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
Matriculation Number 370584
June 20, 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

1 Introduction **12**

1.1 ATTESTOR 12

1.2 Related Work 16

2 Preliminaries **17**

2.1 System Model 17

2.1.1 Transition Systems 17

2.1.2 Recursive State Machines 21

2.1.3 Heap Representation 26

2.2 Linear Temporal Logic 31

3 Hierarchical Model Checking **37**

3.1 Tableaux Construction 38

3.2 Automata-Based Model Checking 40

4 On-The-Fly Hierarchical Model Checking **45**

4.1 Algorithm 45

4.2 Implementation 47

4.3 Evaluation 49

5 Hierarchical Model Checking with Recursive State Machines 51

5.1 Algorithm 51

5.2 Implementation 51

5.3 Evaluation 51

6 Benchmarks 52

6.1 Experimental Setup 52

6.2 Instances 52

6.3 Result 52

7 Conclusion 53

7.1 Discussion 53

7.2 Outlook 53

Chapter 1

Introduction

- introduce current status of model checking in attestor: after state space generation, only checking top level state space
- goal: model check procedure state spaces
- present possible algorithms: automata-based and on-the-fly tableau
- always keep goal of hierarchical model checking in mind

1.1 Attestor

Attestor is a verification tool that generates an abstract state space of the input program and employs LTL model checking to verify specified properties. The state space generation is based on a finite representation of the program heap during execution by a (hyper)graph. Finiteness is achieved by employing abstraction based on graph grammars that specify the data structured maintained by the program and how subgraphs can be summarized in a so called hyperedges. Occurring methods in the input program are summarized by procedure contracts that specify the heap configurations prior to and after their execution. Thus, procedure state spaces do not need to be computed repeatedly for the same heap configuration. In a next step, property verification is performed by model checking the resulting state space against the LTL specifications. Attestor then either outputs that the program fulfills or (possibly) violates the property. In the latter case, a counterexample is provided.

The structure of Attestor can be mainly divided into five parts, namely input, back-end, front-end, API and output, which is depicted in Figure 1.1. In order to specify a verification task, Attestor offers four possible input parameters of which the input program is mandatory. The user can further specify LTL specifications to be verified during model checking, a graph grammar for the data structures present in the input program, and further options such as initial heap configurations or properties to modify abstraction or garbage collection. The core of the Attestor tool

is the back-end which describes the verification process. The ATTESTOR front-end communicates via the ATTESTOR API with the back-end to visualize the output such as the generated state space.

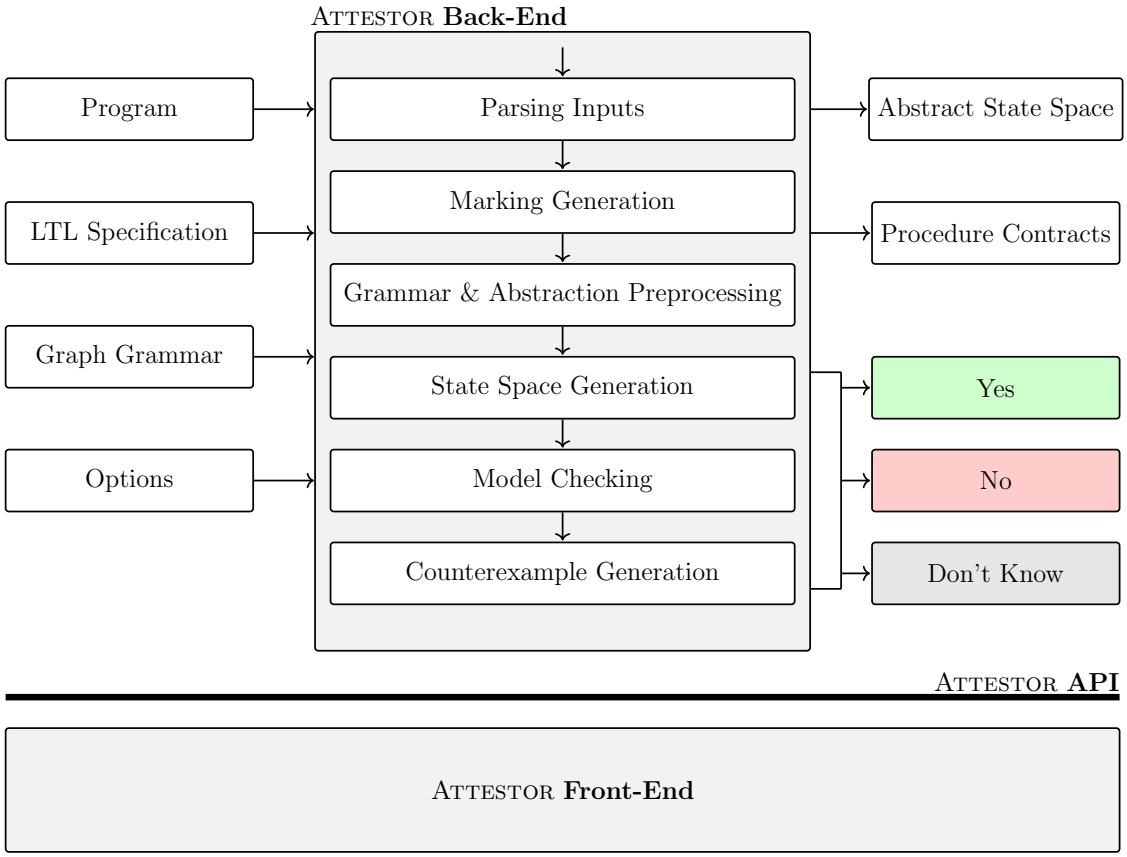


Figure 1.1: ATTESTOR architecture. [2]

The ATTESTOR back-end constitutes the core process of the tool which is divided into six phases. The first three phases comprise the preprocessing of the verification task, followed by the state space generation phase and the model checking phase.

Phase 1: Parsing Inputs In the first phase, the supplied input options passed to ATTESTOR are read including the input program, graph grammars, initial heap configurations as well as procedure contracts that contain predefined pre- and post-conditions that describe the behavior of a procedure execution.

Phase 2: Marking Generation After parsing the input, markings are added to the initial heap if required by the specified LTL properties [5]. 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 transformers required for state space generation itself, e.g., garbage collection.

Phase 4: State Space Generation After the stages of preprocessing, the state space is generated by executing the statements of the input program on the initial heap such that new states are added. The procedure is illustrated in Figure 1.2. The abstract execution loop is executed until either there are no more states left to process or a fixpoint has been reached. The loop starts with picking a state, which has not been processed yet, at random. The abstract semantics of the next statement are applied to the state. Therefore, the heap configuration potentially needs to be locally concretised so that the statement is executed on a concrete heap configuration. Concretisation is achieved by applying the 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, an abstraction of the heap is followed where grammar rules are applied in a backward manner. Therefore, embeddings of the heap configuration in the right hand side of grammar rules have to be found. Finally, the resulting state is labeled with atomic propositions that are satisfied by the heap configuration. 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 state. If not, the state is added to the state space and the algorithm continues with the next state. Else, Attestor checks whether a fixpoint has been reached and terminates the procedure in this case. Otherwise, state space generation continues. The generated state space represents the transformation of the initial heap configuration during program execution for the main method of the input program.

Phase 5: Model Checking State space generation is followed by the model checking phase if LTL formulae have been specified. Attestor currently implements the tableaux method described in Section 3.1 which checks the main state space for LTL formulae. In case a formulae 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 procedural programs contain (recursive) methods and method calls with proper state spaces that are not (directly) checked in the current implementation. Rather, procedure calls are woven into the main state space by

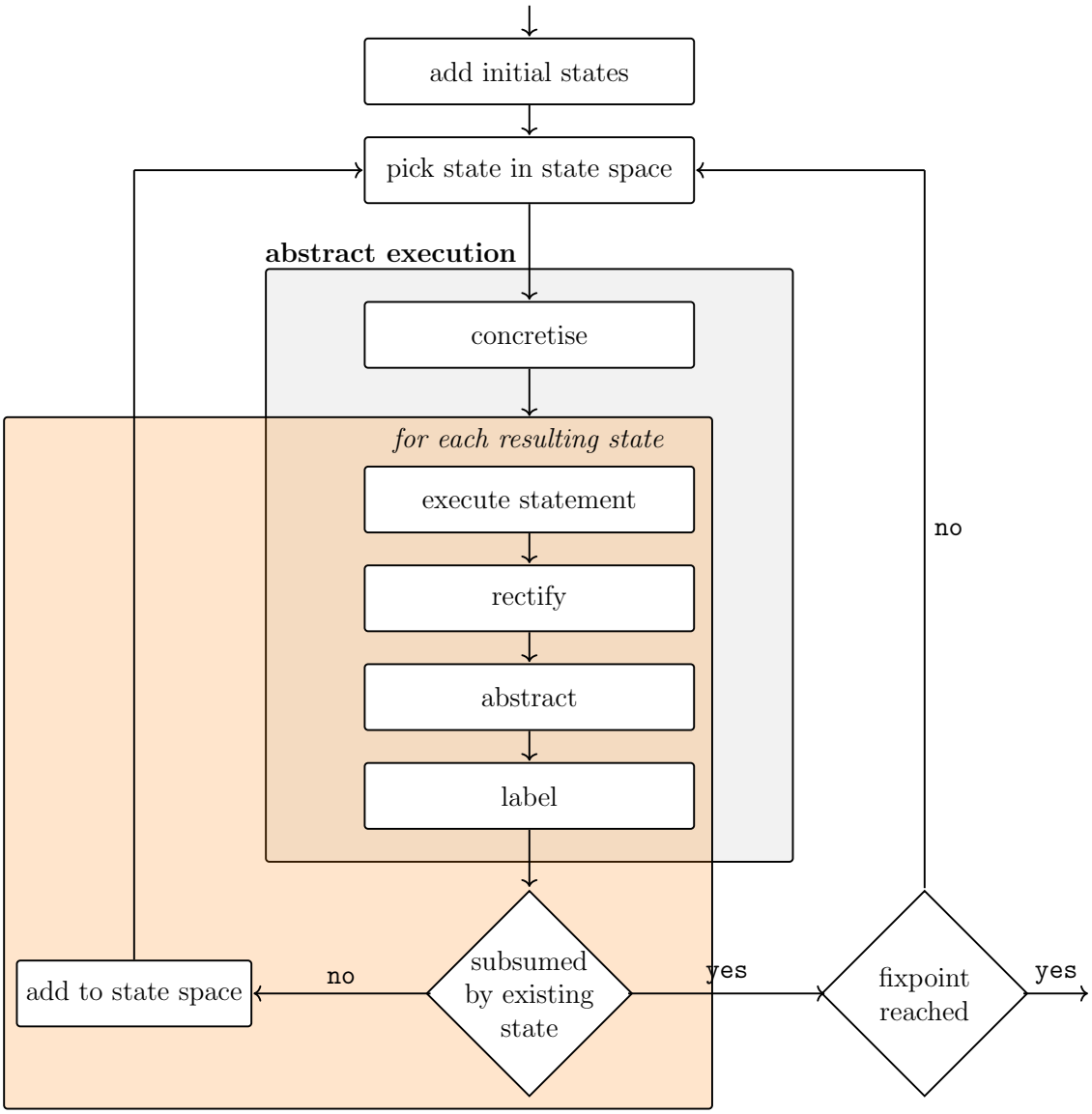


Figure 1.2: Phase 4: State space generation in ATTESTOR [2].

considering their influence on the heap configuration only after the execution. However, properties should also be checked for the 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 5 and 4 that also verifies procedure state spaces for the specified LTL properties.

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.

1.2 Related Work

Chapter 2

Preliminaries

Model checking is a formal verification technique that systematically analyses whether the system under consideration satisfies a set of specified properties. It then either returns that the system fulfills the desired properties or outputs a counterexample if a property is violated. The resulting counterexample offers useful information for debugging purposes. Two parameters are crucial for model checking in order to obtain an expressive and valuable outcome: the *model* of the system under consideration and the formal description of the *properties* the model is to be checked for.

2.1 System Model

An important aspect in model checking is the model of the system under consideration. A model describes the behavior of the system. The more accurate the model represents the system, the more expressive the model checking results are. In this section, we first introduce the general concept of *transition systems* that are commonly used to represent hardware and software systems. In order to describe *hierarchical* system structures relevant to modeling pointer-manipulating programs, we focus on *recursive state machines* that capture the hierarchical (or recursive) nature of method calls in programs.

The states of the transition system consist of heap configurations that accommodate information about the current state of the program execution. As the heap can become unboundedly large, we introduce *hypergraphs* and *graph grammars*, that offer a finite representation for heap configurations, in the second part of this section.

2.1.1 Transition Systems

Transition systems are a model that represent the behavior of a system. A transition system can be regarded as a directed graph, where the nodes of the graph represent

the *states* of the system and the edges indicate the *transition* of one state into another. A state encodes information about the system at a certain moment which are formulated as a set of *atomic propositions*. A transition within a transition system thus reflects that the state of the system changes, e.g. the values of parameters have changed, new variables have been introduced or a process has terminated. These transitions can be annotated by *action names* that capture the possible source of change, e.g. the communication with another system or process, like user interaction or input.

Definition 1.1: Transition System [3]

A *transition system* T is a tuple $(S, Act, \rightarrow, I, AP, L)$ where

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

T is called *finite* if S, Act, and AP are finite.

The set AP of atomic propositions consists of the specified properties a state $s \in S$ might satisfy. The labeling function L maps a state s to a set $L(s) \in 2^{AP}$ stating the atomic proposition $a \in AP$ is satisfied by state s. Based on the set $L(s)$, we can specify that s satisfies a propositional logic formula ϕ if the evaluation induced by $L(s)$ fulfills the formula ϕ. Therefore,

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

The transition relation $\rightarrow$ formally describes how the transition system T evolves starting in an initial state $s_0 \in I$. Thus, the transition $s \xrightarrow{\alpha} s'$ defines that state s evolves to state s' after the action α has been performed. If a state has more than one outgoing transition, the next transition is chosen nondeterministically. This procedure can be continued until a state without any outgoing transitions has been reached. Such a state is called a *terminal state*.

Definition 1.2: Terminal State [3]

A state $s \in S$ in a transition system $T = (S, Act, \rightarrow, I, AP, L)$ is called *terminal*

if and only if

$$\bigcup_{\alpha \in Act} \{s' \in S | s \xrightarrow{\alpha} s'\} = \emptyset.$$

The resulting sequence of executed transitions starting in an initial state $s_0 \in I$ and either ending in a terminal state $s \in S$ or infinitely prolonging, is called an *execution* of the transition system T.

Definition 1.3: Execution

Let $T = (S, Act, \rightarrow, I, AP, L)$. A *finite execution* of T is an alternating sequence

$$s_0 \alpha_1 s_1 \alpha_2 \ldots \alpha_n s_n$$

of states and actions such that

- $s_0 \in I$ is an initial state,
- $s_i \xrightarrow{\alpha_{i+1}} s_{i+1}$ for all $0 \leq i < n$, where $n \geq 0$, and
- s_n is a terminal state.

n is also called the *length* of the execution. An *infinite execution* of T is an infinite, alternating sequence

$$s_0 \alpha_1 s_1 \alpha_2 s_2 \alpha_3 \ldots$$

of states and actions such that

- $s_0 \in I$ is an initial state and
- $s_i \xrightarrow{\alpha_{i+1}} s_{i+1}$ for all $0 \leq i$.

Definition 1.4: Reachable States [3]

A state $s \in S$ in a transition system $T = (S, Act, \rightarrow, I, AP, L)$ is called *reachable* in T if there exists an execution of the form

$$s_0 \alpha_1 s_1 \alpha_2 \ldots \alpha_n s_n = s.$$

$Reach(T)$ denotes the set of all reachable states in T.

For our purpose of model checking pointer-manipulating programs, we focus on the states and the attached atomic propositions of the transition system under consideration, hence, we omit the actions. Multiple transitions between two states with

different actions are thus summarized into a single transition. Therefore, the notion of an execution shifts to the notion of *paths* that denote sequences of states that are visited throughout a run.

Definition 1.5: Paths [3]

A *finite path* π of a transition system $T = (S, Act, \rightarrow, I, AP, L)$ is a finite sequence

$$s_0 s_1 \dots s_n$$

such that

- $s_0 \in I$ is an initial state,
- $s_i \in \bigcup_{\alpha \in Act} \{s \in S | s_{i-1} \xrightarrow{\alpha} s\}$ for all $0 < i \leq n$, where $n \geq 0$, and
- s_n is a terminal state.

An *infinite path* π is an infinite sequence

$$s_0 s_1 s_2 \dots$$

such that

- $s_0 \in I$ is an initial state and
- $s_i \in \bigcup_{\alpha \in Act} \{s \in S | s_{i-1} \xrightarrow{\alpha} s\}$ for all $i > 0$.

$Paths(T)$ denotes the set of all paths in T.

For a path π, $\pi[i]$ denotes the ith state of π, while $\pi[..i]$ and $\pi[i..]$ denote the ith prefix and the ith suffix of π, respectively. Paths display the order of states that are traversed throughout an execution of a transition system. 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 π.

Definition 1.6: Trace [3]

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

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

The *trace* of the infinite path $\pi = s_0 s_1 \dots$ is defined as

$$trace(\pi) = L(s_0)L(s_1)\dots.$$

$Traces(s)$ denotes the set of traces of paths starting in state s and $Traces(T)$ denotes the set of traces of the initial states of a transition system T.

The condition that the transition system T does not have any terminal states is not a restriction since for every transition system it is possible to construct an equivalent one without terminal states. This is achieved by adding a new state s_{stop} with a self-loop to the transition system to which all terminal states have a transition. Thus, the resulting system does not contain any terminal states. In the following we assume that a transition system does not have any terminal states.

Transition systems are a valid model for computer programs as they represent the control flow of the program and thus reflect the program execution. However, our current model does not consider the hierarchical or even recursive nature of programs containing (recursive) method calls. A transition system would hence depict all states of the program execution in a flat setting where method environments are not differentiated. In order to represent the hierarchical structure of method calls, we employ the concept of *recursive state machines* described in the next section.

2.1.2 Recursive State Machines

Often computer programs do not only consist of a linear sequence of commands, but also generate (recursive) calls to methods. Therefore, the execution of these programs contains call- and return-statements to different sections of the input program. In order to capture this hierarchical (or recursive) structure of the system, we introduce the notion of *recursive state machines*, as defined in [1], that encapsulate each method in an own *component*. Each component consists of a set of *nodes* that are the states of the model and *boxes* that are each mapped to a component in the recursive state machine. A box can be understood as an interface with entry and exit nodes that models the transition into another method environment, e.g. 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.

Definition 1.7: Recursive State Machine [1]

A *recursive state machine* (RSM) $\mathcal{A}$ over a finite alphabet Σ is given by a tuple $(A_1, ..., A_k)$, where each *component state machine* (CSM) $A_i = (N_i \cup B_i, Y_i, En_i, Ex_i, \delta_i)$, $1 \leq i \leq k$, consists of

- a set N_i of *nodes* and a (disjoint) set B_i of *boxes*,

- a *labeling* $Y_i : B_i \mapsto \{1, ..., k\}$ that assigns to every box an index $j \in \{1, ..., k\}$ referring to one of the component state machines $A_1, ..., A_k$,
- a set of *entry nodes* $En_i \subseteq N_i$,
- a set of *exit nodes* $Ex_i \subseteq N_i$, and
- a *transition relation* δ_i, where transitions are of the form (u, σ, 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 $j = Y_i(b)$,
  - the label σ is in Σ, 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 $j = Y_i(b)$.

A sample RSM with three components A_1, A_2, A_3 is depicted in Figure 2.1. The nodes drawn at the border of the components represent the entry and exit nodes, respectively. The arrows depict the transitions between the 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 and 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 or from component A_2 to A_3, respectively. This can be understood as an invocation of program methods 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 $\mathcal{A} = (A_1, ..., A_k)$, we first describe the global relation between its component state machines $A_i, 1 \leq i \leq k$. A *global state* of an RSM is a sequence of boxes ending in a node of a component.

Definition 1.8: (Global) State [1]

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

A well-formed state $(b_1, ..., b_r, u)$ of an RSM $\mathcal{A} = (A_1, ..., A_k)$ 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}.

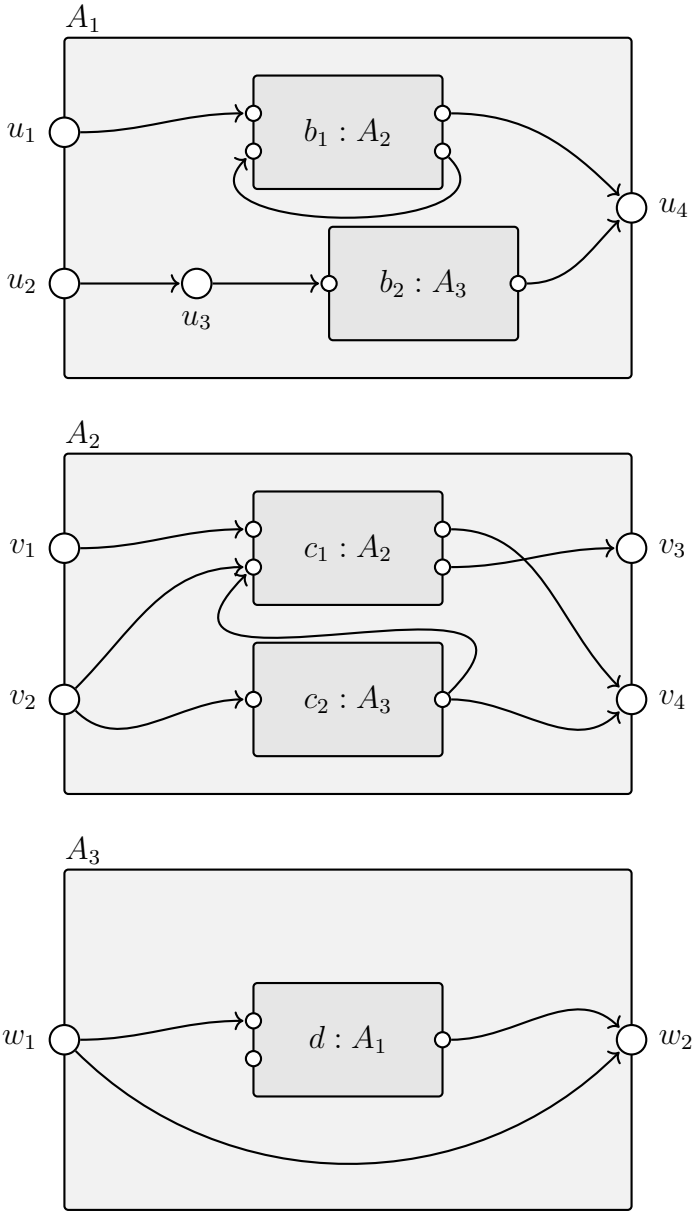


Figure 2.1: A sample recursive state machine. Adapted from [1].

Consider the sample RSM given in Figure 2.1. A global state in the sample RSM is given by

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

where the sequence of boxes protocols 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 Y_i coincides with the component referred to by the next box

in the sequence, e.g., $Y_1(b_1) = A_2$, which is exactly the component in which the following box c_1 is defined.

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

Definition 1.9: (Global) Transition Relation [1]

Let $s = (b_1, ..., b_r, u) \in Q$ be a state with $u \in N_j$ and $b_r \in B_m$ for an RSM $\mathcal{A} = (A_1, ..., A_k)$. A *(global) transition relation* δ 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 $s' = (b_1, ..., b_r, u')$.
2. $(u, \sigma, (b', e)) \in \delta_j$ for a box b' of A_j and $s' = (b_1, ..., b_r, b', e)$.
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 $s' = (b_1, ..., b_{r-1}, u')$.
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 $s' = (b_1, ..., b_{r-1}, b', e)$.

Definition 1.9 defines the possible kinds of transitions between global states $s, s' \in Q$ of an RSM $\mathcal{A}$. Consider the RSM given in Figure 2.1. For each case depicted in Definition 1.9, we can find an example in order to 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 δ, a valid global transition would be $((b_2, d, u_2), \sigma, (b_2, d, u_3)) \in \delta$.

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 regarding the global state (b_1, c_1, c_2, d, u_3) which is located in component A_1. The local transition relation δ_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 δ_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 Σ within the concept of a *labeled transition system* $T_{\mathcal{A}}$, which encodes the execution of $\mathcal{A}$.

Definition 1.10: Labeled Transition System [1]

For an RSM $\mathcal{A} = (A_1, ..., A_k)$, the *labeled transition system* (LTS) $T_{\mathcal{A}} = (Q, \Sigma, \delta)$ consists of

- a set of global states Q,
- a finite alphabet Σ, and
- a global transition relation δ.

The LTS of an RSM $\mathcal{A}$ is also called the *unfolding* of $\mathcal{A}$.

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 Act of actions, 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 1.1).

2.1.3 Heap Representation

After describing recursive state machines as a model for model checking hierarchical programs, we now focus on the representation of the *states* in the transition system. The states under consideration are *heap configurations* holding information on heap objects, program variables, and selectors. Heap configurations are represented by graphs as described in [6], where vertices represent heap objects and edges depict selectors and the mapping of program variables to heap objects. Figure 2.2 illustrates a heap configuration for a doubly-linked list. The list consists of five elements represented by the round vertices of the graph. The selectors `next` and `prev` are represented by the edges of the graph. Furthermore, the program variables `head` and `tail` are attached to the first and the last vertex of the list, respectively.

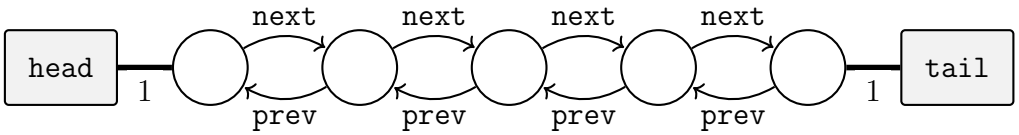


Figure 2.2: Heap configuration of a doubly-linked list. Adapted from [6].

Pointer-manipulating operations are represented by graph transformations. For instance, executing the operation `head := tail.prev` on the heap configuration given in Figure 2.2 results in the heap configuration shown in Figure 2.3.

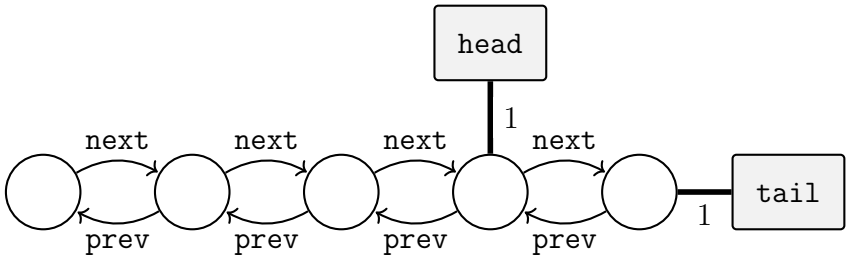


Figure 2.3: Modified heap configuration after pointer-operation.

Over the course of the program execution the size of the heap can become unboundedly large, e.g. when new objects are added to the heap. In order to avoid an unbounded size of the heap representation, we exploit the concept of *hypergraphs*. Hypergraphs are similar to the common graph except that they allow for the graph to contain *abstracted* subgraphs. These subgraphs are connected to the concrete part of the heap by *hyperedges*. Hyperedges differ from commonly known edges in the property that they connect arbitrarily many vertices, instead of only two. The number of vertices a hyperedge connects is captured in its *rank*.

In order to express the abstracted parts of the heap, we require a *ranked alphabet* $\Sigma = \Sigma_N \uplus \Sigma_T$, where Σ_N denotes a 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 Σ_T describe *concrete* heaps that do not contain an abstract part such as the heap depicted in Figure 2.2.

Definition 1.11: Hypergraph [5]

Given a finite ranked alphabet $\Sigma = \Sigma_N \uplus \Sigma_T$ with associated ranking function

$rk : \Sigma \to \mathbb{N}$. A (labeled) hypergraph over Σ is a tuple

$$H = (V, E, att, lab, ext)$$

where

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

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

An example for a hypergraph with an abstracted subgraph is depicted in Figure 2.4. Here, the abstracted subgraph is represented by the hyperedge labeled DLL. This indicates that the hyperedge replaces a doubly-linked list of arbitrary length.

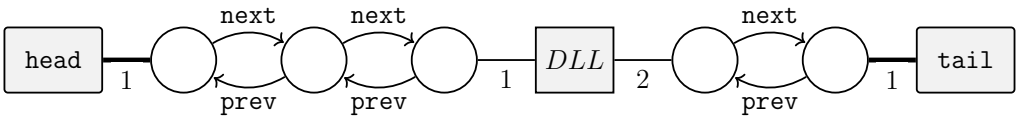


Figure 2.4: A doubly-linked list with abstracted subgraph represented as a hypergraph. Adapted from [6].

In order to obtain all possible heap configurations represented by an abstract subgraph, we require the concept of *graph grammars* or more specifically *hyperedge replacement grammars*. Graph grammars are similar to string grammars and define a set of *rules* for graph manipulation. They prescribe how nonterminals can be replaced by hypergraphs. Continuously applying grammar rules to a hypergraph gradually replaces nonterminals by hypergraphs so that concrete heap configurations can be reached eventually.

Definition 1.12: Hyperedge Replacement Grammar [5]

A *hyperedge replacement grammar* G over the ranked alphabet Σ is a set of production rules of the form $X \to R$, where $X \in \Sigma_N$ is a nonterminal that

forms the left-hand side and $R \in HG_\Sigma$ is the rule graph, a hypergraph with $|ext_R| = rk(X)$, the right-hand side pf the rule.

The language of a hyperedge replacement grammar contains all concrete hypergraphs that are obtained by repeatedly applying the production rules to a given hypergraph.

An example for a hyperedge replacement grammar, that describes the language of all doubly-linked lists with at least two elements, is given in Figure 2.5. The first production rule recursively adds an element to the existing list introducing a new nonterminal DLL in order to allow for adding more elements during another production. The second rule terminates the production by replacing the nonterminal DLL by a concrete graph.

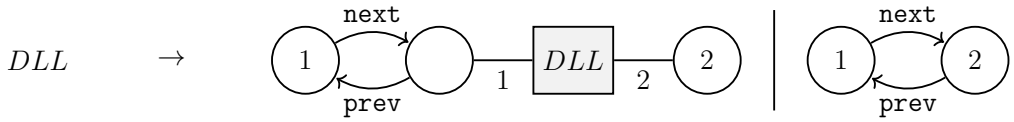


Figure 2.5: A hyperedge replacement grammar for doubly-linked lists. Adapted from [6].

Let us consider the hypergraph from Figure 2.2. We have two options to apply the grammar from Figure 2.5 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.

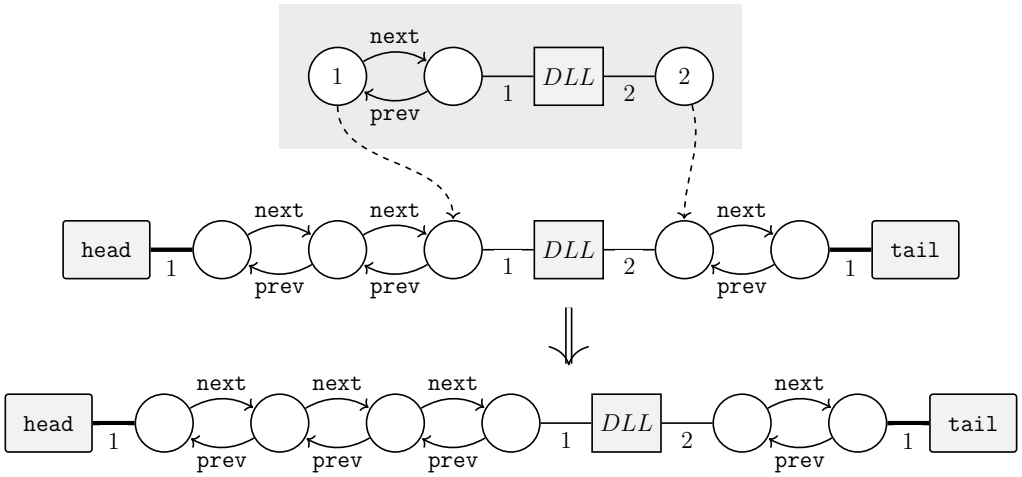


Figure 2.6: By applying the first production rule from Figure 2.5 a list element is added to the (abstract) hypergraph.

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)

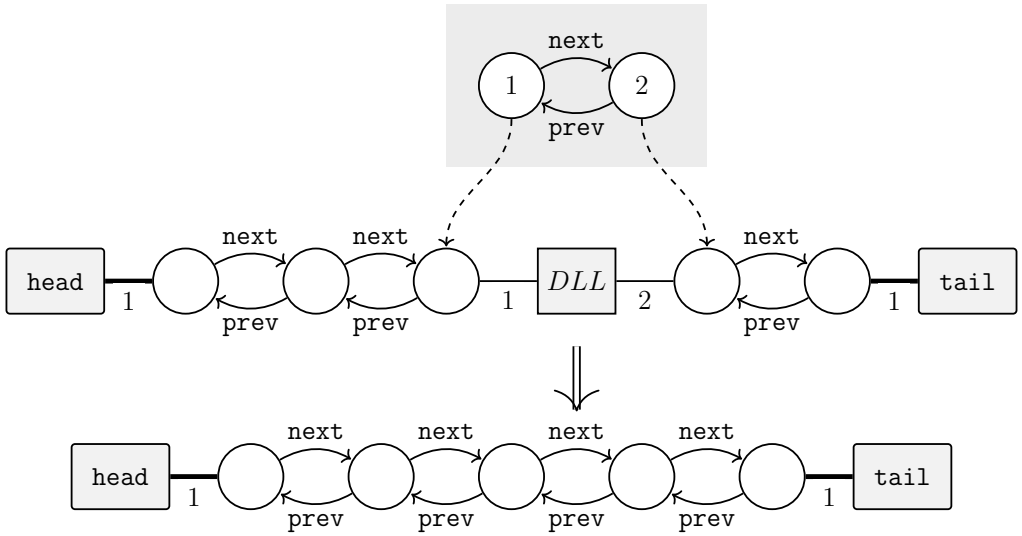


Figure 2.7: By applying the second production rule from Figure 2.5 a concrete hypergraph is obtained.

concrete subgraph. A concretisation step can yield more than one concrete hypergraph if several production rules are applicable. Thus, abstraction of subgraphs yield an over-approximation of the current set of concrete hypergraphs, since information is lost during abstraction. It is not always possible to uniquely identify the initial hypergraph of which the abstracted graph has been derived of. Therefore, concretisation needs to consider all possible hypergraphs. In fact, the application of the production rules of the hyperedge replacement grammar for doubly-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 hence allows us to represent possibly unboundedly large graphs in a finite manner.

Concluding with the concept of hypergraphs and hyperedge replacement grammars we specified the relevant framework for model checking pointer-manipulating programs. We model the program execution by recursive state machines capturing the hierarchical nature of method calls, while hypergraphs offer a finite representation of heap configurations that constitute the states of the model under consideration. The second ingredient to model checking is the formal definition of the properties the model is to be validated for. Here, we focus on *linear temporal logic* described in the following section.

2.2 Linear Temporal Logic

First proposed by Pnueli in 1977, *Linear Temporal Logic* (LTL) is a modal temporal logic suited to describe *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 2.1: Linear-Time Property [3]

A *linear-time property* over the set of atomic propositions AP is a subset of $(2^{AP})^{\omega}$.

Thus, a linear-time property can be interpreted as a language of 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 if and only if all traces of T are included in the set P meaning that every trace of T is a word in the language induced by $P \subseteq (2^{AP})^{\omega}$.

Definition 2.2: Satisfaction Relation for Linear-Time Properties [3]

Let P be a linear-time property over AP and $T = (S, Act, \rightarrow, I, AP, L)$ a transition system. Then, T *satisfies* P, denoted $T \models P$, iff $Traces(T) \subseteq P$. A state $s \in S$ satisfies P, denoted $s \models P$, iff $Traces(s) \subseteq P$.

Linear-time properties can be specified by LTL formulae that encode temporal specifications for paths. In LTL, time is understood as a discrete time-unit where a point in time is followed by a single time-unit. In contrast to LTL, CTL (Computation Tree Logic) considers tree-like paths which can split into alternative courses. Here, we focus on LTL formulae.

LTL formulae are composed of three components: the boolean operators *negation* ($\neg$) and *conjunction* ($\wedge$), the temporal operators *next* ($\bigcirc$) and *until* ($\boldsymbol{U}$), 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 set of LTL formulae is defined as follows:

Definition 2.3: Syntax of LTL [3]

Given a set AP of atomic propositions with $a \in AP$, *LTL formulae* are recur-

sively defined by

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

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 can be derived using the operators given in Definition 2.3 as follows:

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

Before formally defining the satisfaction relation for LTL formulae and a transition system T, we first constitute an intuitive understanding of temporal operators by visualizing their semantics in Figure 2.8.

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

Definition 2.4: Semantics of LTL (Interpretation over Words) [3]

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

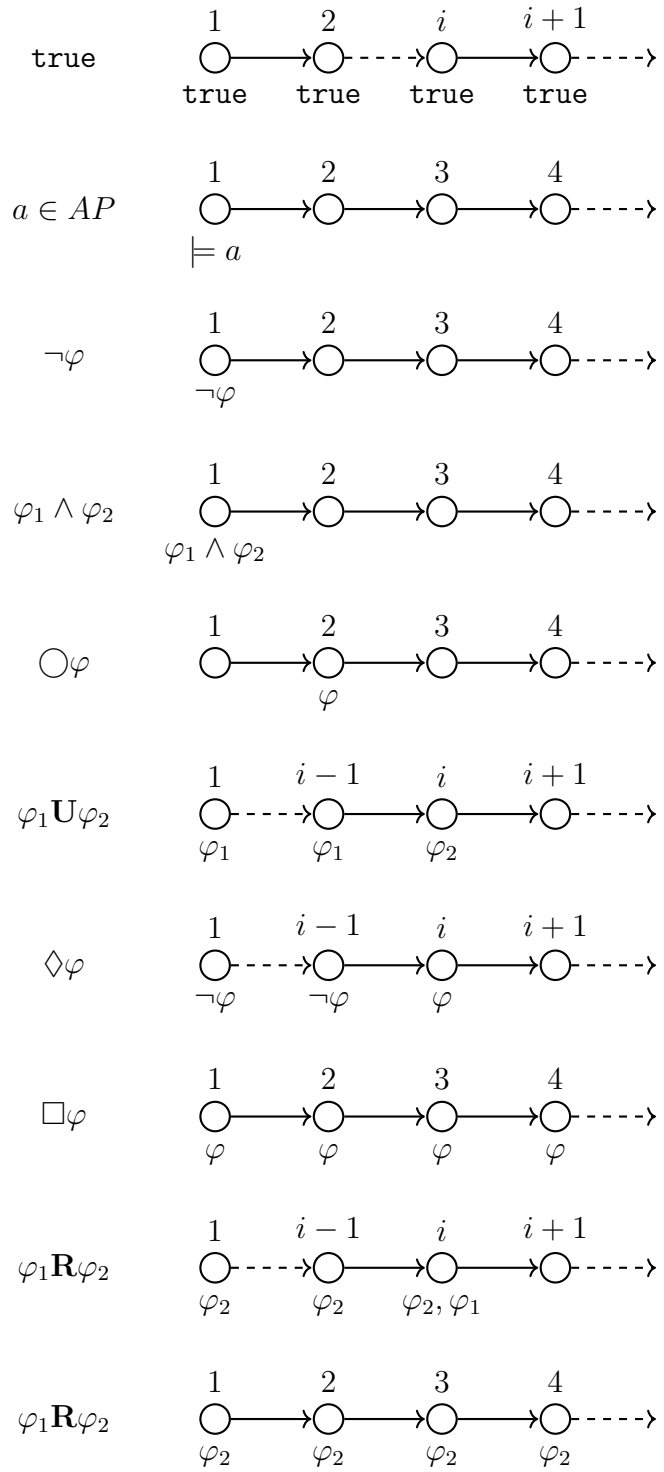
$$Words(\varphi) = \{\sigma \in (2^{AP})^\omega \mid \sigma \models \varphi\}$$

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

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

The interpretation of LTL formulae over words can be used to describe the semantics of LTL formulae over paths and states of a transition system T.

Definition 2.5: Semantics of LTL over Paths and States [3]



$\pi \models$ `true` since every state satisfies `true`. Thus, $\varphi =$ `true` is fulfilled by every path π.

$\pi \models a$ if the first state of path π satisfies the proposition a.

$\pi \models \neg\varphi$ if the first state of path π does not satisfy φ.

$\pi \models \varphi_1 \wedge \varphi_2$ if the first state of path π satisfies both formulae φ_1 and φ_2 at the same time.

$\pi \models \bigcirc\varphi$ if the next state in path π satisfies the formula φ.

$\pi \models \varphi_1 \mathbf{U} \varphi_2$ if φ_1 holds for states 1 to $i-1$ and there exists a state i that satisfies φ_2.

$\pi \models \Diamond\varphi$ if there exists a state i on path π which satisfies φ.

$\pi \models \Box\varphi$ if all states on path π satisfy φ.

$\pi \models \varphi_1 \mathbf{R} \varphi_2$ if φ_2 holds for states 1 to $i-1$ and there exists a state i that satisfies φ_1.

$\pi \models \varphi_1 \mathbf{R} \varphi_2$ if φ_2 holds for all states of π.

Figure 2.8: Intuitive semantics of temporal operators.

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

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

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

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

T satisfies φ, denoted $T \models \varphi$, if $Traces(T) \subseteq Words(\varphi)$.

From Definition 2.5, it follows that

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

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

Definition 2.6: Semantics of LTL [3]

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

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

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

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

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

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

$$\bigcirc\{\texttt{SLList}\}$$

where `SLList` is assumed to be an atomic proposition describing that the heap is a singly-linked list. Hence, the formula states that the heap of the next state is a singly-linked list. Another example is the formula

$$\Box\{\texttt{SLList}\}$$

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 $\Box\{\texttt{SLList}\}$. The formula

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

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 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* (PNF), where negations are only allowed on the level of literals [3].

Definition 2.7: 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

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

The existence of an equivalent PNF formula for every LTL formula is due to the following equivalences that allow to push negations inside [5]:

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

As an example consider the LTL formula

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

where `terminated` and `SLList` are atomic propositions. `terminated` describes that a state is a terminating state and `SLList` states that the heap of a state is a singly-linked list. An equivalent formula in PNF is achieved by the following equivalences:

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

Chapter 3

Hierarchical Model Checking

One of the main challenges in model checking programs with method calls is that we do not only need to consider the state space of the main method, but also the state spaces induced by method executions, denoted as *procedure state spaces*. Here, we face two main difficulties:

- How can we finitely represent the state space of a program with recursive method calls, at best avoiding repetitions in the state space?

- How can we efficiently model check procedure state spaces, at best avoiding checking a procedure state space multiple times?

In this chapter, we introduce two approaches to model checking LTL properties for pointer-manipulating programs with method calls. The underlying state space of procedural programs is a hierarchical one where each procedure contributes an own state space that is connected to nodes of other state spaces reflecting method invocation. In a flat setting, where hierarchy is not actively considered, this corresponds to an edge connecting the calling state with the "entry" state of the procedure state space. For the flat setting, the first section of this chapter presents an on-the-fly LTL model checking approach that constructs a proof structure based on a set of tableaux rules that reflect the semantics of LTL [4].

As a flat state space does not capture the modularity of methods, we represent the state space as a recursive state machine. In the second section of this chapter, we present an LTL model checking approach for RSMs by Yannakakis et al. which is based on the automata-based approach by Vardi and Wolper as described in [8].

Based on the presented algorithms, we introduce our modified approaches to hierarchical model checking in the sequel of this thesis.

3.1 Tableaux Construction

This section presents the on-the-fly approach by Grumberg et al. that constructs a *proof structure* based on a set of *tableaux rules* in order to show whether a formula φ is satisfied by a transition system T.

A proof structure is a directed graph (V, E), where the set of vertices V are composed of a set of *assertions* Λ and the set E of edges contains the edge (λ_1, λ_2) between two assertions λ_1 and λ_2 if the underlying tableaux contains the rule $\frac{\lambda_1}{\lambda_2}$.

Definition 1.1: Proof Structure [4]

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

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

The underlying tableaux rules for the proof structure are specified in the following. They model the semantics of LTL.

$$(R^{\models}) \quad \frac{s \vdash \Phi \cup \{a\}}{\texttt{true}} \text{ if } s \models a$$

$$(R^{\not\models}) \quad \frac{s \vdash \Phi \cup \{a\}}{s \vdash \Phi} \text{ if } s \not\models a$$

$$(R^{\vee}) \quad \frac{s \vdash \Phi \cup \{\varphi_1 \vee \varphi_2\}}{s \vdash \Phi \cup \{\varphi_1\} \cup \{\varphi_2\}}$$

$$(R^{\wedge}) \quad \frac{s \vdash \Phi \cup \{\varphi_1 \wedge \varphi_2\}}{s \vdash \Phi \cup \{\varphi_1\} \qquad s \vdash \Phi \cup \{\varphi_2\}}$$

$$(R^{\mathbf{U}}) \quad \frac{s \vdash \Phi \cup \{\varphi_1 \mathbf{U} \varphi_2\}}{s \vdash \Phi \cup \{\varphi_2, \varphi_1\} \qquad s \vdash \Phi \cup \{\varphi_2, \bigcirc(\varphi_1 \mathbf{U} \varphi_2)\}}$$

$$(R^{\mathbf{R}}) \quad \frac{s \vdash \Phi \cup \{\varphi_1 \mathbf{R} \varphi_2\}}{s \vdash \Phi \cup \{\varphi_2\} \qquad s \vdash \Phi \cup \{\varphi_1, \bigcirc(\varphi_1 \mathbf{R} \varphi_2)\}}$$

$$(R^{\bigcirc}) \quad \frac{s \vdash \{\bigcirc\varphi_1, \ldots, \bigcirc\varphi_n\}}{s_1 \vdash \{\varphi_1, \ldots, \varphi_n\} \qquad \ldots \qquad s_m \vdash \{\varphi_1, \ldots, \varphi_n\}}$$

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

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

The vertices V of a proof structure (V, E) are assertions of the form $s \vdash \Phi$, where s is a state in T and Φ 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 can be interpreted as a verification goal that aims at proving that $s \models \bigvee_{\varphi \in \Phi} \varphi$. In order to do so, the assertion is broken down into subgoals according to the tableaux rules. By proving a sequence of subgoals the validity of the assertion λ can be concluded from the validity of the subgoals. Hence, the proof structure of an assertion λ contains all subgoals of λ.

The rules $(R^{\mathbf{U}})$ and $(R^{\mathbf{R}})$ can introduce cycles into the proof structure if φ_1 is fulfilled for every state in the underlying transition system for formulae of the form $\varphi_1 \mathbf{U} \varphi_2$ or $\varphi_1 \mathbf{R} \varphi_2$ φ_1, while no state fulfills φ_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 rules specified in the tableaux cannot introduce cycles as their application reduces the size of the formulae.

In the following, we describe when a proof structure (V, E) for an assertion λ can be concluded to be *successful* [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 λ is called successful if $\lambda = \texttt{true}$.
- An infinite path $\lambda_1 \lambda_2 \ldots$ 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 lead as well as every of its infinite paths is successful.

Theorem 1.2 states that the tableaux construction is indeed a suitable procedure to model check a transition system T for an LTL formula φ 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 1.2: Correctness of the Tableaux Construction [4]

Given a concrete transition system $T = (S, Act, \rightarrow, I, AP, L)$ with $s \in S$ and an LTL formula φ. Let (V, E) be the proof structure for $s \vdash \{\varphi\}$. Then it holds that $s \models \varphi$ iff (V, E) is successful.

Consider a method for reversing a singly-linked list given by

```
public static SLList reverse(SLList head) {

SLList reversedList = null;
SLList current = head;

while (current != null) {
SLList next = current.next;
current.next = reversedList;
reversedList = current;
current = next;
}

return reversedList;
}
```

Figure 3.1 shows the state space of reversing a singly-linked list with two elements according to the method `reverse`. For this state space, we want to check whether the formula $\varphi = \Box\Diamond\{\texttt{terminated}\} \rightarrow \Box\Diamond\{\texttt{SLList}\}$ is satisfied. From Section 2.2 we know that

$$\varphi \equiv \texttt{true}\ \mathbf{U}\ (\neg\texttt{true}\ \mathbf{R}\ \neg\{\texttt{terminated}\}) \vee (\neg\texttt{true}\ \mathbf{R}\ (\texttt{true}\ \mathbf{U}\ \{\texttt{SLList}\}))$$

in PNF.

3.2 Automata-Based Model Checking

Recursive programs can be modeled as recursive state machines that capture the hierarchical nature of the underlying state space. Based on RSMs, LTL model checking can be performed to check whether an RSM, and thus the represented system, satsfies a temporal property. This section presents the basic ideas of the automata-theoretic approach to model checking transition systems for LTL specifications as introduced in [8] by Vardi and Wolper. The algorithm can be adapted for RSMs. For an LTL formula φ and a transition system T the algorithm by Vardi and Wolper either returns "yes" if $T \models \varphi$ or "no" and a failure trace of the path that violates

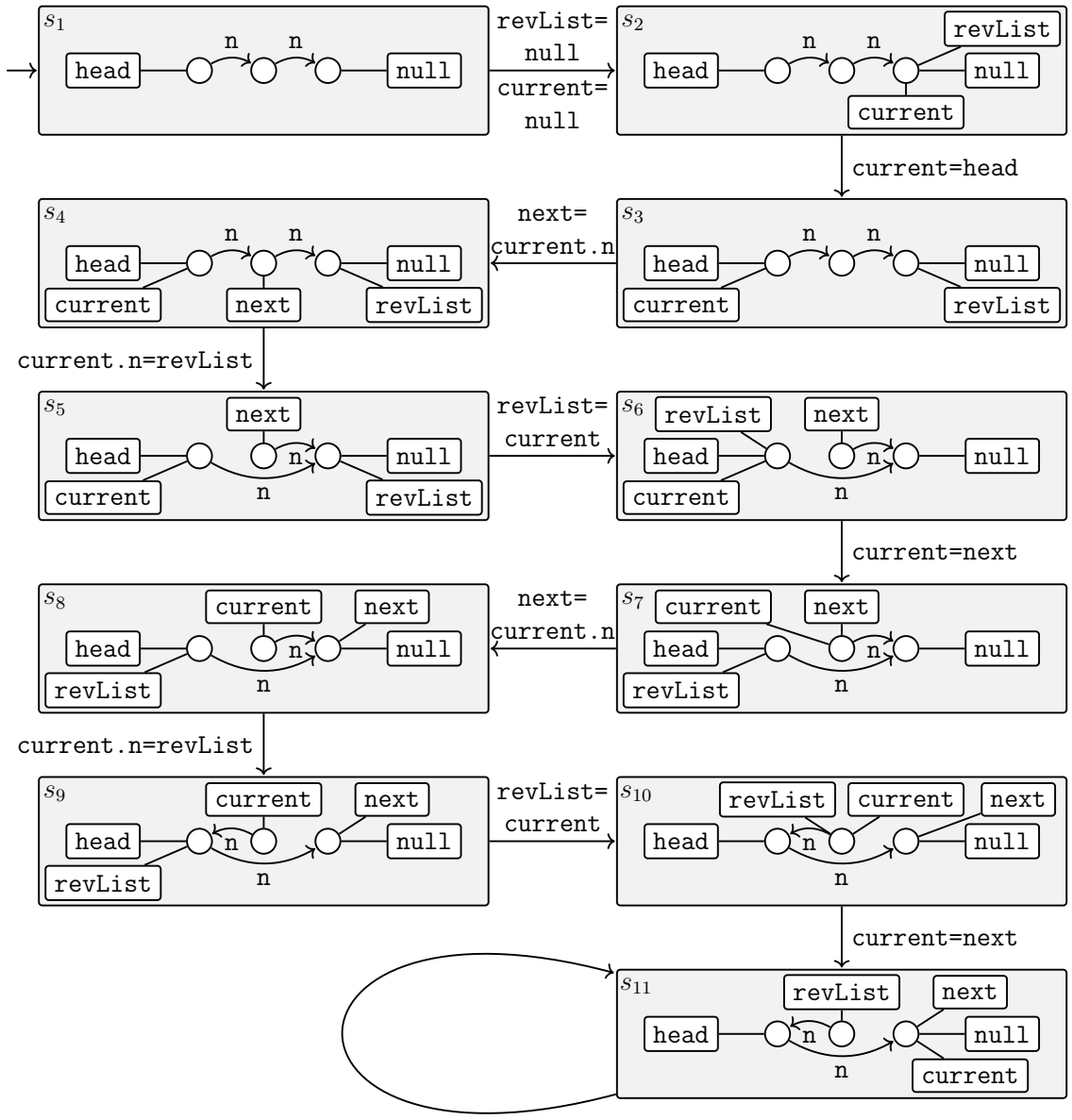


Figure 3.1: State space for reversing a singly-linked list. Adapted from [5].

the formula φ.

The approach is based on the fact that every LTL formula φ can be represented by a *nondeterministic Büchi automaton*.

Definition 2.1: Nondeterministic Büchi Automaton [3]

A *nondeterministic Büchi automaton* (NBA) $\mathcal{B}$ is a tuple $\mathcal{B} = (Q, \Sigma, \delta, Q_0, F)$ where

- Q is a finite set of states,
- Σ is an alphabet,
- $\delta : Q \times \Sigma \to 2^Q$ is a transition function,
- $Q_0 \subseteq Q$ is a set of initial states, and
- $F \subseteq Q$ is a set of *accept* or final states, called the *acceptance set*.

A run $\sigma = B_0 B_1 B_2 \cdots \in \Sigma^\omega$ of an NBA $\mathcal{B}$ is an infinite sequence $q_0 q_1 q_2 \ldots$ of states in $\mathcal{B}$ such that $q_0 \in Q_0$ and $q_i \xrightarrow{B_i} q_{i+1}$ for $i \geq 0$. A run $q_0 q_1 q_2 \ldots$ is *accepting* if $q_i \in F$ for infinitely many indices $i \in \mathbb{N}$. The *accepted* language of $\mathcal{B}$ is

$$\mathcal{L}_\omega(\mathcal{B}) = \{\sigma \in \Sigma^\omega | \text{ there exists an accepting run for } \sigma \text{ in } \mathcal{B}\}.$$

The size $|\mathcal{B}|$ of $\mathcal{B}$ is defined as the number of states and transitions in $\mathcal{B}$.

Theorem 2.2

For any LTL formula φ over AP there exists an NBA $\mathcal{B}_\varphi$ with

$$Words(\varphi) = \mathcal{L}(\mathcal{B}_\varphi)$$

which can be constructed in time and space $2^{\mathcal{O}(|\varphi|)}$. [3]

The automata-based approach for model checking an LTL formulae φ for a transition system T is based on the idea to find a path π in T that satisfies the formula $\neg\varphi$. If such path is found, then it follows that $T \not\models \varphi$ since a path is found that satisfies the negation of the desired formula φ. Hence, the answer to the model checking procedure is "no" and π is returned as a failure trace. Otherwise, it can be concluded that $T \models \varphi$ as no path is found to satisfy $\neg\varphi$. This conclusion is valid, since

$$\begin{aligned} T \models \varphi \text{ iff } & Traces(T) \subseteq Words(\varphi) \\ \text{iff } & Traces(T) \cap ((2^{AP})^\omega \; Words(\varphi)) = \emptyset \\ \text{iff } & Traces(T) \cap Words(\neg\varphi) = \emptyset. \end{aligned}$$

Therefore, for NBA $\mathcal{B}_{\neg\varphi}$

$$T \models \varphi \text{ iff } Traces(T) \cap \mathcal{L}(\mathcal{B}_{\neg\varphi}) = \emptyset$$

as $\mathcal{L}(\mathcal{B}_{\neg\varphi}) = Words(\neg\varphi)$.

Algorithm 1 Automata-Based LTL Model Checking

Input: finite transition system T, LTL formula φ
Output: "yes", if $T \models \varphi$; "no" and a counterexample, otherwise
  Construct an NBA $\mathcal{B}_{\neg\varphi}$ for $\neg\varphi$.
  Construct the product transition system $T' = T \otimes \mathcal{B}_{\neg\varphi}$.
  if $\exists$ path π in T' satisfying the acceptance condition of $\mathcal{B}$ **then**
    return "no" and counterexample
  else
    return "yes"
  end if

The automata-based approach is described in Algorithm 1. In a first step, an NBA $\mathcal{B}_{\neg\varphi}$ is constructed for the formula $\neg\varphi$. Thereafter, the product transition system $T' = T \otimes \mathcal{B}_{\neg\varphi}$ is constructed from the transition system T and the NBA $\mathcal{B}_{\neg\varphi}$. The problem of determining whether a path π exists in T that satisfies the acceptance condition of the product transition system T', can be reduced to checking for emptiness of the intersection of the sets of $Traces(T)$ and $\mathcal{L}(\mathcal{B}_{\neg\varphi})$. If $Traces(T) \cap \mathcal{L}(\mathcal{B}_{\neg\varphi}) = \emptyset$, then there does not exist any path π in T that satisfies $\neg\varphi$ and hence does not violate φ. However, if the intersection is not empty, then an according path π is detected and φ is violated.

The automata-based approach can be adapted to recursive state machines as described in [1]. In this context, the notion of *recursive Büchi automata* is introduced that augment RSMs with Büchi-acceptance conditions.

In order to improve the performance of the algorithm, it is possible to execute the automata-based approach in an on-the-fly manner. That is, instead of sequentially computing the automata $\mathcal{B}_{\neg\varphi}$ and the product transition system T', both automata are computed on demand as long as no path π is found that satisfies $\neg\varphi$. Hence, the automata do not need to be constructed entirely if a violating path can be detected at an early stage.

However, LTL model checking is still computationally hard and it can be shown that the LTL model checking problem is PSPACE-complete [3].

Theorem 2.3

The LTL model checking problem is PSPACE-complete.

As the automaton-construction of the NBA $\mathcal{L}(\mathcal{B}_{\neg\varphi})$ as well as the product construction are quite costly in practice, we approach automata-based model checking by combining the structures of recursive state machines and the procedures of the *tableaux construction* for model checking LTL formulae. We depict the tableaux construction in the next section. The resulting algorithm from our combined approach is presented in Chapter 5.

Chapter 4

On-The-Fly Hierarchical Model Checking

The tableaux construction is an on-the-fly approach to model checking transition systems for an LTL formula φ. In this chapter, we describe how we adapt the tableaux construction by Grumberg et al. to hierarchical tableaux construction that accounts for model checking of hierarchical structures including procedure state spaces. In a next step, we present the implementation of hierarchical tableaux construction in the ATTESTOR framework and conclusively evaluate our proceedings.

4.1 Algorithm

Given a transition system (or state space) T and a set of LTL formulae Φ, the goal of the hierarchical tableaux construction is to verify whether the state space, including procedure state spaces induced by the method executions, satisfies the temporal properties specified by Φ. We assume the state space to be flat and finite. The basis of the algorithm is the tableaux construction by Grumberg et al. that we extend by the following functionalities:

- Interweave state space construction and state space model checking,
- Model check procedure state spaces, and
- Create model checking contracts for procedure state spaces in order to reuse model checking results that have been computed beforehand.

The algorithm is based on the procedure `generateAndCheck` (cf. Figure 4.1) that validates a set Φ of formulae for an input program based on the algorithm of the ATTESTOR state space generation described in Section 1.1. After initialising the state space and the proof structure, the procedure `generateAndCheck` starts constructing the proof structure based on the tableaux rules defined in Section 3.1. Successor

states are generated on demand if the proof structure needs to be extended to further states due to validating ◯-formulae. State generation involves that the statement of the current state is executed. In contrast to the `Attestor` state space generation, statement execution, and therefore state generation, includes model checking of the generated state. Hence, statements that invoke method executions trigger the model checking of procedure state spaces, that is, the procedure `generateAndCheck` is executed on the calling state of the method and the current set Φ' of formulae. It follows that a hierarchy of calls to `generateAndCheck` is constructed where each level represents the model checking procedure of a method execution. Once state space generation and model checking is completed for a method execution, the successor states and model checking results are returned to the above lying level. Model checking results include information on whether any formula is violated as well as the updated set of formulae for which the successor states need to be checked for. The model checking results are stored in contracts of the form

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

where HC_{in} denotes the input heap configuration for which the set Φ of formulae is to be validated. The tuple (HC_{in}, Φ) is mapped to a tuple of model checking results containing the resulting set Φ' of formulae to be checked for possible successor states, a boolean value $\delta_{0/1}$ that indicates whether model checking was successful, and a failure trace π in case any formulae is found to be violated. Model checking contracts are stored in the context of *procedure contracts* which unambiguously define the executed method. Procedure contracts capture the overall effect of a procedure by defining pairs of pre- and post-conditions. Pre- and post-conditions specify the heap configurations before and after the execution of a procedure, respectively. Details on procedure contracts are described in [7]. Thus, model checking contracts can be used to avoid repetitive model checking of a procedure state space for a set of input formulae.

Together with the set Φ' of formulae, the resulting successor states induce new assertions that are added to the proof structure. If the proof structure is (still) successful, then the new assertions 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 Φ is not satisfied for the complete program state space. Thus, an early termination of the model checking procedure is possible that does not require building the complete state space. The concept of the hierarchical tableaux construction is depicted in Figure 4.1.

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

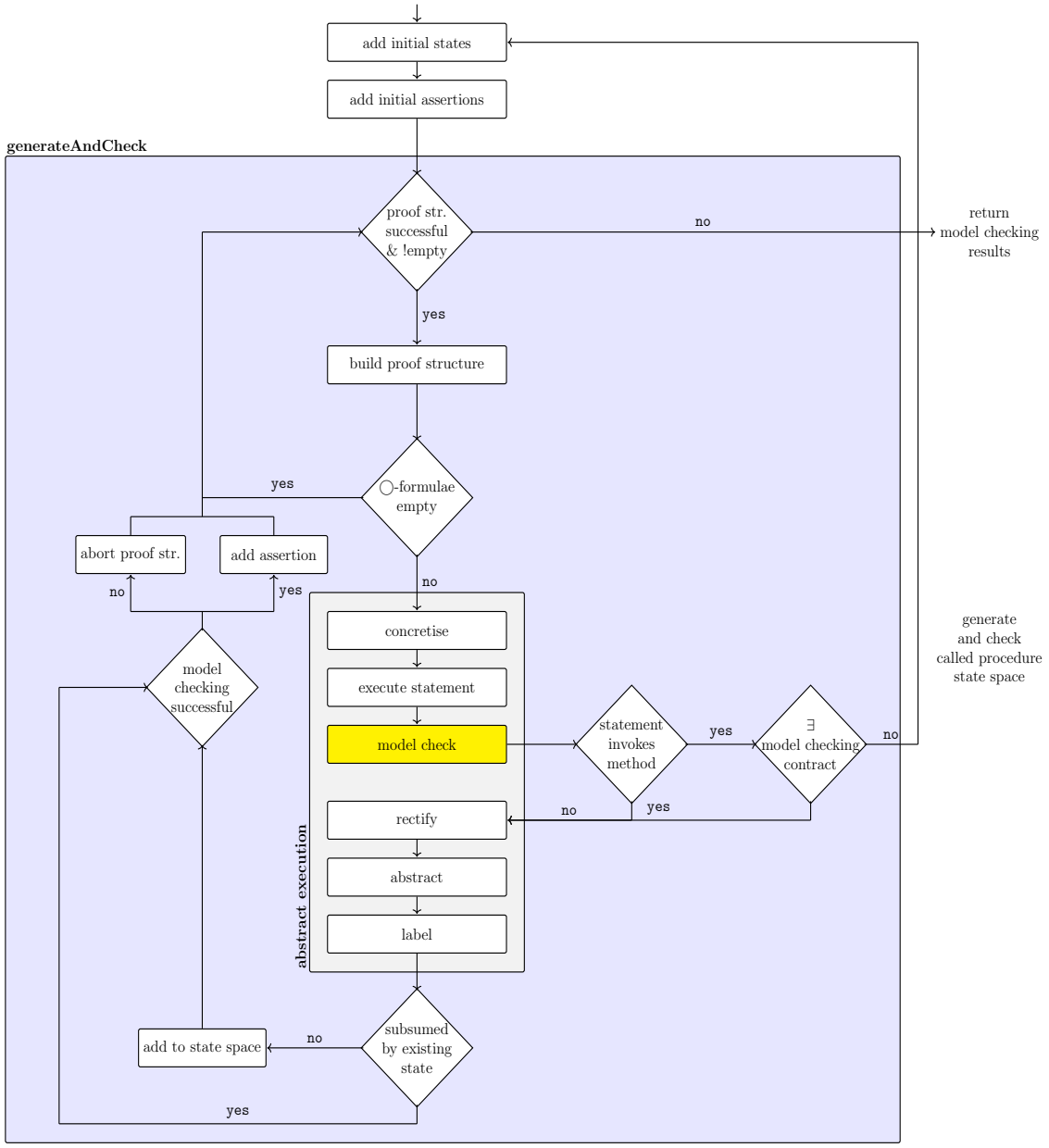


Figure 4.1: On-the-fly hierarchical model checking by a hierarchical tableaux construction.

the state space generation algorithm defined in [2].

4.2 Implementation

We implemented the on-the-fly tableaux construction for hierarchical state spaces within the ATTESTOR framework written in JAVA. The code can be found at `https:`

//github.com/SallyChau/master_thesis/tree/master/attestor. As both state space generation and model checking are performed, the on-the-fly hierarchical model checking algorithm encompasses ATTESTOR's present state space generation and model checking phases. We refer to the new phase as the *on-the-fly model checking phase*.

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 the building the proof structure. Second, we require a possibility to query for a list of successor states for a given state s.

Let us turn to the on-the-fly tableaux construction first. The `ProofStructure` class implements the tableaux construction for a given state space and an LTL formula. By calling the method `build` on a given state space and an LTL formula to be checked, the proof structure is constructed. This method requires a completely generated state space to operate on. This class `OnTheFlyProofStructure` modifies the current implementation 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 reached. Since the underlying state space is not present, the proof structure cannot be continued. Thus, we require the generation of the successors of the current state. At this point, the control is handed to the `StateSpaceGenerator`, where ATTESTOR's state space generation is performed.

By calling the `generate` method of the class `StateSpaceGenerator` the state space generation for the input program is performed and the completely generated state space is returned upon termination. For the on-the-fly approach, we do not require the complete state space, but rather a list of successor states for a given state s. We realize this functionality in the class `OnTheFlyStateSpaceGenerator` which generates the successors to a state returned by the previously constructed proof structure. The list of successors are then communicated back to the proof structure by adding new assertions and reentering the model checking loop (cf. Figure 4.1).

Further adjustments are required in the labeling of states in procedure state spaces which maps an atomic proposition $a \in AP$ to a state s if $s \models a$. Up until now, only the states of the top-level state space are labeled, as procedure state spaces were 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 configuration of the state s is adapted to the scope of

the invoked method such that local variables are excluded. Thus, we obtain a *scoped heap configuration*. 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 configuration, 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, so that the property, that the heap is a singly-linked list, is violated. Thus, the state will not be labeled with the corresponding atomic proposition, resulting in a negative outcome of the model checking procedure despite the fact that the heap is still a singly-linked list outside of the method call. Hence, 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 configuration under consideration might have been scoped multiple times. Thus, the collection of scopes is summarized in the scope hierarchy. Now, before computing the labels for a state under consideration, the scope hierarchy is applied back to the state in reversed order, such that the original heap configuration is restored. Based on the resulting state, the corresponding atomic propositions are determined.

4.3 Evaluation

Model checking of hierarchical state spaces requires model checking of the procedure state spaces. ATTESTOR's current model checking approach only considers the top-level state space for model checking and thus disregards possible erroneous behavior within method executions. The on-the-fly approach on hierarchical model checking solves this gap by model checking procedure state spaces during their generation. Thus, the algorithm successfully interweaves state space generation and state space 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 the complete state space needs to be generated.

However, in case of model checking a procedure state space for multiple distinct LTL formulae, the procedure state space needs to be computed afresh as the on-the-fly approach does not necessarily compute complete state spaces that cannot 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 hierarchical model checking algorithm described in this chapter.

Chapter 5

Hierarchical Model Checking with Recursive State Machines

5.1 Algorithm

5.2 Implementation

5.3 Evaluation

Chapter 6

Benchmarks

6.1 Experimental Setup

Describe Technical details here

6.2 Instances

Describe code examples and properties here

6.3 Result

Table of values

Chapter 7

Conclusion

7.1 Discussion

7.2 Outlook

- hierarchical failure trace and counter example generation, spuriosity - hybrid method between on-the-fly and RSM

Bibliography

[1] Alur, Rajeev, Kousha Etessami and Mihalis Yannakakis: *Analysis of recursive state machines.* In *International Conference on Computer Aided Verification*, pages 207–220. Springer, 2001.

[2] Arndt, Hannah, Christina Jansen, Joost-Pieter Katoen, Christoph Matheja and Thomas Noll: *Let this Graph Be Your Witness!* In *International Conference on Computer Aided Verification*, 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 *Proceedings of Tenth Annual IEEE Symposium on Logic in Computer Science*, pages 388–397. IEEE, 1995.

[5] Heinen, Jonathan: *Verifying Java programs-a graph grammar approach.* Verlag Dr. Hut, 2015.

[6] Heinen, Jonathan, Christina Jansen, Joost-Pieter Katoen and Thomas Noll: *Verifying pointer programs using graph grammars.* Science of Computer Programming, 97:157–162, 2015.

[7] Jansen, Christina and Thomas Noll: *Generating abstract graph-based procedure summaries for pointer programs.* In *International Conference on Graph Transformation*, pages 49–64. Springer, 2014.

[8] Vardi, Moshe Y and Pierre Wolper: *Automata-theoretic techniques for modal logics of programs.* Journal of Computer and System Sciences, 32(2):183–221, 1986.