure that does not only verify an LTL formula on-the-fly according to the tableaux construction, but also generates the required states of the program under consideration entirely on-the-fly. We illustrate the procedure of the algorithm in Figure 4.1.

The two core parts of the on-the-fly algorithm are the on-the-fly state space generation and the on-the-fly model checking algorithm. The on-the-fly state space generation is based on the ATTESTOR state space generation described in Section 2.1. Figure 4.1 shows how we extend the framework (cf. Figure 2.2) by further processes (marked yellow) that enable the conjunction of both on-the-fly state space generation and on-the-fly model checking. After initializing the state space and the proof structure for an input program, the proof structure is constructed based on the tableaux rules defined in Section 3.1. The construction starts with a single provided state until a  $\bigcirc$ -formula is encountered.  $\bigcirc$ -formulae define properties for successor states of the current state. Hence, the tableaux rule ( $R^\bigcirc$ ) specifies a state switch such that successor states need to be generated. State generation involves execution of the program statement at the current state, i.e. abstract execution. At this point, we extend ATTESTOR's abstract execution by an intermediate step that model checks the just generated state. Therefore, we differentiate between two types of statements: ones that invoke a method execution, and ones that are executed in place. Statements that invoke a method execution trigger the model checking of a procedure state space induced by the called procedure. That is, the on-the-fly hierarchical model checking procedure is executed on the calling state of the method and the current set  $\Phi$  of LTL formulae. The set  $\Phi$  is a set of formulae resulting from successively applying tableaux rules to the initial LTL formula  $\varphi$ . Model checking results for a procedure state space and a set of LTL formulae are stored in model checking contracts.

After abstract execution, it is checked whether model checking of the resulting successor states is successful. In case the verification returns `true`, new assertions induced by the successor states and a set  $\Phi'$  of resulting LTL formulae are added to the proof structure and the procedure is continued. Otherwise, a violating path has been found and the proof structure is declared to be unsuccessful. Hence, the proof structures of the above lying state spaces can be aborted as a violating path has been found such that it can be concluded that the set of formulae  $\Phi$  is not satisfied for the complete program state space. Thus, an early termination of the model checking procedure is possible that does not require the computation of an

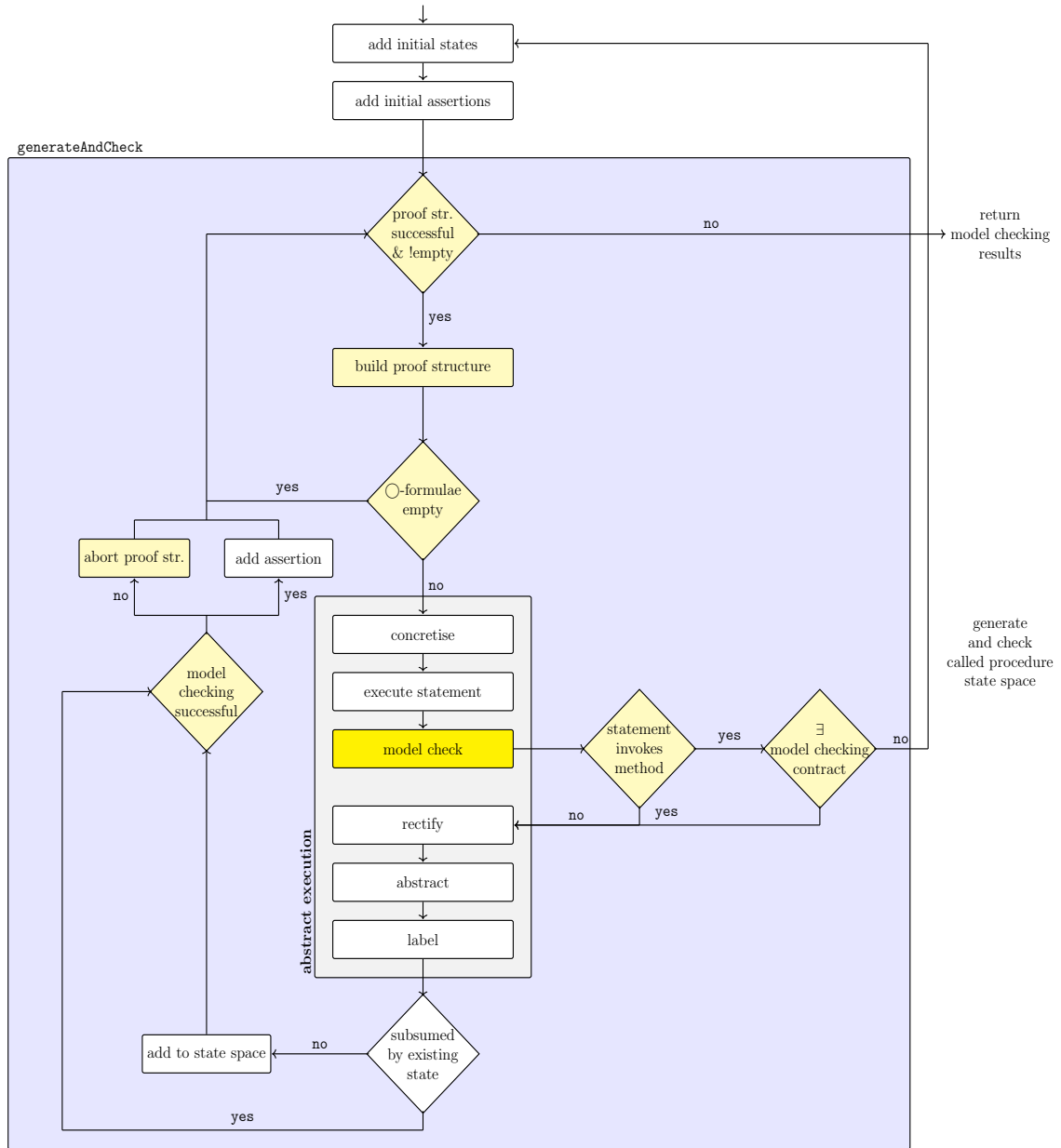


Figure 4.1: On-the-fly hierarchical model checking by a hierarchical tableaux construction.

entire state space.

The correctness of the on-the-fly tableaux construction for hierarchical state spaces follows from the correctness of the tableaux construction [4] and the correctness of the state space generation algorithm defined in [2].

## 4.2 Implementation

We implement the on-the-fly hierarchical model checking algorithm within the ATTESTOR framework written in JAVA. The code is available at [https://github.com/SallyChau/master\\_thesis/tree/master/attestor](https://github.com/SallyChau/master_thesis/tree/master/attestor). As both state space generation and model checking are performed, the algorithm encompasses ATTESTOR's state space generation and model checking phases. Therefore, the on-the-fly hierarchical model checking algorithm constitutes a separate phase in ATTESTOR referred to as the *on-the-fly model checking phase*. It is hence an alternative to the execution of ATTESTOR's verification phases.

The implementation of the on-the-fly model checking phase requires two main adaptations in the ATTESTOR framework. First, we need to adapt the tableaux construction algorithm such that it does not work on a complete state space, but rather successively demands for new states as soon as they are required for building the proof structure. Second, instead of generating an entire state space during a separated phase, we require a possibility to query for a list of successor states for a given state  $s$ . Figure 4.2 sketches the architecture of our implementation.

We first describe the implementation of the on-the-fly model checking procedure. In ATTESTOR, the `ProofStructure` class implements the tableaux construction in order to verify an LTL formula for a given input state space. By calling the method `ProofStructure.build` the proof structure is constructed. It requires a completely generated state space to operate on. As we aim at an approach that only computes states when required for expanding a proof structure, we modify the current implementation by introducing the class `OnTheFlyProofStructure`. Here, the tableaux construction is realized in a way such that it is capable of constructing the proof structure for an on-the-fly constructed state space. Instead of requiring a completely generated state space, the on-the-fly version solely requires initial assertions. These are expanded according to the tableaux rules until an assertion which only contains  $\bigcirc$ -formula is encountered. Since the underlying state space is not present, the proof structure construction cannot be continued. Thus, we require the generation of the successors of the state associated with the current assertion. At this point, the control is handed to the `OnTheFlyStateSpaceGenerator`, where successor states are generated for a given state. This step is indicated by a directed arrow labeled " $\bigcirc$ -formulae" in Figure 4.2 connecting the classes `OnTheFlyProofStructure` and

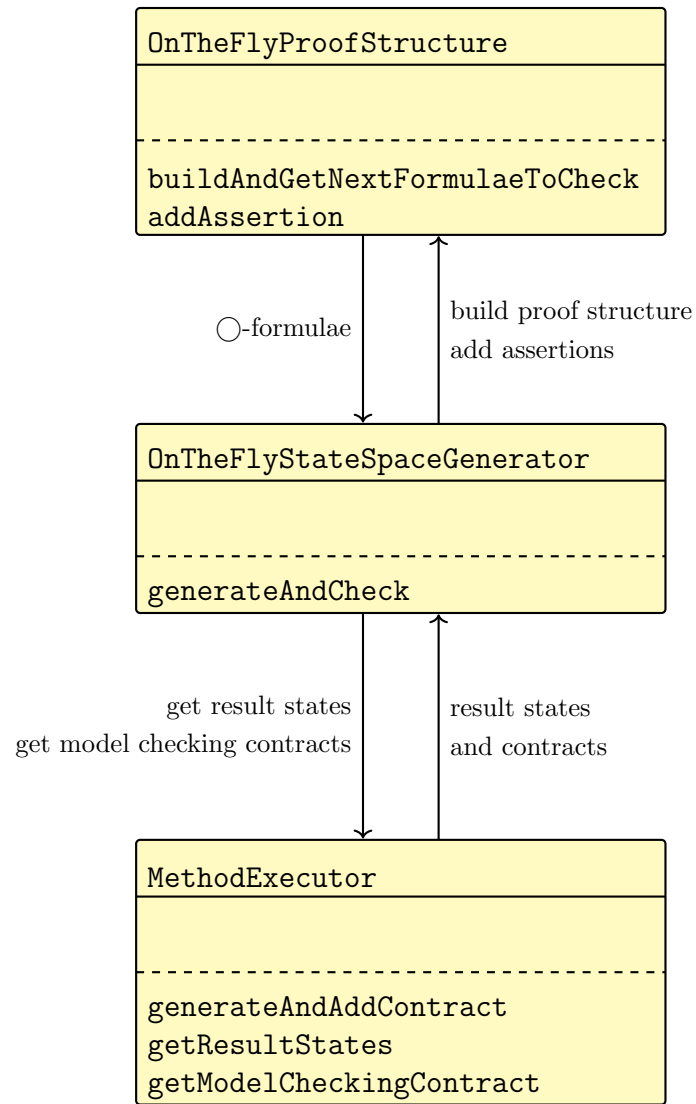


Figure 4.2: Conceptual UML-diagram for on-the-fly hierarchical model-checking.



`OnTheFlyStateSpaceGenerator`.

The class `OnTheFlyStateSpaceGenerator` realizes the computation of successor states for a given state  $s$ . It extends ATTESTOR's `StateSpaceGenerator` class, where an entirely generated state space is returned for an input program. In order to compute successor states for a state  $s$ , the program statement at state  $s$  is executed on the current heap. If the statement invokes a procedure execution, the verification of the called procedure is triggered. ATTESTOR's `MethodExecutor` class handles procedure calls by initiating (on-the-fly) state space generation for the called procedure. As procedure invocation induces a procedure state space, model checking is also performed for these such that the `MethodExecutor` class does not only return a list of successor states of the calling state  $s$ , but also information on model checking results. These are communicated to the proof structure by adding new assertions and reentering the model checking loop. This process is depicted by the directed arrow pointing from the `OnTheFlyStateSpaceGenerator` class to the `OnTheFlyProofStructure` class in Figure 4.1. The communication between the classes depicted in the diagram in Figure 4.1 continues until one of the following cases occur: a verification decision is made, i.e., the proof structure returns that the property under consideration is satisfied or violated, all states have been verified, or computational resources, i.e., memory, have been exceeded.

We execute the implementation of the on-the-fly hierarchical model checking for the example from Listing 1.2, where the method `noop` introduces errors in the execution of the main method `traverse`. The example shows that top level model checking is not sufficient to detect errors in procedure state spaces. In contrast to this, executing the on-the-fly hierarchical model checking algorithm for the erroneous `traverse` method effectively returns that the property, that a list is never mutated, is violated. The returned failure trace of state IDs

$$\underbrace{[[1, 14, 15, 44, 56, 67, 68]]}_{\text{traverse}}, \underbrace{[0, 1, 3, 5]}_{\text{noop}}$$

indicates that the source of the failure lies within a procedure call, as the failure trace contains two distinct traces, i.e., the first list of states describes the trace in the top level state space of the main method `traverse`, the second list describes the trace in the state space of the procedure `noop`.

## 4.3 Discussion

When model checking procedural programs with possibly recursive procedure calls, procedure state spaces need to be taken into account. ATTESTOR's current model checking approach only verifies the top-level state space and thus disregards pos-

sible erroneous behavior within method executions. The on-the-fly approach on hierarchical model checking presented in this chapter solves this gap by verifying procedure state spaces during their generation. Thus, the algorithm successfully interweaves state space generation and model checking. Violations can also be tracked on procedure state space level and hence offer more precise debugging instances. In order to avoid repetitive model checking of model checking instances, the algorithm applies model checking contracts for procedure state spaces in order to reuse model checking results that have been computed beforehand. Furthermore, the on-the-fly state space construction allows for early termination of the model checking procedure such that time and memory can be saved as not all states need to be determined.

However, in case of model checking a procedure state space for multiple distinct LTL formulae, the procedure state space needs to be computed afresh since they are not stored. This is due to the fact that the on-the-fly approach does not guarantee to return complete state spaces that can be reused. This might cause an overhead of state space generations.

Thus, the on-the-fly approach is especially suitable for cases in which erroneous behavior is expected within method executions in order to quickly find a counterexample, whereas positive validation of a property might cause an overhead of computations.

Chapter 6 presents benchmarks on the on-the-fly model checking algorithm described in this chapter.

## Chapter 5

# RSM-based Hierarchical Model Checking

Recursive state machines model the control flow of sequential imperative programs containing (recursive) procedure calls. Within the concept of RSMs, two kinds of states are differentiated: states that constitute a statement that is executed in place, and states that invoke method executions. This differentiation is helpful in modeling the hierarchical structure of procedural programs. Alur et al. [1] present an automata-based approach for model checking recursive state machines. The algorithm constructs a product automaton of the RSM under consideration and Büchi automaton representing the negation of an LTL formula to be verified. The product automaton is then checked for language emptiness. If the accepted language of the product automaton is empty, it can be concluded that the intersection of the languages accepted by the RSM and the Büchi automaton is empty. Therefore, the property specified by the LTL formula is satisfied. As the automata-based approach requires the construction of three automata (RSM, Büchi, and product automaton) for verifying an LTL formula, we propose an algorithm that only operates on the RSM itself. Therefore, we employ the tableaux construction by Grumberg et al. [4] for model checking RSMs. In an RSM, component state machines store the pre-generated procedure state spaces of procedures. In order to account for verification of procedure state spaces, a proof structure is constructed according to the tableaux construction for a state space of a component when the represented procedure is called. Results are stored as model checking contracts within a component state machine such that they can be reused if required. In the sequel of this chapter, we introduce the RSM-based algorithm in detail and describe its implementation within the ATTESTOR framework. Finally, we evaluate the approach in terms of time and space efficiency.

## 5.1 Algorithm

Given a transition system  $T$  and an LTL formula  $\varphi$ , the goal of hierarchical model checking is to verify whether  $T \models \varphi$ . The RSM-based hierarchical model checking algorithm solves the verification problem in a two-step fashion: First, a recursive state machine is constructed for an input program. Second, model checking is executed on the generated RSM.

We illustrate the process of the RSM-based hierarchical model checking algorithm in Figure 5.1. In contrast to the on-the-fly approach described in the previous chapter, this approach separates state space generation and model checking into two phases. The first phase constructs an RSM for an input program. We assume that the (procedure) state spaces are generated in a previous step, e.g., in ATTESTOR's state space generation phase, and are therefore given. The construction of the RSM starts with generating a component state machine for every method of the input program. Procedure state spaces are mapped to the corresponding component based on the called method. Calling dependencies between states of a procedure state space and called procedures are stored in a set of boxes that indicate which method is invoked by which state within a component. In this context, one component might contain a set of state spaces as a procedure might have been called by different calling states.

The second phase executes LTL model checking for the generated RSM and a set  $\Phi$  of LTL formulae. Model checking an RSM boils down to model checking its components. Thus, starting with the component  $A_1$  corresponding to the main method of the program, a tableaux is constructed for the state space of  $A_1$ . If during the construction, a state  $s$  is encountered that invokes a method  $j$ , the according box is entered and model checking is continued for component  $A_j$  for the set  $\Phi'$  of formulae. By this means, a hierarchy of procedure model checking processes is created, as presented in Figure 3.6. The model checking results for a component  $A_j$  are returned to the calling state, and included in the proof structure of the calling component. Similar to the on-the-fly hierarchical model checking approach, repetitive model checking of the same set of formulae for a given procedure state space is avoided by storing model checking results as model checking contracts. Thus, if a previously solved verification task appears, the proof structure does not need to be repeatedly computed, but stored results are applied. If a violating state has been encountered, the proof structure construction is aborted for all components, otherwise the procedure is continued until all relevant proof structures have been constructed such that the proof structure of the main state space, thus of the component  $A_1$ , comes to a conclusion.

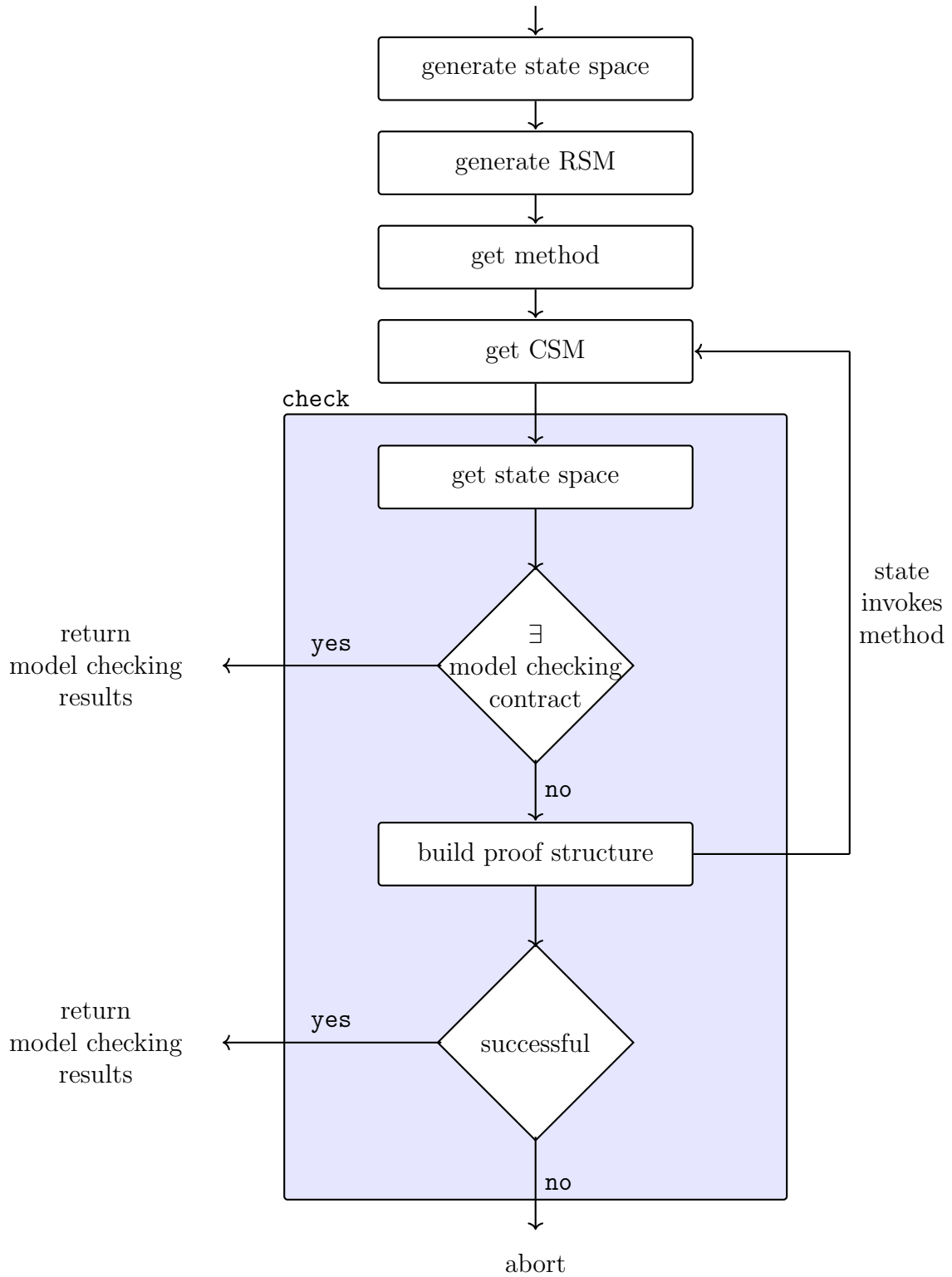


Figure 5.1: RSM-based hierarchical model checking using the tableaux construction.

## 5.2 Implementation

We implemented the RSM-based hierarchical model checking algorithm within the ATTESTOR framework. The implementation is available at [https://github.com/SallyChau/master\\_thesis/tree/master/attestor](https://github.com/SallyChau/master_thesis/tree/master/attestor). The approach is an alternative to the tool’s currently implemented model checking phase that uses the tableaux construction to verify the top-level state space of an input program for an LTL formula  $\varphi$ . In order to realize the RSM-based algorithm within ATTESTOR, we require the following adaptations:

- Retrieve procedure state spaces from state space generation phase,
- Implement classes for recursive state machines and component state machines, and
- Enrich RSM and components with proof structures to account for model checking procedure state spaces.

Figure 5.2 sketches the implementation of the required adaptations.

We retrieve procedure state spaces and the corresponding calling states from ATTESTOR’s state space generation phase. Given a set of procedure state spaces, the class `RecursiveStateMachine` builds a corresponding RSM by creating an object of the class `ComponentStateMachine` for each method of an input program. Since distinct calls to a method originating from distinct states might result in distinct state spaces, a `ComponentStateMachine` internally maps the calling state to the resulting state space. The class `RecursiveStateMachine` manages the calling information of a state  $s$  by creating boxes in the state’s `ComponentStateMachine` that contains the information which `ComponentStateMachine` is called by  $s$ . Moreover, a `ComponentStateMachine` stores the model checking results for executed tableaux constructions in order to avoid repetitive computation of proof structures, similar to the implementation in Chapter 4. However, since the context of the method is unambiguously determined by the `ComponentStateMachine`, we do not require the context of procedure contracts as it is the case for on-the-fly model checking.

In order to account for model checking procedure state spaces, we require the class `ComponentStateMachine` to execute tableaux construction for the according state space. Therefore, we implement the class `HierarchicalProofStructure` which adapts ATTESTOR’s proof structure to work on state spaces stored in a `ComponentStateMachine`. If during proof structure construction a state is encountered that invokes a method execution, the corresponding `ComponentStateMachine` representing the called method starts model checking for the according procedure state space. After termination of the sub-model checking procedure, we return to

the previous proof structure applying the recent model checking results. If the result is a final one, i.e., the proof structure is unsuccessful or the proof structure is ultimately successful (there are no more formulae to check), the result is reported to the dependent proof structures, such that model checking is terminated. Finally, the model checking result is returned including a failure trace in case of property violation.

In **ATTESTOR**, the RSM-based hierarchical model checking algorithm is executed by setting the option

`--hierarchical-model-checking`

in the settings of the verification task. The RSM-based approach then replaces **ATTESTOR**'s top level model checking. Executing the RSM-based hierarchical algorithm on the example instance given in Listing 1.2, returns that the property, that the list is never mutated, is violated by the method **traverse**. In this context, the algorithm returns the failure trace of state IDs

$$[[\underbrace{7, 103, 104, 105, 106, 107, 108}_{\text{traverse}}, \underbrace{0, 1, 2, 3}_{\text{noop}}],$$

that indicates that the source of the violating path lies within the procedure **noop**. This result is similar to the one of the on-the-fly hierarchical model checking algorithm except for the state IDs as both algorithms employ differing state space generation algorithms.

## 5.3 Discussion

In this chapter, we presented an algorithm for model checking procedure state spaces represented as a recursive state machine. The hierarchical structure of the state space is captured by the RSM as every method is represented by a component. The component stores corresponding procedure state spaces, boxes that represent method invocations, and model checking results. Model checking of a procedure state space is controlled by the corresponding component, such that verification results are communicated to the calling state.

RSM-based hierarchical model checking has the advantage that procedure state spaces are stored within the component state machine of a method. Thus, procedure state spaces only need to be computed once during the state space generation phase and are reused in future model checking tasks. Similarly, model checking of a formula  $\varphi$  can be skipped if the results are known due to previous model checking instances as results are stored in contracts.

However, the advantage of reusing previously computed results induces the require-

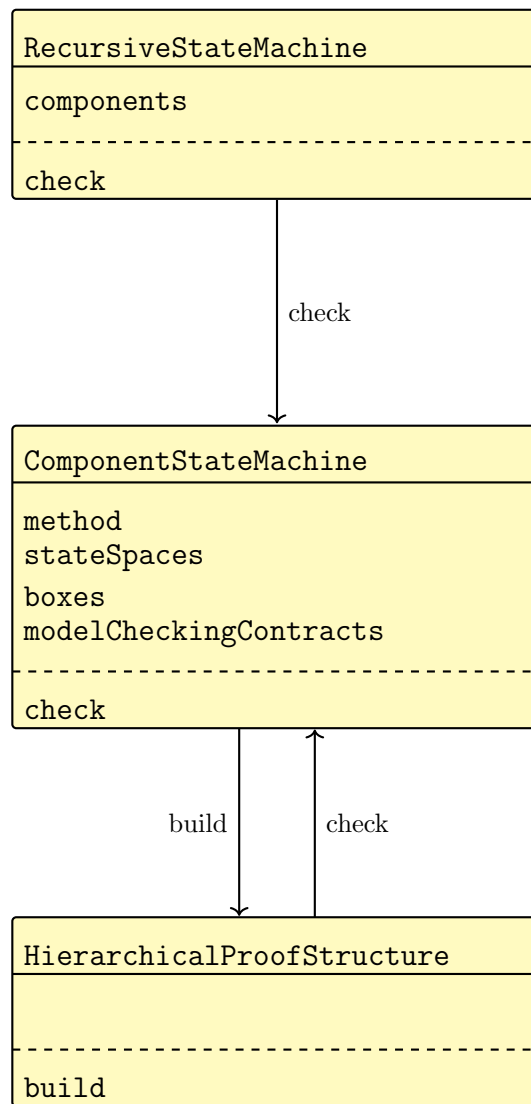


Figure 5.2: UML for hierarchical model checking with RSMs.



---

ment to store the state spaces and the model checking results. Hence, the memory usage of the algorithm increases with the number of distinct procedures called during program execution. Furthermore, generation of the state spaces prior to model checking requires that the complete state space is generated. Therefore, early termination of state space generation in case only a subset of states is sufficient for tableaux construction is not enabled.

In order to evaluate the algorithm in practice, Chapter 6 presents benchmarks on the RSM-based model checking algorithm and compares the approach to the performance of the previously introduced on-the-fly hierarchical model checking and ATTESTOR's top level model checking procedure.

# Chapter 6

## Benchmarks

We presented two algorithms for model checking hierarchical state spaces that result from analyzing input programs with procedure calls. In Chapter 4, we presented the on-the-fly hierarchical model checking algorithm that uses the tableaux construction by Grumberg et al. [4] to build a proof structure for checking a property while generating the underlying state space on-the-fly. Chapter 5 introduced the RSM-based hierarchical model checking algorithm that models the hierarchical state space as a recursive state machine such that procedure state spaces are depicted. Based on the collection of state spaces, the tableaux construction is used to verify the property under consideration. In this chapter, we compare the performances of the algorithms with the model checking algorithm implemented in ATTESTOR that uses the tableaux construction to check the top level state space. In this context, we evaluate the execution time and the number of generated states of the individual algorithms on 79 model checking instances.

### 6.1 Experimental Setup

The benchmarks are performed on an Intel Core i7-8650U CPU @ 1.90 GHz with the Java virtual machine (OpenJDK version 1.80\_112). The benchmark instances are executed by the ATTESTOR tool with the help of a benchmark helper class, that is based on the ATTESTOR BENCHMARK HELPER. For each model checking algorithm, i.e.

- ATTESTOR's top level model checking,
- on-the-fly hierarchical model checking, and
- RSM-based hierarchical model checking,

we determine the following statistics:

- the number of generated states of the main state space and the total number of generated states,

- the execution time of the individual algorithm consisting of the time needed for state space generation and model checking in seconds.

A settings file specifies the options for the execution of a benchmark instance. Generally, the options include but are not limited to

- `--description` a description of the performed model checking task,
- `--classpath` the path to the `.java` file that contains the input program,
- `--class` the class to be analyzed,
- `--method` the method to be analyzed,
- `--predefined-grammar` the grammar to be employed during the analysis specifying which data structure is analyzed,
- `--initial` a file that specifies the heap of the initial state. If none is specified the heap is assumed to be initially empty.
- `--model-checking` the LTL formula to be checked,
- `--hierarchical-model-checking` an option that indicates that the RSM-based hierarchical model checking algorithm is executed instead of the top level model checking algorithm. The on-the-fly hierarchical model checking algorithm involves an own phase in ATTESTOR such that it is executed next to the tool's standard state space generation and model checking.

## 6.2 Instances

We compare the hierarchical model checking algorithms and ATTESTOR's top level model checking based on 79 benchmark instances. Each instance is a model checking task that verifies a property given as an LTL formula for an input program that operates on a specified data structure.

The input program contains methods working on data structures whose grammar is predefined in the ATTESTOR tool, i.e., singly-linked lists (`SLList`), doubly-linked lists (`DLList`), and trees (`Tree`). For each data structure, we implement several operations in JAVA code such as `SLList.traverse` or `DLList.reverse` that run through all elements of a singly-linked list or reverse the order of the contained elements in a doubly-linked list, respectively.

Each operation is verified for a number of properties specified as LTL formulae. We analyze the following properties during the model checking procedures:

- *Memory Safety* (M) is checked during state space generation. In this context, we analyze the LTL formula  $\Box \text{true}$  such that the on-the-fly hierarchical model checking generates the entire states space. Without a specified formula, the on-the-fly approach would not be executed as there is no specified formula to be verified.
- The *completeness property* (C) verifies whether all elements of the input are visited by a specific variable. For example, the formula

$$\Box \Diamond \{\text{terminated}\} \rightarrow \Diamond \{\text{visited}(\text{current})\}$$

describes that all elements of the input are eventually visited by the variable `current` upon termination of the program.

- The *correctness property* (Co) summarizes properties that verify that an input program delivers correct results. Therefore, we consider formulae such as

$$\Box \Diamond \{\text{terminated}\} \rightarrow \Diamond \{(\text{return} == \text{null}) \ \& \ (\text{terminated})\}$$

stating that the return value is `null` upon termination of the program. The converse property can be verified by using this formula if the model checking algorithm outputs that the property is violated.

- The *neighborhood preservance property* (N) proves whether the input data structure corresponds to the data structure upon termination of the program. An LTL formula to specify this property is

$$\Box \Diamond \{\text{terminated}\} \rightarrow \Diamond \{\text{terminated} \ \& \ \text{identicNeighbours}\}.$$

- The *reachability property* (R) verifies whether a variable is reachable from another one via specified pointers upon termination. An example for an LTL formula specifying a reachability property is given by

$$\Diamond \{\bigcirc \{\text{terminated}\} \rightarrow \text{isReachable}(\text{head}, \text{tail}, [\text{next}])\}$$

which states that the variable `tail` is reachable from the variable `head` via `next` pointers.

- The *shape property* (S) checks whether the heap is of a certain shape. For example, the formula

$$\Box \Diamond \{\text{terminated}\} \rightarrow \Box \Diamond \{\text{SLList}\}$$

defines that the heap is of the shape of a singly-linked list.

For the completeness and neighborhood preservance properties it is required to track object identities during program execution. This is done by marking objects such that they are identified across different states in the state space [2, 7].

In order to analyze the effectiveness and efficiency of the algorithms in terms of hierarchical model checking, the instances include input programs with and without procedure calls as well as recursive methods. Furthermore, we introduced erroneous code snippets such that property violation (on procedure level) is examined.

An overview of all implemented methods and the verified properties is depicted in Table A.1. The table also shows the experimental results for executing each introduced model checking algorithm in this thesis for every instance.

## 6.3 Results

The executed benchmark instances give us a basis to compare the three model checking algorithms in terms of their *effectiveness* and *efficiency* in model checking hierarchical state spaces. We base the effectiveness of an algorithm on the correctness of the model checking result and its efficiency on the required number of generated states and time in order to perform a verification task.

### 6.3.1 Effectiveness

We executed 79 benchmark instances both on the hierarchical model checking algorithms and ATTESTOR's top level model checking algorithm. The model checking results for each algorithm is displayed in Table A.1. For most instances, the three algorithms yield the same result. However, within the benchmark instances six instances include code that introduce manipulations to the heap within a procedure call such that the property under consideration is violated within a procedure state space. Executing these examples shows that the top level model checking algorithm is not able to spot the violations within procedure state spaces, thus returns that the property is satisfied. However, the hierarchical model checking algorithms find failure traces within procedure states and output that the property under consideration is violated. Consider the example in Listing 1.2 from Chapter 3, where the method `SLList.traverseModifyList` implements a list traversal equipped with a `noop`-method that swaps out the current list with the `null` value back and forth, so that the `noop`-method has no lasting effect on the original list. Verifying the property that the initial list is never mutated, the top level model checking algorithm outputs that the property is verified. However, both hierarchical model checking algorithms indicate that the property is violated within a procedure state space.

Consequently, in contrast to the top level model checking algorithm the hierarchical approaches verify hierarchical state spaces for each level such that erroneous behavior within procedure calls are detected.

### 6.3.2 Efficiency

The efficiency of the hierarchical model checking algorithms is evaluated based on the determined number of generated states required for model checking and the execution time. We compare these figures with the ones of the top level model checking algorithm that includes a preceding state space generation phase. We compare the algorithms' performances by computing the differences in percent. Grouped by different categories of benchmark instances, the differences in the number of generated states and execution times are depicted in Figures 6.1, 6.2, and 6.3, respectively.

Figure 6.1 displays the difference in the number of generated states between the on-the-fly hierarchical model checking algorithm and the top level model checking algorithm. The overall average difference of 12% states that the on-the-fly hierarchical approach generally generates 12% more states than the top level approach. This is due to the fact that (procedure) state spaces are not stored such that states are possibly computed multiple times. However, there are also cases in which the on-the-fly hierarchical approach requires less states in order to conduct model checking. These include instances where properties are found to be violated as displayed in Figures 6.1c and 6.1f where up to 50% less states are generated compared to the top level model checking algorithm. This is due to the fact that violating states are found at a stage where not the complete state space has been generated yet, so that the state space generation is terminated. The RSM-based model checking algorithm employs ATTESTOR's state space generation such that there are no differences in the number of generated states compared to ATTESTOR's verification procedure.

Next to the number of generated states, the execution time of the model checking algorithms is an important factor to consider in evaluating their efficiency. Figures 6.2 and 6.3 show the difference in percent in the execution times compared to ATTESTOR's model checking algorithm including the state space generation for the on-the-fly hierarchical algorithm and the RSM-based hierarchical algorithm, respectively. Both algorithms require around 108% or 77% more time than the top level approach. This is due to the fact that the hierarchical algorithms generally verify more states than the top level algorithm as they also consider procedure states in the model checking procedure. It is striking that especially the on-the-fly hierarchical model checking algorithm requires considerably more time than the top level approach. This coincides with the greater amount of states generated during the on-the-fly state space generation. In this context, the only bars indicating that the on-the-fly approach is faster than ATTESTOR's implementation are the bars for in-

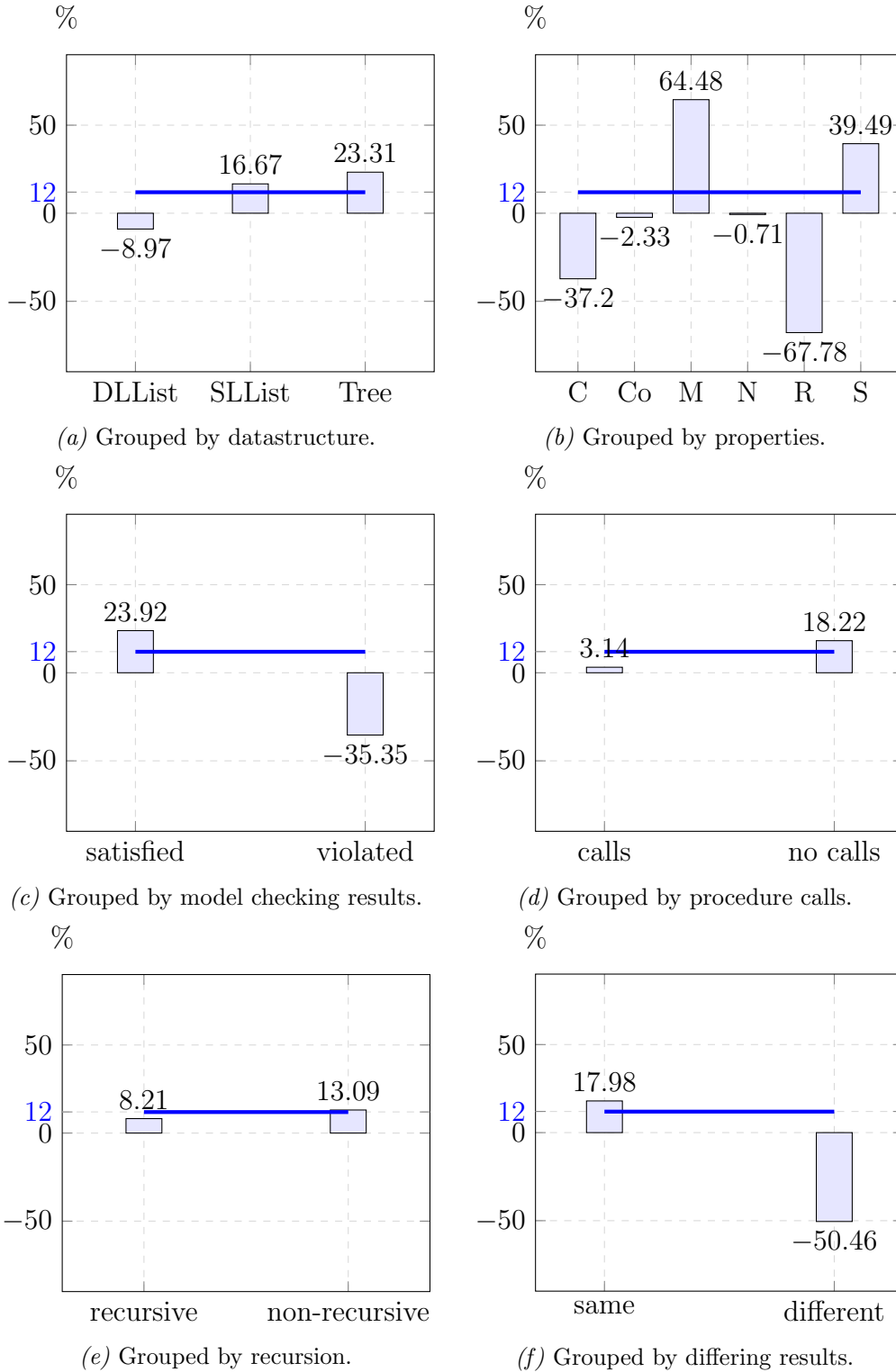


Figure 6.1: Difference in the number of generated states in % between the on-the-fly hierarchical model checking algorithm and the top level model checking grouped by different categories.

stances that verify reachability properties in Figure 6.2b and the instances for which the top level and the hierarchical algorithms deliver differing model checking results, which are also the categories for which the on-the-fly approach generates less states than ATTESTOR's state space generation.

Comparing the on-the-fly and the RSM-based hierarchical model checking algorithms with each other, it becomes visible that the RSM-based algorithm is faster in average. As the number of generated states coincide with the figures of the top level algorithm, the RSM-based approach also generates less states in average than the on-the-fly procedure. Although the RSM-based approach is generally slower than the top level model checking algorithm, it comes with the advantage that procedure state spaces are also verified during model checking.



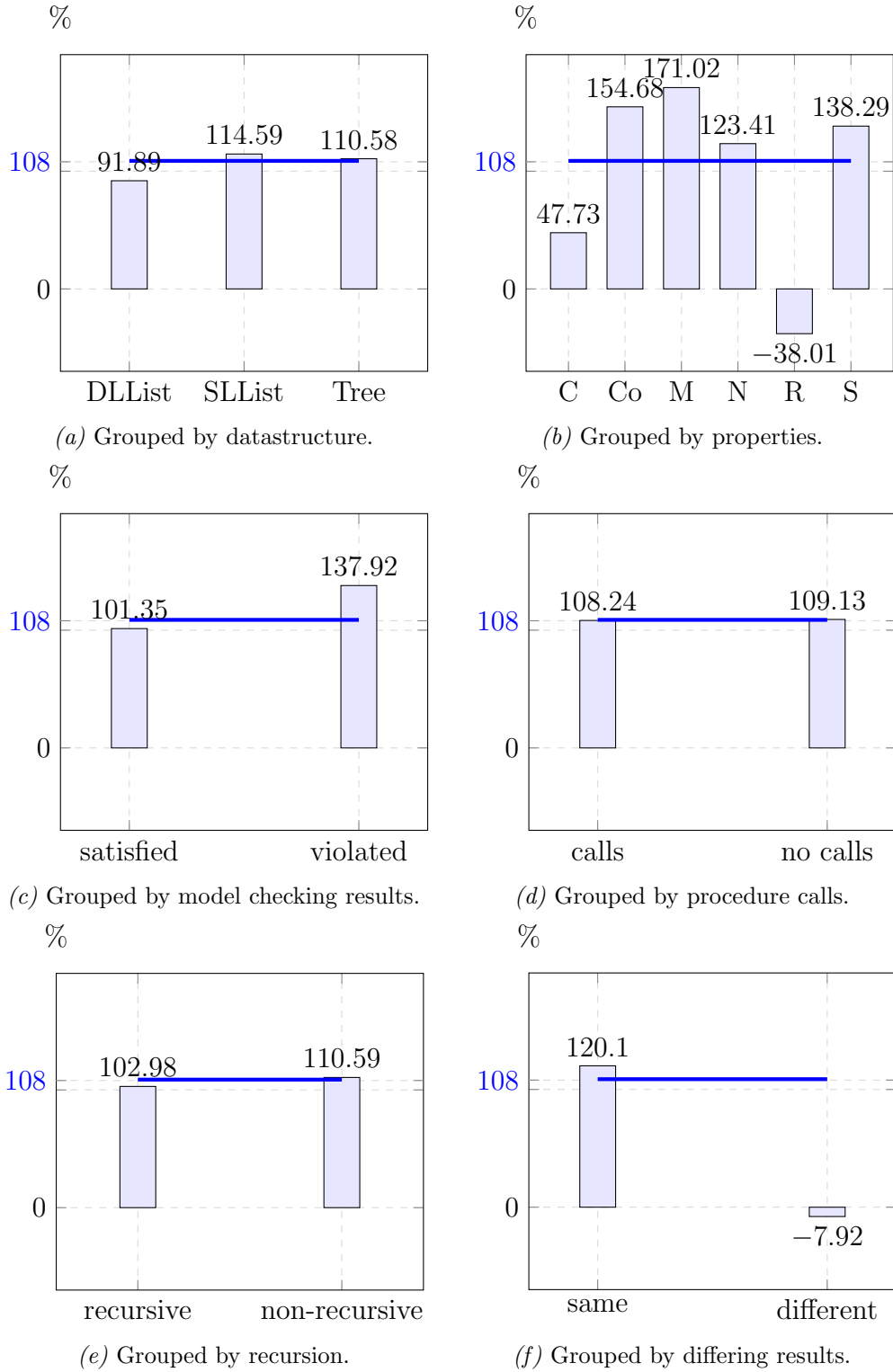


Figure 6.2: Difference in the execution time in % between the on-the-fly hierarchical model checking algorithm and the top level model checking grouped by different categories.

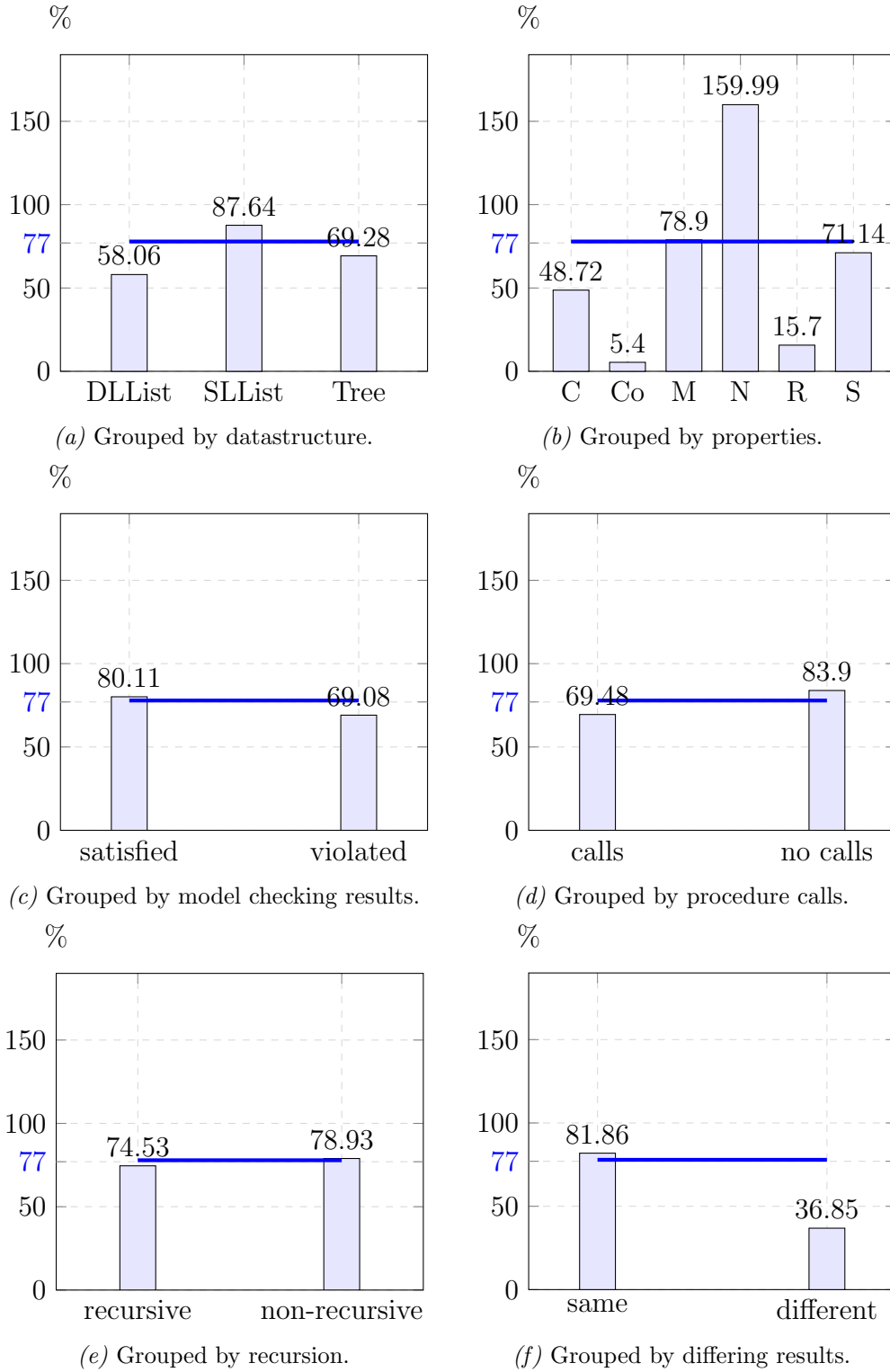


Figure 6.3: Difference in the execution time in % between the RSM-based hierarchical model checking algorithm and the top level model checking grouped by different categories.

## Chapter 7

# Conclusion and Future Work

We introduced two hierarchical model checking algorithms, i.e., the on-the-fly and the RSM-based hierarchical model checking algorithms. We implemented the algorithms within the ATTESTOR tool in JAVA code and executed the algorithms on a set of 79 model checking tasks. Based on the performed benchmarks, we compared the two hierarchical model checking algorithms with ATTESTOR's model checking procedure that verifies a property under consideration for the main state space of an input program.

Although all three model checking algorithms presented in this thesis employ the tableaux construction Grumberg et al. [4], the algorithms demonstrate differing performance results in terms of effectiveness and efficiency. The differences are traced back to the underlying state space of the model checking task. While the state space is constructed on-the-fly (and terminates as soon as a statement can be made about the validity of the input formula) in the on-the-fly hierarchical model checking approach, ATTESTOR's top level analysis works on an entirely generated state space returned by the preceding state space generation phase. As the tableaux construction does not always require the complete state space, the on-the-fly hierarchical model checking approach generates the states on demand during the tableaux construction. Furthermore, states in procedure state spaces are directly model checked, so that not only the top level state space is considered for model checking. Thus, verifying properties that are expected to not require a completely generated state space, i.e., properties that are expected to be violated, or properties referring to a limited set of states, is performed efficiently by the on-the-fly hierarchical model checking algorithm. The on-the-fly nature of the algorithm, however, also entails the disadvantage that states might be generated multiple times as procedure state spaces are not stored. Model checking multiple LTL formulae for the same state space thus requires recomputation of the state space for every verification task. This problem is approached by the RSM-based model checking algorithm that stores all state spaces generated during ATTESTOR's state space generation phase such that procedure state spaces are available during model checking. The RSM-based approach

then structures the collection of state spaces within a recursive state machine and performs the tableaux construction for model checking based on the RSM. This approach has the advantage that states are only computed once and procedure state spaces can be checked. This is especially beneficial in cases where a hierarchical state space is checked for multiple different LTL formulae. However, storing all procedure state spaces infers a high demand of storage.

Consequently, the introduced hierarchical model checking algorithms show that top level model checking is not sufficient to trace failures that are introduced within procedure calls. Thus, including procedure state spaces in model checking reveals possibly erroneous behavior that is not visible from the context of the main program. Based on the handling of state space generation, different goals in model checking are approached; a combination of an on-the-fly state space generation with an on-the-fly tableaux construction delivers fast results for verifying possibly violated LTL formulae while RSM-based model checking is suitable to verify a set of properties on different levels of a method stack.

Further extensions in the area of hierarchical model checking include the generation of a counter example such that the input program can be tested and improved. In this context, the implemented hierarchical model checking algorithms introduced in this thesis deliver a failure trace that indicates a path within the state space that violates the property under consideration. It is left to implement the procedure of translating the failure trace into a counter example.

Furthermore, as both the on-the-fly and the RSM-based hierarchical model checking algorithm include advantages and disadvantages in model checking procedure state spaces, a possible solution is to create a hybrid method that generates procedure state spaces on-the-fly and storing them as an RSM. This approach allows us to benefit from the on-the-fly approach as well as from the RSM-based procedure. On the one hand, in case a model checking result is determined without requiring the entirely generated state space, we benefit from the early termination property of the on-the-fly approach. On the other hand, the RSM allows us to access previously (partly) generated state spaces. Hence, the disadvantages of both algorithms are mitigated by combining their advantages.

In a next step, our benchmark results are to be refined by including instances for additional data structures and properties such that the algorithms are tested on a greater and more varied number of instances.

# Bibliography

- [1] ALUR, RAJEEV, KOUSHA ETESSAMI and MIHALIS YANNAKAKIS: *Analysis of Recursive State Machines*. In *CAV*, volume 2102 of *Lecture Notes in Computer Science*, pages 207–220. Springer, 2001.
- [2] ARNDT, HANNAH, CHRISTINA JANSEN, JOOST-PIETER KATOEN, CHRISTOPH MATHEJA and THOMAS NOLL: *Let this Graph Be Your Witness! - An Attestor for Verifying Java Pointer Programs*. In *CAV (2)*, volume 10982 of *Lecture Notes in Computer Science*, pages 3–11. Springer, 2018.
- [3] BAIER, CHRISTEL and JOOST-PIETER KATOEN: *Principles of model checking*. MIT Press, 2008.
- [4] BHAT, GIRISH, RANCE CLEAVELAND and ORNA GRUMBERG: *Efficient On-the-Fly Model Checking for CTL\**. In *LICS*, pages 388–397. IEEE Computer Society, 1995.
- [5] ESPARZA, JAVIER, DAVID HANSEL, PETER ROSSMANITH and STEFAN SCHWOON: *Efficient Algorithms for Model Checking Pushdown Systems*. In *CAV*, volume 1855 of *Lecture Notes in Computer Science*, pages 232–247. Springer, 2000.
- [6] ESPARZA, JAVIER and STEFAN SCHWOON: *A BDD-Based Model Checker for Recursive Programs*. In *CAV*, volume 2102 of *Lecture Notes in Computer Science*, pages 324–336. Springer, 2001.
- [7] HEINEN, JONATHAN: *Verifying Java Programs - A Graph Grammar Approach*. PhD thesis, RWTH Aachen University, 2015.
- [8] HEINEN, JONATHAN, CHRISTINA JANSEN, JOOST-PIETER KATOEN and THOMAS NOLL: *Juggrnaut: using graph grammars for abstracting unbounded heap structures*. *Formal Methods in System Design*, 47(2):159–203, 2015.
- [9] JANSEN, CHRISTINA and THOMAS NOLL: *Generating Abstract Graph-Based Procedure Summaries for Pointer Programs*. In *ICGT*, volume 8571 of *Lecture Notes in Computer Science*, pages 49–64. Springer, 2014.

# Appendix A

## Benchmark results

87

Method	P	Top Level MC						On-The-Fly HMC				RSM-based HMC					
		States			Time in <i>s</i>			States			Time in <i>s</i>	States			Time in <i>s</i>		
		t/f	Main	$\Sigma$	SSG	MC	$\Sigma$	t/f	Main	$\Sigma$	$\Sigma$	t/f	Main	$\Sigma$	SSG	MC	$\Sigma$
DLList.build	M	t	30	62	0.0578	0.0032	0.0609	t	30	62	0.1976	t	30	62	0.0553	0.0203	0.0756
DLList.build	S	f	30	62	0.0503	0.0015	0.0519	f	12	36	0.0621	f	30	62	0.0510	0.0072	0.0582
DLList.findLastCall	C	f	33	629	0.5845	0.0024	0.5869	t	17	236	0.5374	t	33	629	0.4778	0.0425	0.5202
DLList.reverse	C	t	572	572	0.0859	0.0067	0.0926	t	444	444	0.0897	t	572	572	0.0587	0.0580	0.1167
DLList.reverse	Co	f	82	82	0.0336	0.0062	0.0397	f	82	82	0.1528	f	82	82	0.0342	0.0109	0.0451
DLList.reverse	Co	f	86	86	0.0392	0.0047	0.0439	f	82	82	0.0813	f	86	86	0.0331	0.0096	0.0427
DLList.reverse	M	t	70	70	0.0041	0.0007	0.0048	t	133	133	0.0139	t	70	70	0.0062	0.0023	0.0085
DLList.reverse	N	f	1332	1332	0.5705	0.0310	0.6015	f	837	837	4.1222	f	1332	1332	0.7106	1.1568	1.8674
DLList.reverse	N	f	1332	1332	0.6196	0.0134	0.6330	f	817	817	0.8044	f	1332	1332	0.6230	0.3187	0.9417
DLList.reverse	R	t	100	100	0.0488	0.0003	0.0491	t	4	4	0.0067	t	100	100	0.0484	0.0004	0.0488
DLList.reverse	R	t	100	100	0.0363	0.0002	0.0365	t	4	4	0.0023	t	100	100	0.0335	0.0003	0.0338
DLList.reverse	S	t	70	70	0.0086	0.0009	0.0095	t	133	133	0.0222	t	70	70	0.0090	0.0025	0.0115
DLList.traverse	C	t	308	308	0.0474	0.0017	0.0491	t	228	228	0.0388	t	308	308	0.0353	0.0243	0.0596
DLList.traverse	M	t	38	38	0.0027	0.0002	0.0029	t	69	69	0.0065	t	38	38	0.0035	0.0008	0.0043
DLList.traverse	N	t	820	820	0.1068	0.0034	0.1103	t	1458	1458	0.3267	t	820	820	0.1308	0.2683	0.3991
DLList.traverse	N	t	820	820	0.1321	0.0175	0.1496	t	1458	1458	0.3805	t	820	820	0.1581	0.4518	0.6099
DLList.traverse	R	t	56	56	0.0174	0.0001	0.0176	t	4	4	0.0019	t	56	56	0.0140	0.0004	0.0144
DLList.traverse	S	f	38	38	0.0056	0.0001	0.0057	f	15	15	0.0017	f	38	38	0.0061	0.0003	0.0064
SLList.buildAndConcat	M	t	47	91	0.0205	0.0018	0.0222	t	87	150	0.0884	t	47	91	0.0221	0.0220	0.0441
SLList.buildAndConcat	S	t	47	91	0.0322	0.0046	0.0368	t	19	104	0.1314	t	47	91	0.0315	0.0617	0.0932
SLList.buildRecursive	M	t	13	92	0.0079	0.0004	0.0083	t	16	169	0.0343	t	13	92	0.0085	0.0059	0.0144
SLList.buildRecursive	S	f	13	92	0.0165	0.0007	0.0171	f	26	112	0.0598	f	13	92	0.0149	0.0123	0.0272
SLList.build	M	t	14	32	0.0029	0.0002	0.0032	t	24	48	0.0098	t	14	32	0.0024	0.0019	0.0042
SLList.build	S	t	14	32	0.0033	0.0003	0.0037	t	27	79	0.0171	t	14	32	0.0054	0.0047	0.0101
SLList.findMiddleFaultyCall	R	t	4	57	0.0132	0.0002	0.0134	f	2	29	0.0177	f	4	57	0.0197	0.0016	0.0213
SLList.findMiddleFaulty	M	t	53	53	0.0040	0.0019	0.0059	t	53	53	0.0116	t	53	53	0.0067	0.0020	0.0087
SLList.findMiddleFaulty	R	f	53	53	0.0139	0.0008	0.0147	f	27	27	0.0154	f	53	53	0.0138	0.0010	0.0147

SList.findMiddle	C	t	384	384	0.0269	0.0064	0.0333	t	362	362	0.0450	t	384	384	0.0225	0.0277	0.0502
SList.findMiddle	C	f	421	421	0.0489	0.0059	0.0548	f	211	211	0.1296	f	421	421	0.0419	0.1010	0.1429
SList.findMiddle	M	t	84	84	0.0027	0.0006	0.0033	t	152	152	0.0079	t	84	84	0.0055	0.0017	0.0072
SList.findMiddle	N	t	453	453	0.0356	0.0040	0.0396	t	843	843	0.1401	t	453	453	0.0336	0.0479	0.0816
SList.findMiddle	R	t	84	84	0.0090	0.0008	0.0098	t	152	152	0.0300	t	84	84	0.0081	0.0024	0.0105
SList.findMiddle	S	t	90	90	0.0055	0.0012	0.0068	t	162	162	0.0166	t	90	90	0.0064	0.0017	0.0081
SList.find	C	t	80	80	0.0033	0.0007	0.0041	t	52	52	0.0043	t	80	80	0.0050	0.0019	0.0068
SList.find	C	f	100	100	0.0082	0.0018	0.0100	f	52	52	0.0237	f	100	100	0.0069	0.0075	0.0144
SList.find	M	t	29	29	0.0012	0.0003	0.0016	t	52	52	0.0036	t	29	29	0.0016	0.0012	0.0029
SList.find	N	t	165	165	0.0104	0.0018	0.0122	t	293	293	0.0287	t	165	165	0.0117	0.0222	0.0340
SList.find	S	t	31	31	0.0028	0.0005	0.0033	t	54	54	0.0076	t	31	31	0.0031	0.0013	0.0045
SList.reverseRecursive	M	t	18	128	0.0029	0.0002	0.0031	t	32	313	0.0094	t	18	128	0.0049	0.0017	0.0066
SList.reverseRecursive	N	f	132	2097	0.0416	0.0016	0.0433	f	117	567	0.0666	f	132	2097	0.0400	0.0222	0.0622
SList.reverseRecursive	R	t	15	117	0.0136	0.0001	0.0137	t	5	5	0.0013	t	15	117	0.0121	0.0006	0.0127
SList.reverseRecursive	S	t	18	128	0.0069	0.0002	0.0071	f	11	57	0.0099	f	18	128	0.0113	0.0018	0.0131
SList.reverse	C	t	174	174	0.0085	0.0009	0.0094	t	82	82	0.0064	t	174	174	0.0095	0.0027	0.0122
SList.reverse	M	t	37	37	0.0009	0.0003	0.0013	t	68	68	0.0028	t	37	37	0.0016	0.0011	0.0026
SList.reverse	N	f	372	372	0.0326	0.0104	0.0430	f	263	263	0.2313	f	372	372	0.0497	0.1183	0.1681
SList.reverse	N	f	372	372	0.0375	0.0026	0.0402	f	290	290	0.0507	f	372	372	0.0420	0.0502	0.0922
SList.reverse	R	t	61	61	0.0103	0.0001	0.0104	t	5	5	0.0012	t	61	61	0.0116	0.0004	0.0120
SList.reverse	S	t	40	40	0.0034	0.0013	0.0047	t	40	40	0.0303	t	40	40	0.0067	0.0096	0.0163
SList.reverse	S	t	37	37	0.0028	0.0004	0.0033	t	68	68	0.0071	t	37	37	0.0042	0.0014	0.0056
SList.traverseModifyHeap	S	t	4	16	0.0026	0.0001	0.0027	f	2	5	0.0015	f	4	16	0.0027	0.0005	0.0032
SList.traverseModifyList	M	t	29	41	0.0023	0.0003	0.0025	t	49	65	0.0057	t	29	41	0.0027	0.0017	0.0043
SList.traverseModifyList	N	t	184	220	0.0189	0.0016	0.0205	f	81	120	0.0171	f	184	220	0.0158	0.0218	0.0376
SList.traverseRecursiveModifyHeap	S	t	15	68	0.0035	0.0001	0.0036	f	2	5	0.0012	f	15	68	0.0062	0.0005	0.0067
SList.traverseRecursive	C	f	32	489	0.0091	0.0004	0.0094	f	45	214	0.0145	f	32	489	0.0129	0.0059	0.0187
SList.traverseRecursive	N	t	90	1407	0.0152	0.0001	0.0153	t	11	11	0.0004	t	90	1407	0.0265	0.0009	0.0275
SList.traverseRecursive	R	t	10	101	0.0068	0.0001	0.0069	t	3	3	0.0007	t	10	101	0.0116	0.0003	0.0119
SList.traverseRecursive	S	t	10	45	0.0020	0.0002	0.0022	t	15	152	0.0102	t	10	45	0.0038	0.0022	0.0060
SList.traverseRecursive	S	t	10	45	0.0028	0.0001	0.0029	t	15	152	0.0111	t	10	45	0.0050	0.0014	0.0064
SList.traverse	C	t	47	47	0.0011	0.0002	0.0013	t	30	30	0.0017	t	47	47	0.0014	0.0005	0.0019
SList.traverse	M	t	18	18	0.0004	0.0001	0.0005	t	32	32	0.0011	t	18	18	0.0008	0.0005	0.0013
SList.traverse	N	t	95	95	0.0033	0.0001	0.0034	t	11	11	0.0004	t	95	95	0.0063	0.0005	0.0068
SList.traverse	N	t	95	95	0.0031	0.0005	0.0036	t	168	168	0.0074	t	95	95	0.0042	0.0027	0.0069
SList.traverse	R	t	34	34	0.0033	0.0001	0.0034	t	3	3	0.0005	t	34	34	0.0044	0.0002	0.0046
SList.traverse	S	t	18	18	0.0010	0.0003	0.0013	t	32	32	0.0026	t	18	18	0.0011	0.0006	0.0018
SList.traverse	S	t	18	18	0.0010	0.0001	0.0011	t	32	32	0.0023	t	18	18	0.0017	0.0004	0.0021
SList.zipDummy	M	t	9	437	0.0200	0.0001	0.0201	t	9	591	0.0394	t	9	437	0.0229	0.0267	0.0496
Tree.build	M	t	472	480	0.1485	0.0138	0.1623	t	472	480	0.5840	t	472	480	0.1799	0.1086	0.2885
Tree.build	S	t	472	480	0.1979	0.0163	0.2142	t	10	26	0.0248	t	472	480	0.2029	0.1506	0.3535
Tree.getLeftmostChildModifyList	S	t	58	86	0.0181	0.0009	0.0190	f	35	65	0.0149	f	58	86	0.0195	0.0020	0.0215
Tree.getLeftmostChild	M	t	22	22	0.0015	0.0003	0.0019	t	46	46	0.0038	t	22	22	0.0015	0.0007	0.0022
Tree.getLeftmostChild	N	t	805	805	0.1383	0.0556	0.1939	t	1813	1813	0.5297	t	805	805	0.0962	0.8691	0.9653
Tree.getLeftmostChild	S	t	43	43	0.0037	0.0004	0.0040	t	91	91	0.0135	t	43	43	0.0042	0.0009	0.0051
Tree.getLeftmostChild	S	t	43	43	0.0059	0.0015	0.0074	t	91	91	0.0169	t	43	43	0.0071	0.0027	0.0097
Tree.traverseRecursiveModifyHeap	S	t	38	1037	0.0624	0.0005	0.0629	f	21	930	0.1070	f	38	1037	0.0561	0.0120	0.0681
Tree.traverseRecursive	C	f	220	27083	0.8981	0.0051	0.9032	f	52	23151	2.5851	f	220	27083	0.8303	0.0482	0.8785
Tree.traverseRecursive	M	t	34	972	0.0264	0.0006	0.0270	t	40	1525	0.0714	t	34	972	0.0272	0.0130	0.0402
Tree.traverseRecursive	N	t	1033	301704	11.0335	0.0296	11.0631	t	754	3614	0.4402	t	1033	301704	10.1003	8.6893	18.7896
Tree.traverseRecursive	S	t	34	972	0.0369	0.0005	0.0374	t	40	1525	0.1071	t	34	972	0.0540	0.0052	0.0593
Tree.traverseRecursive	S	t	34	972	0.0247	0.0005	0.0252	t	40	722	0.0594	t	34	972	0.0342	0.0136	0.0478

Table A.1: Benchmark results comparing model checking (MC) and hierarchical model checking (HMC) algorithms on the generated number of states during state space generation (SSG) and the execution times in seconds.