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**

July 7, 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

# 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 takes a Java program as an input and then generates an abstract state space. Attestor 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 hyperedges. Methods in 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 heap. In a next step, model checking is performed by checking the resulting state space against the 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 a property violation, a counterexample is provided.

Attestor divides up into input, back-end, front-end, and output as depicted in Figure 1.1. Attestor offers several options to specify a verification task. The user defines a Java program as the program to be analyzed. LTL formulae describe the properties that are checked for the program during model checking. Attestor requires graph grammars in order to account for state space abstraction and concretization. Next to the pre-defined grammars in the tool, the user has the option to define custom graph grammars that define the data structures present in the input

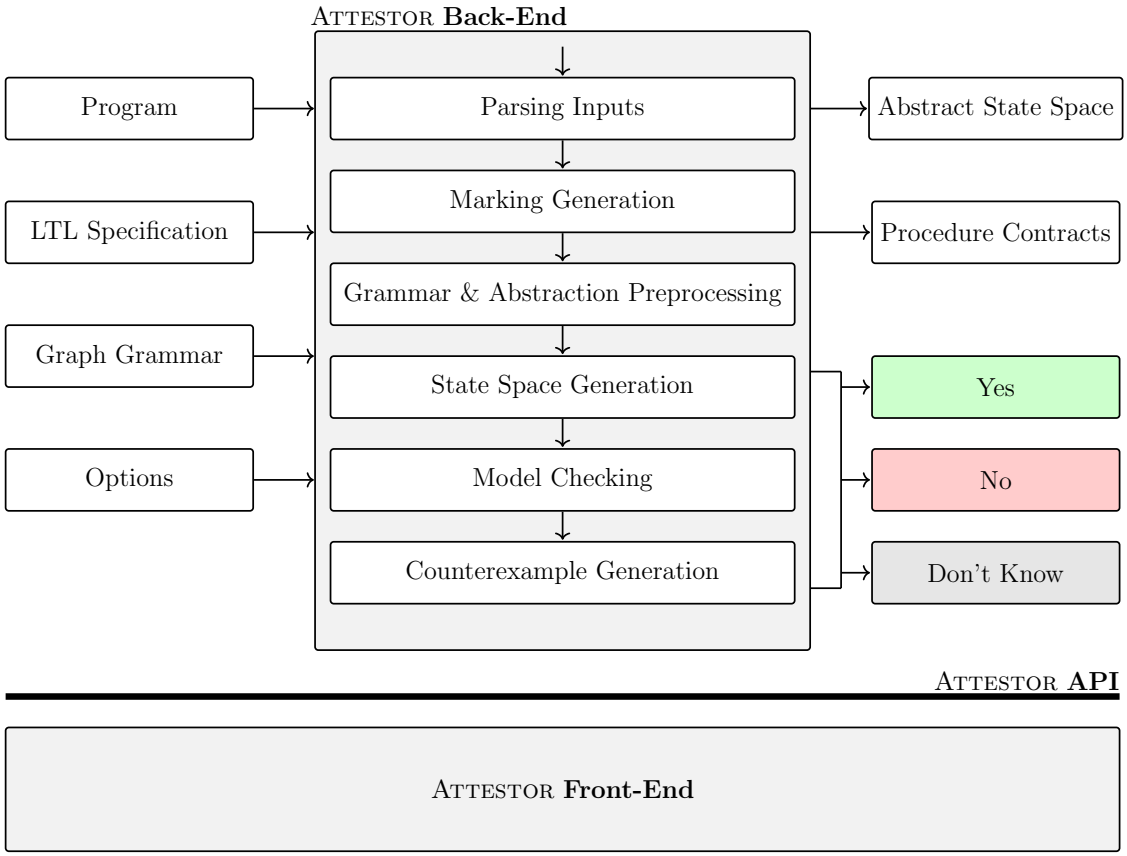


*Figure 1.1:* ATTESTOR architecture. [2]

program. Further options include the specification of initial heaps or properties to modify abstraction or garbage collection. ATTESTOR's core is the back-end which contains the program analysis. The front-end communicates via the ATTESTOR API with the back-end to visualize the output such as the generated state space.

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, 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 [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 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 1.2. 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. Therefore, the heap potentially needs to be concretised so that the statement is executed on a concrete heap. 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, heap abstraction is performed by applying grammar rules in a backward manner. Finally, the resulting state is labeled with atomic propositions that are satisfied by the heap. 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 whether a fixpoint has been reached and terminates the procedure in this case. 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** 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 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 procedural programs contain (recursive) methods and method calls with inherent state spaces that are not (directly) checked in the current implementation. Rather, procedure calls are woven into the main state space by considering their influence on the heap only after the execution. However, properties should also be checked for the procedure state spaces themselves as they might introduce

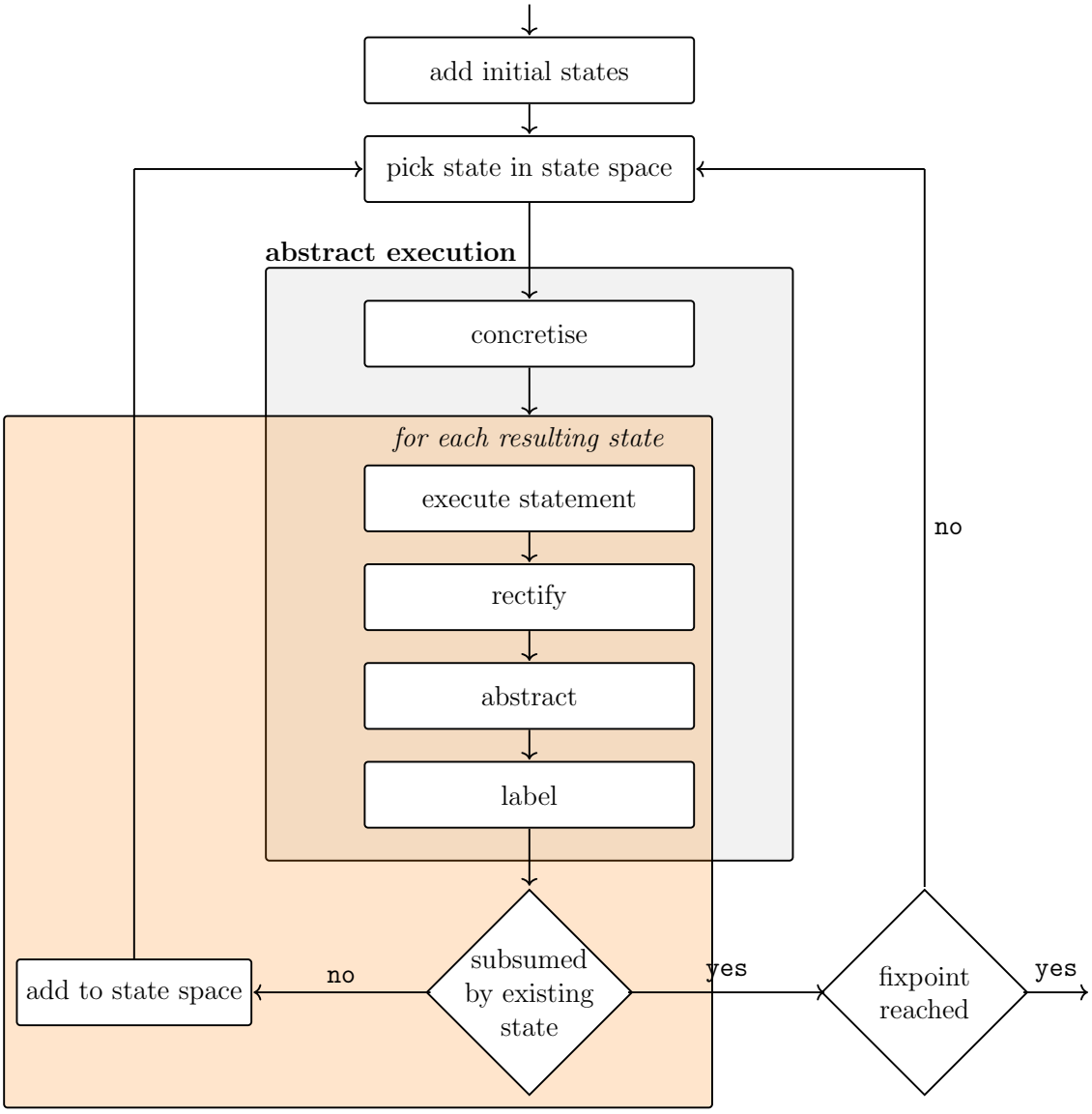


*Figure 1.2:* Phase 4: State space generation in Attestor [2].

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 program under consideration satisfies a specified property. 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.

Two parameters are crucial for model checking in order to obtain an expressive and valuable outcome: the model of the program under consideration and the formal description of the properties which are checked. The following sections introduce how we model input programs for model checking. Sections 2.1 and 2.2 describe how states and transitions of pointer-manipulating programs are represented by graphs and transitions between them, respectively. In order to describe hierarchical structures relevant to modeling programs with procedure calls, we focus on recursive state machines in Section 2.3. This chapter concludes by presenting linear temporal logic that formalizes properties that are checked for a program.

## 2.1 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. 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

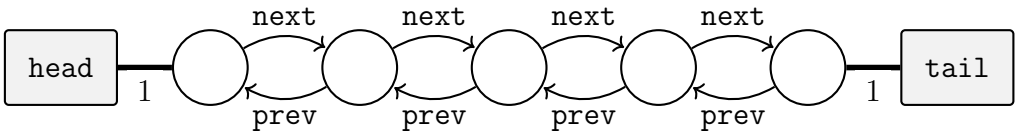


*Figure 2.1:* Heap for a doubly-linked list. Adapted from [7].

selector points from its source node to its target node.

Consider the class definition of a doubly-linked list (DLL) in JAVA code in Listing 2.1. The class defines a doubly-linked list to have the selectors `next` and `prev` to point to the successor and predecessor node, respectively.

```
1   public class DLList {
2
3     private DLList next;
4     private DLList prev;
5
6     public DLList(DLList list) {
7
8       this.next = list;
9       this.prev = null;
10      list.prev = this;
11    }
12  }
```

Listing 2.1: JAVA class definition for doubly-linked lists.

Figure 2.1 illustrates a heap for a doubly-linked list. The list consists of five elements represented by circles. The selectors `next` and `prev` are represented by directed edges of the graph. Furthermore, the program variables `head` and `tail` are attached to the first and the last vertex of the list, respectively.

Pointer-manipulating operations are represented by graph transformations. For instance, executing the operation `head := tail.prev` on the heap given in Figure 2.1 attaches the variable `head` to the vertex pointed to by the `prev` selector of the last vertex. The resulting heap is shown in Figure 2.2.

The number of objects in a heap can become unboundedly large. For instance, a program that contains a loop in which a new element is added to a list. If the loop is visited infinitely often, the resulting heap size will increase with every iteration. In order to obtain a finite representation of a heap, 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 2.3. The graph shows a doubly-linked list with a placeholder

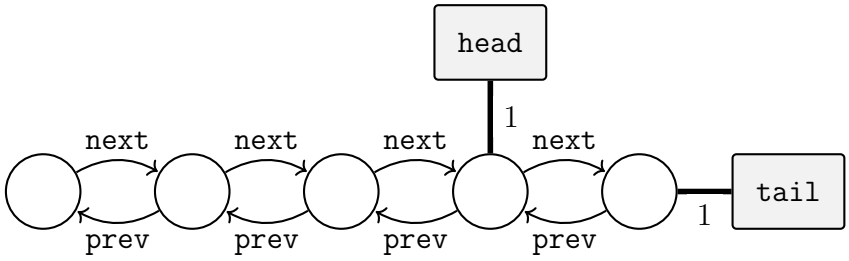


*Figure 2.2:* Modified heap after pointer-operation `head := tail.prev`.

labeled *DLL* which represents a doubly-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 2.3, the hyperedge labeled *DLL* is the 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 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 an abstract part such as the heap depicted in Figure 2.1.

**Definition 2.1: 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 $\Sigma$ 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

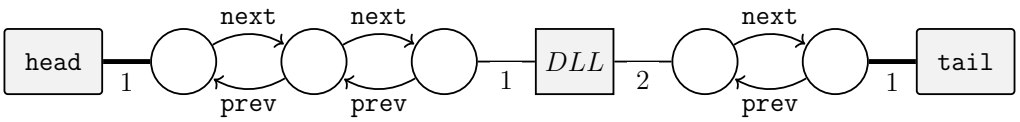


*Figure 2.3:* A doubly-linked list with abstracted subgraph represented as a hypergraph. Adapted from [7].

- $ext \in V^*$ 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 $\Sigma$ 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 $DLL$ in Figure 2.3. 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. 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 [5]**

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

The language of a hyperedge replacement grammar contains all concrete hypergraphs that are obtained by repeatedly applying the production rules to a given hypergraph. In this context, a nonterminal in a hypergraph is replaced by a hypergraph according to the production rules. The replacement requires that the external nodes of the replacing hypergraph are mapped to the placeholders in the hypergraph. To illustrate an hyperedge replacement, consider the hyperedge replacement grammar given in Figure 2.4. The grammar describes the language of all doubly-linked lists with at least two elements. 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. Let us consider the hypergraph from Figure 2.1. We have two options to apply the hyperedge replacement

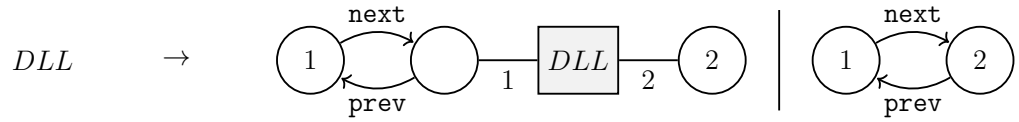


*Figure 2.4:* A hyperedge replacement grammar for doubly-linked lists. Adapted from [7].

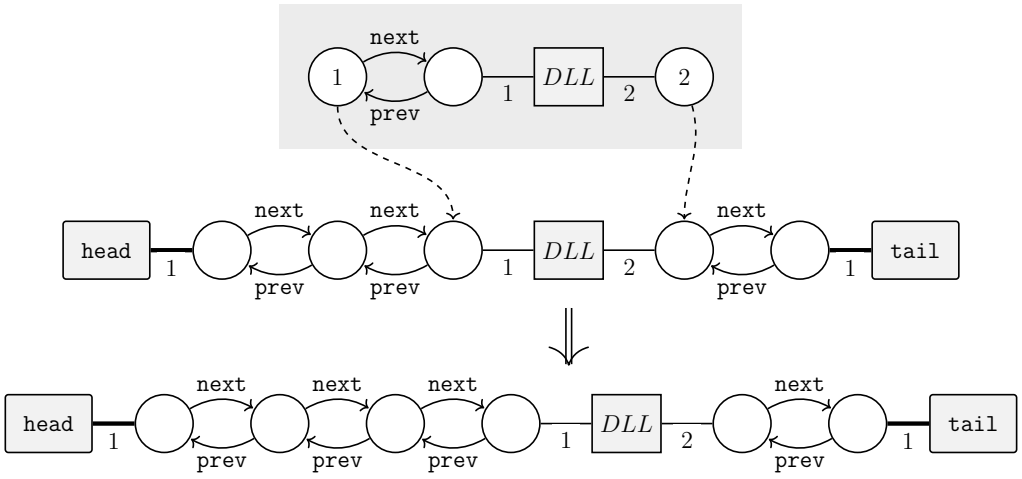


*Figure 2.5:* By applying the first production rule from Figure 2.4 a list element is added to the (abstract) hypergraph.

grammar for doubly-linked lists to the hypergraph under consideration. Applying the first rule yields the (abstract) hypergraph depicted in Figure 2.5, while applying the second rule yields a concrete hypergraph as shown in Figure 2.6.

Figures 2.5 and 2.6 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 doubly-linked lists in Figures 2.5 and 2.6 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 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.

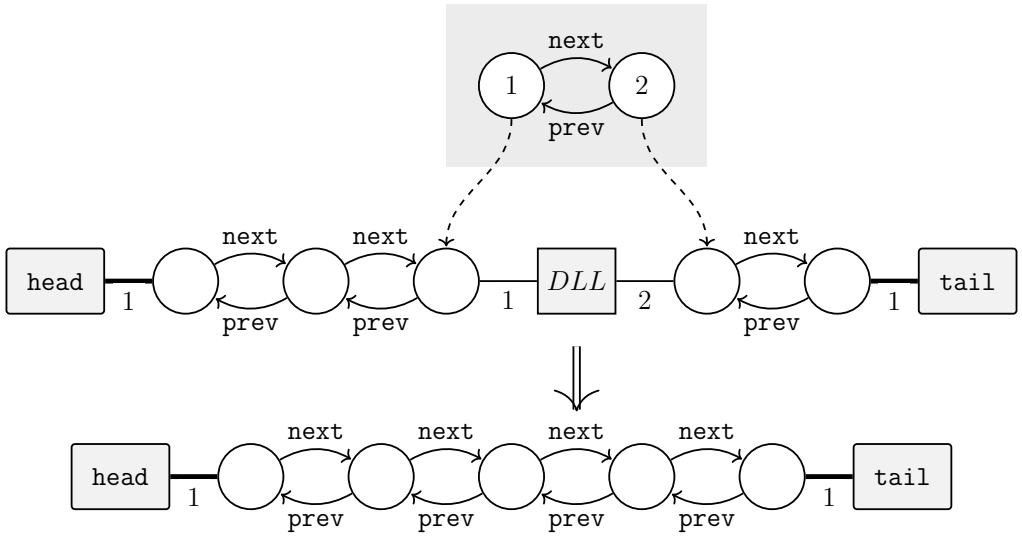


*Figure 2.6:* By applying the second production rule from Figure 2.4 a concrete hypergraph is obtained.

## 2.2 Transition Systems

Hypergraphs model the heap of a program at a certain moment. 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.

**Definition 2.3: Transition System [3]**

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

- $S$ is a set of states,
- $I \subseteq S$ is a set of initial states,
- $\rightarrow \subseteq S \times S$ 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 $S$, $Act$, 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,

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

The transition relation $\rightarrow$ formally describes how the transition system $T$ evolves starting in a state $s \in S$. Thus, the transition $(s, s') \in \rightarrow$ 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 $T = (S, I, \rightarrow, AP, L)$ is called *terminal* if and only if

$$\bigcup \{s' \in S | s \rightarrow s'\} = \emptyset.$$

Figure 2.7 depicts an example of a transition system $T = (S, I, \rightarrow, AP, L)$. 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 doubly-linked list with two elements. The variables `head` and `tail` are attached to the first and last node of the list, respectively. $s_0$ is succeeded by state $s_1$ which contains a list of three elements. $s_1$ has two successor states: itself and $s_2$, which contains a list of four elements. The transition relation $\rightarrow$ is represented by arrows connecting two states. For example, $(s_0, s_1) \in \rightarrow$ and $(s_1, s_2) \in \rightarrow$. The self-loop at state $s_1$ depicts the transition $(s_1, s_1) \in \rightarrow$. State $s_2$ is a terminal state as it does not have any outgoing transitions. The states in $T$ are labeled with atomic propositions that they satisfy. State $s_0$ satisfies proposition $a$. Therefore, $L(s_0) = \{a\}$. $s_1$ does not satisfy any propositions, while $s_2$ satisfies propositions $a$ and $b$. From the stated propositions, the set $AP$ of atomic propositions is implicitly given as $AP = \{a, b\}$.

A sequence of 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 a *path* of transition system $T$.

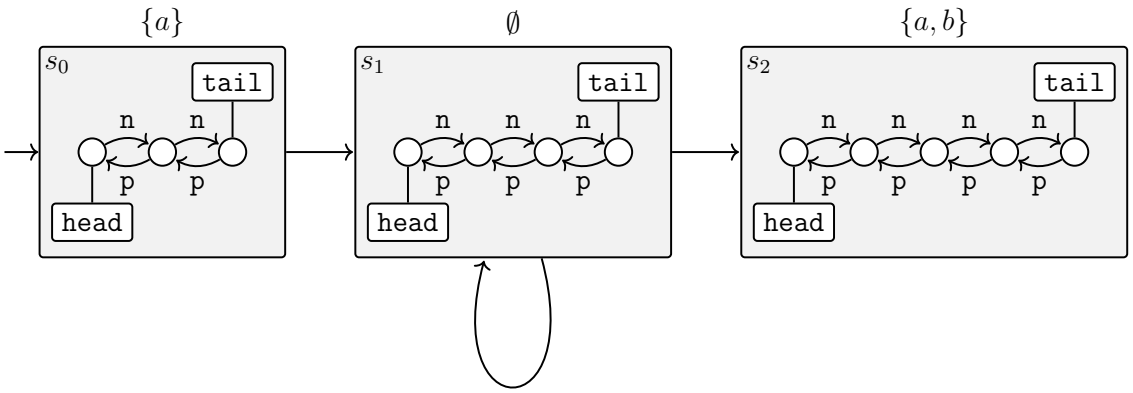


*Figure 2.7:* Sample state space.

**Definition 2.5: Paths [3]**

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

$$s_0 s_1 \dots s_n$$

such that

- $s_0 \in I$ is an initial state,
- $s_i \in \bigcup\{s \in S | s_{i-1} \to s\}$ 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

$$s_0 s_1 s_2 \ldots$$

such that

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

$Paths(T)$ 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 $\pi$.

**Definition 2.6: Trace [3]**

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

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

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

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

$Traces(s)$ 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 a transition system 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.

Consider the sample transition system $T$ in Figure 2.7. A path $\pi$ in $T$ is the finite sequence $s_0 s_1 s_2$. The corresponding trace of $\pi$ is given by

$$L(s_0)L(s_1)L(s_2) = \{a\}\emptyset\{a, b\}.$$

An infinite path $\pi'$ in $T$ is the sequence $s_0 s_1 s_1 s_1 \ldots$ that infinitely loops around the state $s_1$. $\pi'$ implies the trace

$$L(s_0)L(s_1)L(s_1)L(s_1)\cdots = \{a\}\emptyset\emptyset\emptyset\ldots.$$

## 2.3 Recursive State Machines

Often computer programs do not only consist of a sequence of commands, but also contain (recursive) calls to methods. The execution of procedural programs contains call- and return-statements to different sections of the input program. In order to capture the hierarchical (or recursive) structure, we introduce the notion of *recursive state machines*, as defined in [1], that encapsulate each method in its 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 2.7: Recursive State Machine [1]**

A *recursive state machine* (RSM) $\mathcal{A}$ over a finite alphabet $\Sigma$ is given by a tuple $(A_1, ..., A_k)$, 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* $\delta_i$, where transitions are 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 $j = Y_i(b)$,
- 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 $j = Y_i(b)$.

A sample RSM with three components $A_1, A_2, A_3$ is depicted in Figure 2.8. 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 $A_2$. 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 2.8: Global State [1]**

A *global state* of an RSM $\mathcal{A} = (A_1, ..., A_k)$ is a tuple $(b_1, ..., b_r, u)$, where $b_1 \in B_1, ..., b_r \in B_r$ 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 $A_{j_r}$.

Consider the sample RSM given in Figure 2.8. 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 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 $Y_i$ 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.

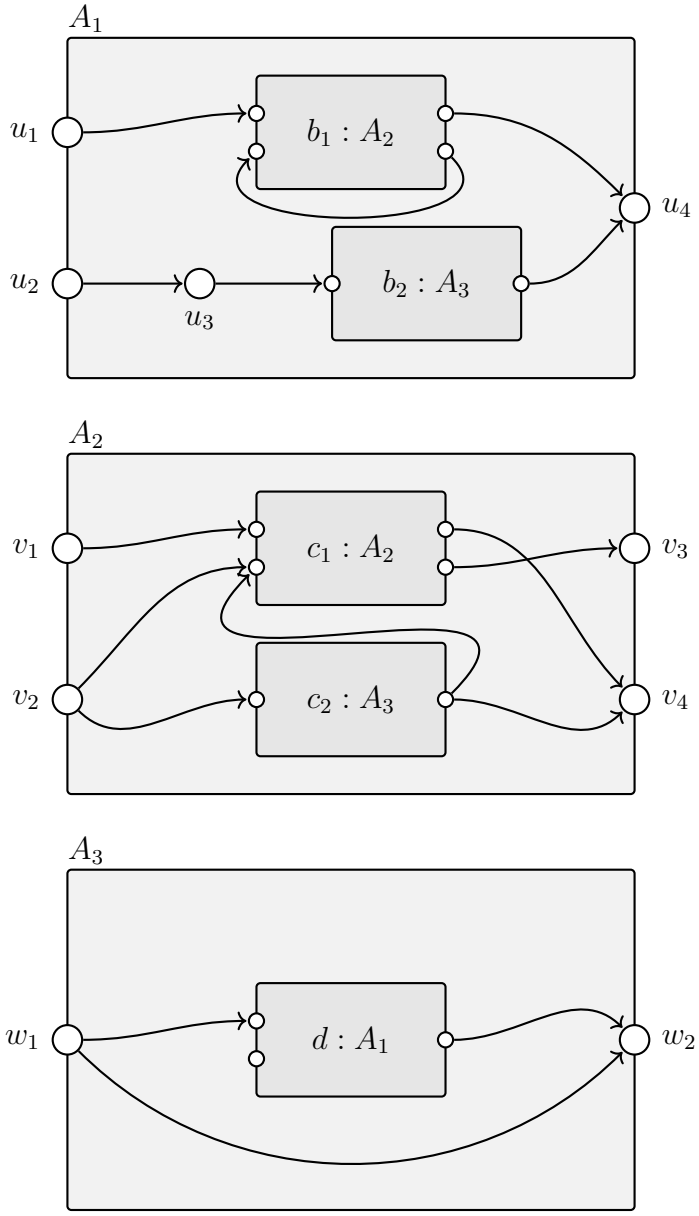


*Figure 2.8:* A sample recursive state machine. Adapted from [1].

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

**Definition 2.9: 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 $\mathcal{A} = (A_1, ..., A_k)$. 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 $s' = (b_1, ..., b_r, u')$.
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 2.9 specifies the possible kinds of transitions between global states $s, s' \in Q$ of an RSM $\mathcal{A}$. Consider the RSM given in Figure 2.8. 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 regarding 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]**

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

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

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. We model the program execution 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. Here, we focus on *linear temporal logic* described in the following section.

## 2.4 Linear Temporal Logic

First proposed by Pnueli in 1977, *Linear Temporal Logic* (LTL) is a 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 2.11: 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 $P$ meaning that 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 $T = (S, I, \rightarrow, AP, L)$ 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.

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 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:

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

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 are derived by the operators given in Definition 2.13 as follows:

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

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

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

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

$$\begin{array}{lll}
\sigma \models \texttt{true} & & \\
\sigma \models a & \text{iff} & a \in A_0, \text{ where } \sigma = A_0A_1A_2\ldots \\
\sigma \models \varphi_1 \wedge \varphi_2 & \text{iff} & \sigma \models \varphi_1 \text{ and } \sigma \models \varphi_2 \\
\sigma \models \neg\varphi & \text{iff} & \sigma \not\models \varphi \\
\sigma \models \bigcirc\varphi & \text{iff} & \sigma[1\ldots] = A_1A_2A_3\ldots \models \varphi \\
\sigma \models \varphi_1 \mathbf{U} \varphi_2 & \text{iff} & \exists j \geq 0.\sigma[j\ldots] \models \varphi_2 \text{ and } \sigma[i\ldots] \models \varphi_1, \forall 0 \leq i < j.
\end{array}$$

We constitute an intuitive understanding of temporal operators by visualizing their semantics in Figure 2.9.

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

Let $T = (S, I, \rightarrow, AP, L)$ 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

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

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

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

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

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 can specify the semantics of LTL for a transition system $T$.

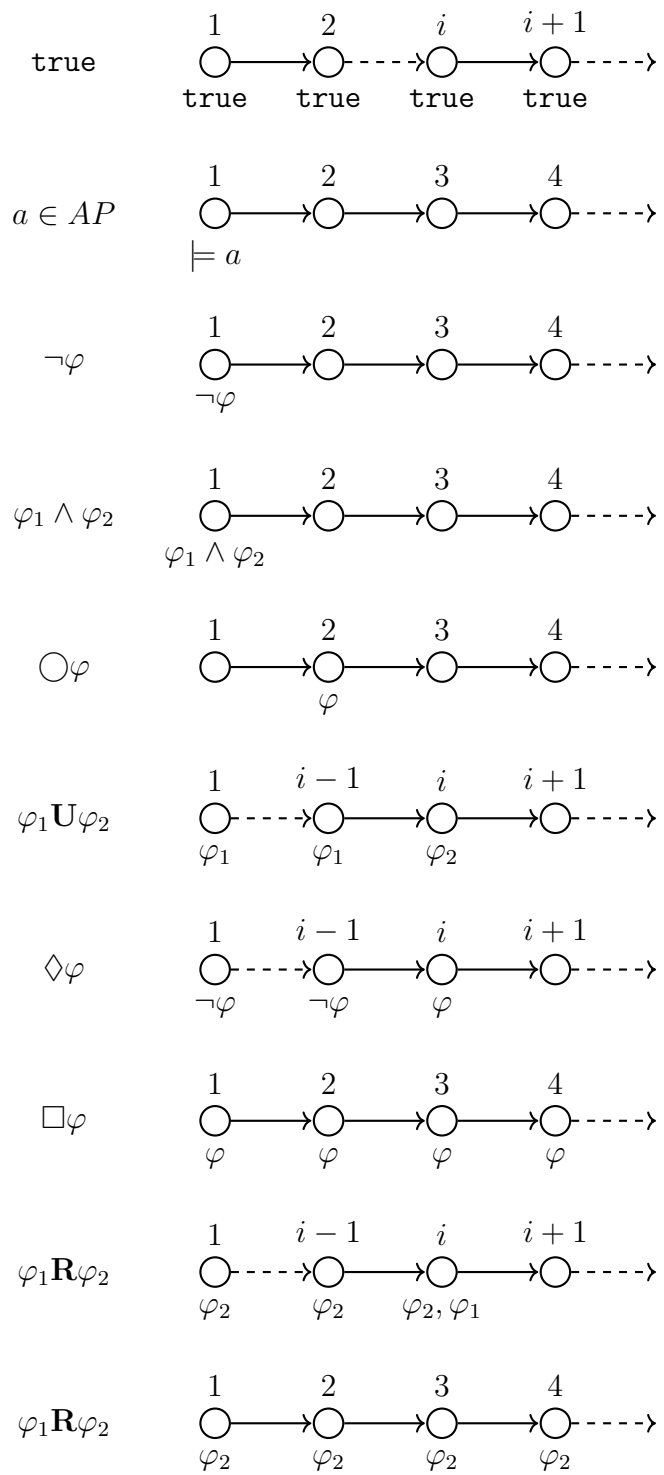


*Figure 2.9:* Intuitive semantics of temporal operators for a path $\pi$.

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

Given an LTL formula $\varphi$, 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 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* (PNF), 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

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

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

$$\square\lozenge\{\texttt{terminated}\} \rightarrow \square\lozenge\{\texttt{SLList}\}$$

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].

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. [4] that constructs a *proof structure* based on a set of *tableaux rules* in order to show whether a formula $\varphi$ is satisfied by a transition system $T$.

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

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

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

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

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

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

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

*Figure 3.1:* Tableaux rules for LTL model checking.

A proof structure is a directed graph $(V, E)$, where the set of vertices $V$ are composed of a set of *assertions* $\Lambda$ 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 $V \subseteq (\Lambda \cup \texttt{true})$ 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.

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

The underlying tableaux rules for the proof structure 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

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

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 $\Phi$ 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 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 $\lambda$ 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 the 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 $\varphi_1 \mathbf{R} \varphi_2$ $\varphi_1$, 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 $\mathbf{R}$. 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 $\lambda$ 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 $\lambda$ 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 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]**

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

$s \models \varphi$ iff $(V, E)$ 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 = reversedList;
9       revList = current;
10      current = next;
11    }
12
13    return reversedList;
14  }
```

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 the tableaux method to check whether the formula $\varphi = \Box\Diamond\{\texttt{SLList}\}$ is satisfied. $\varphi$ states that the heap will always be a singly-linked list. An equivalent formula to $\varphi$ in PNF is

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

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 the assertion into two assertions that are connected to the root assertion via directed edges. By exhaustively applying rules from the tableaux, the assertions are divided into sub-problems. The rule application continues until no more rules can be applied, i.e., the proof structure is found to be successful or not. A path in a proof structure either ends in a leaf or is infinite, i.e., contains a loop. In our example, both cases are represented. A leaf in the example proof structure is the Boolean value `true`. All paths ending in this leaf are successful. Moreover, the example at hand contains two infinite paths including assertions for state $s_{10}$: the infinite path $\pi_0$ looping around the sets of LTL formulae $\{\varphi\}, \{\neg\texttt{true}, \bigcirc\varphi\}$, and $\{\bigcirc\varphi\}$, and the infinite path $\pi_2$ looping around the sets $\{\texttt{true}\ \mathbf{U}\ \texttt{SLList}\}, \{\texttt{SLList}, \bigcirc(\texttt{true}\ \mathbf{U}\ \texttt{SLList})\}$, and $\bigcirc(\texttt{true}\ \mathbf{U}\ \texttt{SLList})$. 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, the $\pi_1$ 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 $\neg\texttt{true}\ \mathbf{R}\ (\texttt{true}\ \mathbf{U}\ \texttt{SLList} = \Box\Diamond\{\texttt{SLList}\}$.

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 $\bigcirc\texttt{SLList}$ 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 $\bigcirc\texttt{SLList}$ is satisfied. An on-the-fly approach thus circumvents the generation of the complete state space. We describe an implementation of the on-the-fly state space generation in the following chapter.

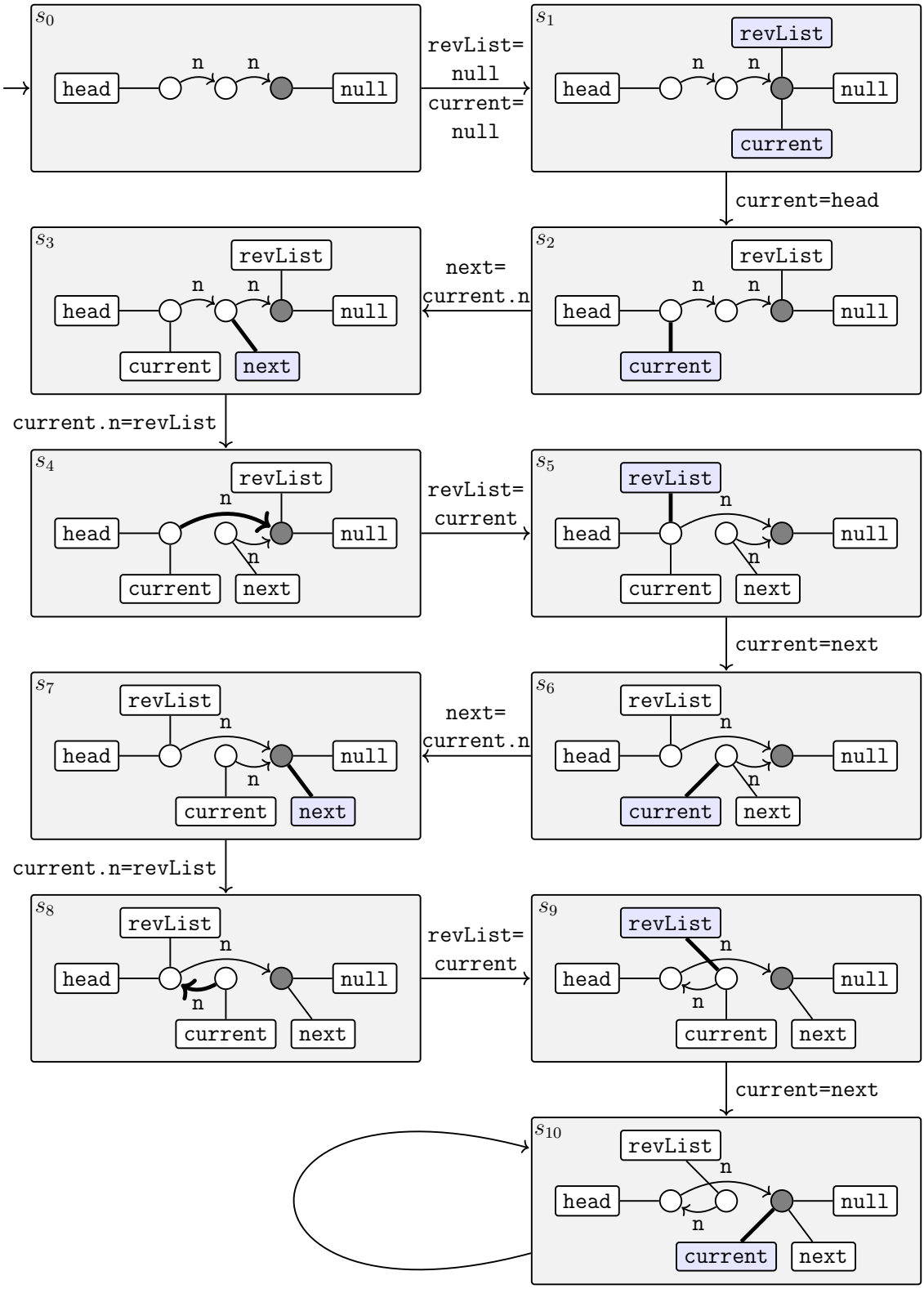


*Figure 3.2:* State space for reversing a singly-linked list. Adapted from [5].

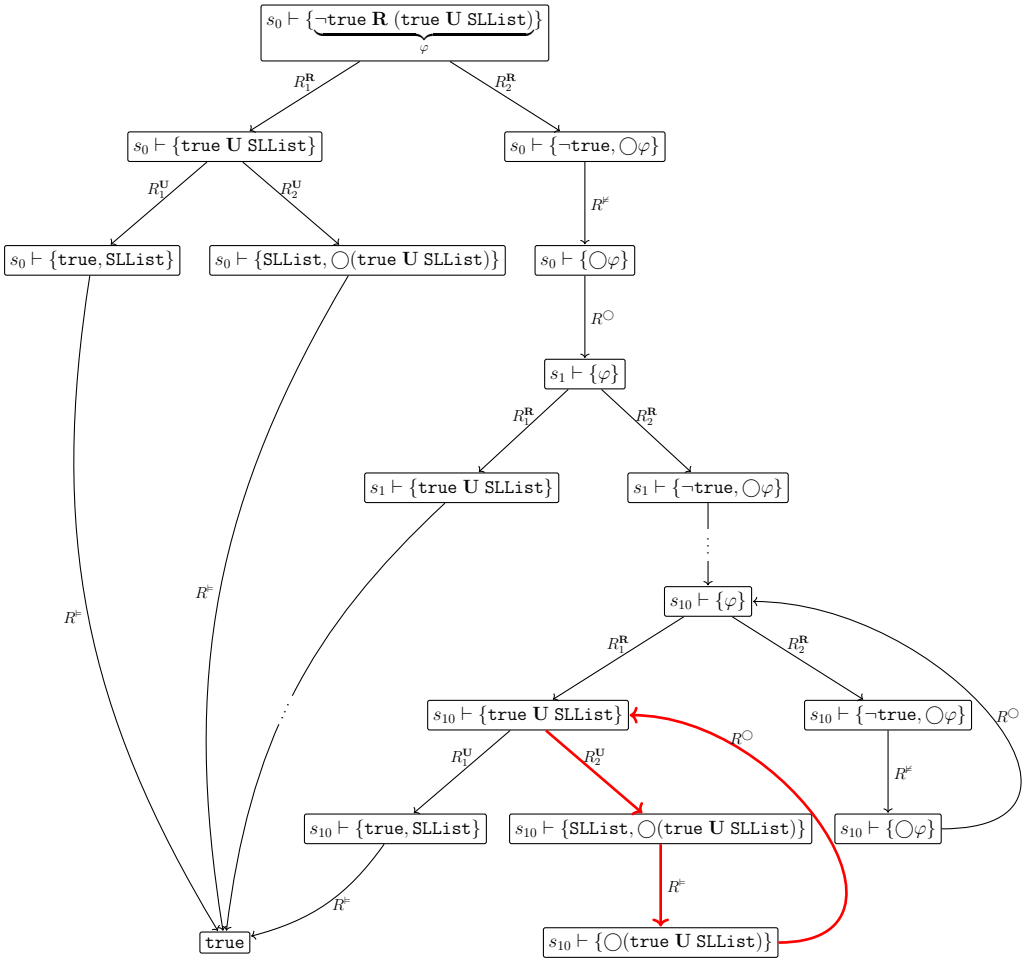


*Figure 3.3:* Proof structure for model checking method `reverse` for formula $\varphi = (\neg\texttt{true}\ \mathbf{R}\ (\texttt{true}\ \mathbf{U}\ \texttt{SLList}))$.

# Chapter 4

# On-The-Fly Model Checking

The tableaux construction is an on-the-fly approach to model checking transition systems for an LTL formula $\varphi$. In this chapter, we describe how we adapt the tableaux construction by Grumberg et al. describe in Section 3.1 to account for model checking of hierarchical structures including procedure state spaces. In a next step, we present the implementation of the algorithm within the ATTESTOR framework and conclusively evaluate our proceedings.

## 4.1 Algorithm

Given a transition system (or state space) $T$ and a set $\Phi$ of LTL formulae, the goal of the hierarchical tableaux construction is to verify whether the state space, including procedure state spaces induced by method executions, satisfies the temporal properties specified by $\Phi$. We assume the state space to be flat and finite. Currently, ATTESTOR implements the tableaux construction by Grumberg et al. for model checking the generated state space of the main method of an input program. We extend the current approach by the following functionalities:

- Interweave state space generation and 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 4.2) that validates a set $\Phi$ of formulae for an input program and simultaneously generates the state space under consideration on-the-fly. The state space generation is based on 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. For $\bigcirc$ formulae the tableaux rule ($R^{\bigcirc}$) specifies a state switch. Thus, successor

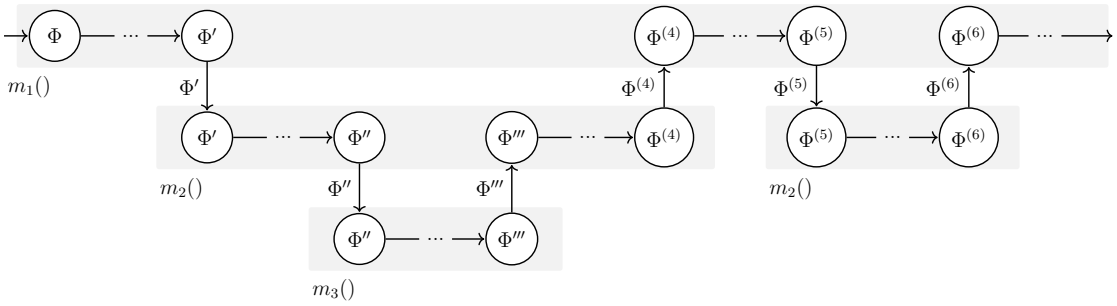


*Figure 4.1:* Conceptual flow of model checking information in a state space with method invocations.

states need to be generated. State generation involves the statement execution of the current state. At this point, we extend the state space generation in ATTESTOR by model checking the generated state. 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 the corresponding procedure state space. That is, the procedure `generateAndCheck` is executed on the calling state of the method and the current set $\Phi'$ of formulae. The set $\Phi'$ is a set of formulae resulting from successively applying tableaux rules to the formulae in the set $\Phi$. 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. Figure 4.1 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 model checked during generation. The state labels $\Phi^{(k)}$ denote the set of LTL formulae to be validated.

Model checking results include information on whether a 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 for which the set $\Phi$ of formulae is to be validated. The tuple $(HC_{in}, \Phi)$ is mapped to a tuple of model checking results containing the resulting set $\Phi'$ 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 $\pi$ 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 before and after the

execution of a procedure, respectively. Details on procedure contracts are described in [8]. 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 $\Phi'$ 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 $\Phi$ is not satisfied for the complete program state space. Thus, an early termination of the model checking procedure is possible that does not require building the complete state space. The concept of the hierarchical tableaux construction is depicted in Figure 4.2.

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 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 building the proof structure. Second, we require a possibility to query for a list of successor states for a given state $s$. Figure 4.3 sketches the architecture our implementation.

Let us turn to the on-the-fly tableaux construction first. The `ProofStructure` class implements the tableaux construction. By calling the method `build` for 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. The class `OnTheFlyProofStructure` modifies the current implementation such that it is capable of constructing the proof structure for an on-the-fly constructed state space.

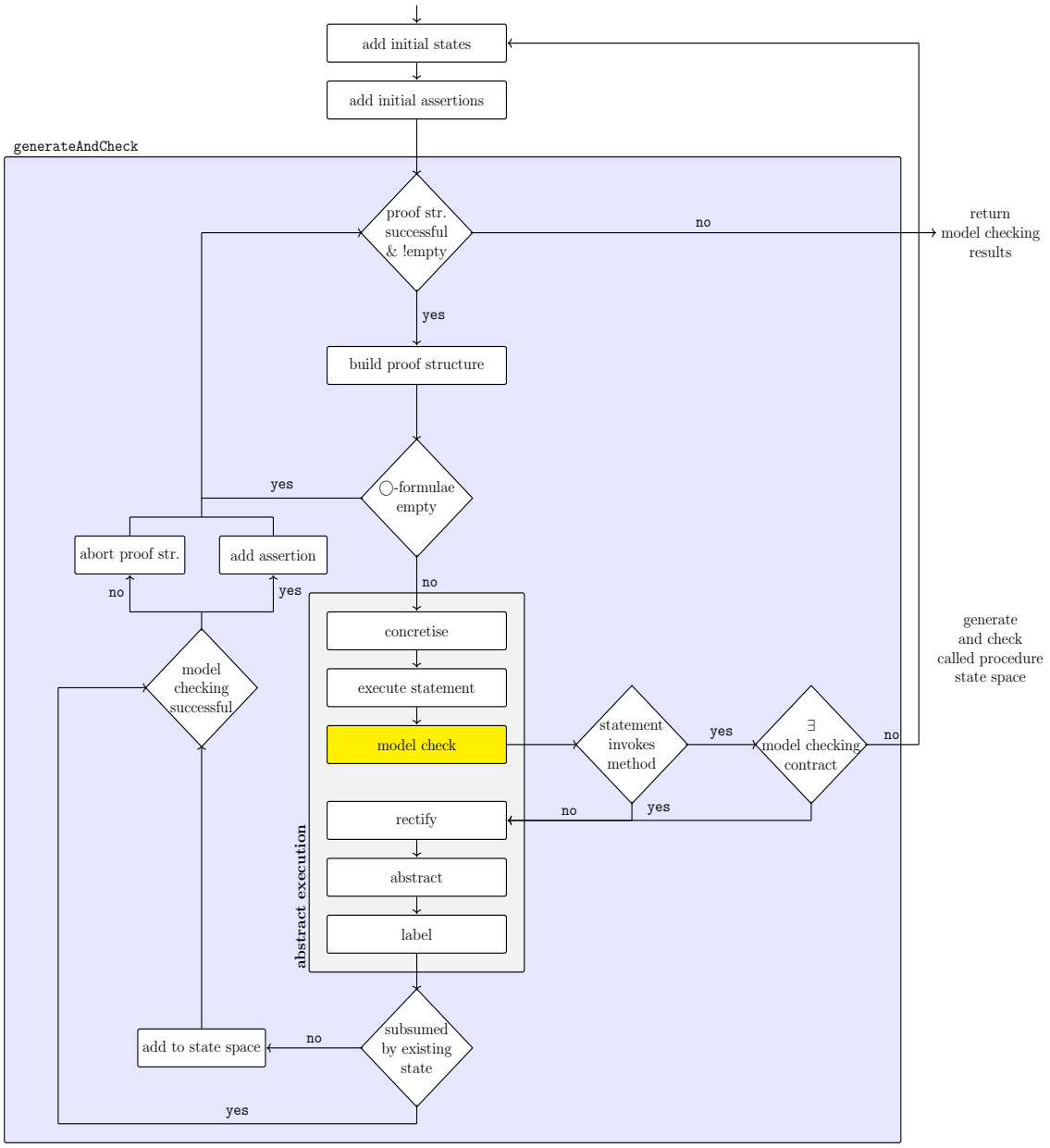


*Figure 4.2:* On-the-fly hierarchical model checking by a hierarchical tableaux construction.

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`. The list of successors is then communicated to the proof structure by adding new assertions and reentering the model checking loop (cf. Figure 4.2).

Further adjustments are required in the labeling of procedure states. An atomic proposition $a \in AP$ is added 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 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, 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 current scope. 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 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 is restored. Based on the resulting state, the corresponding atomic propositions are determined.

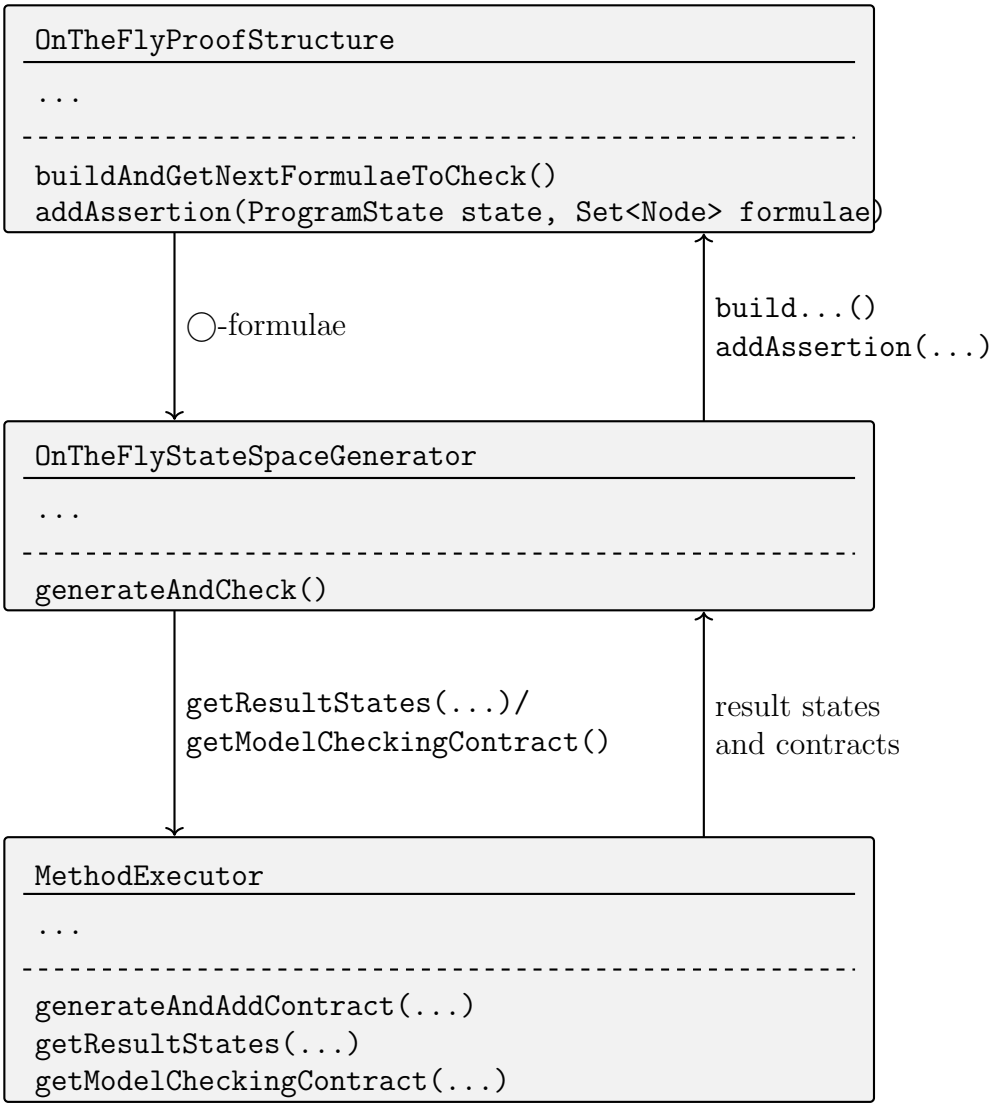


*Figure 4.3:* UML for on-the-fly model-checking.

## 4.3 Evaluation

When model checking procedural programs with possibly recursive method invocations, 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 possible erroneous behavior within method executions. The on-the-fly approach on hierarchical model checking presented in this chapter 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 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 procedure state spaces 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

# Hierarchical Model Checking with Recursive State Machines

Recursive state machines model the control flow of sequential imperative programs containing possibly recursive method calls [1]. In the concept of RSMs two kinds of states are differentiated: ordinary 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. Thus, we modify the previously presented on-the-fly model checking approach for procedure state spaces by model checking recursive state machines. However, instead of applying the automata-based approach by Vardi and Wolper presented in Chapter **??**, we propose an algorithm that constructs a tableaux for a component $A_i$ of an RSM $\mathcal{A}$ that stores the pre-generated procedure state space of the method $i$. By this means, model checking of procedure state spaces draws on stored procedure state spaces such that state spaces are not repeatedly generated. 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 this approach in terms of time and space efficiency.

## 5.1 Algorithm

Given a transition system $T$ and a set $\Phi$ of LTL formulae, the goal of hierarchical model checking is to verify whether $T \models \Phi$. The RSM-based model checking algorithm consists of two main steps: First, a recursive state machine is constructed from the input program. Second, model checking is executed on the RSM.

For the first step, we assume the (procedure) state spaces to be given. The construction of the RSM $\mathcal{A}$ starts with generating a component state machine $A_i$ for every procedure state space. Each component $A_i$ stores a set of procedure state spaces and a set of boxes that indicate which method is invoked by which state within the

component $A_i$.

The second step executes LTL model checking for the RSM $\mathcal{A}$ and a set $\Phi$ of formulae. Model checking the RSM $\mathcal{A}$ boils down to model checking the components $A_i$ in $\mathcal{A}$. 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 found 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 described in the on-the-fly approach (cf. Figure 4.1). The model checking results from constructing the proof structure for a component $A_j$ are returned to the calling state, and included in the proof structure of the calling component. 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. A sketch of the algorithm is depicted in Figure 5.1.

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 the model checking results in form of contracts. Thus, if a previously solved model checking instance appears, the proof structure does not need to be repeatedly computed, but the stored results can be applied.

Early termination of the algorithm can be achieved if a violating state is found or the set of formulae is determined to be true for all upcoming states. Thus, the proof structure does not need to be built entirely.

## 5.2 Implementation

An implementation of the RSM model checking using the tableaux construction can be found at `https://github.com/SallyChau/master_thesis/tree/master/attestor`. We implemented the model checking algorithm within the ATTESTOR framework. The RSM-based model checking approach is an alternative to the 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

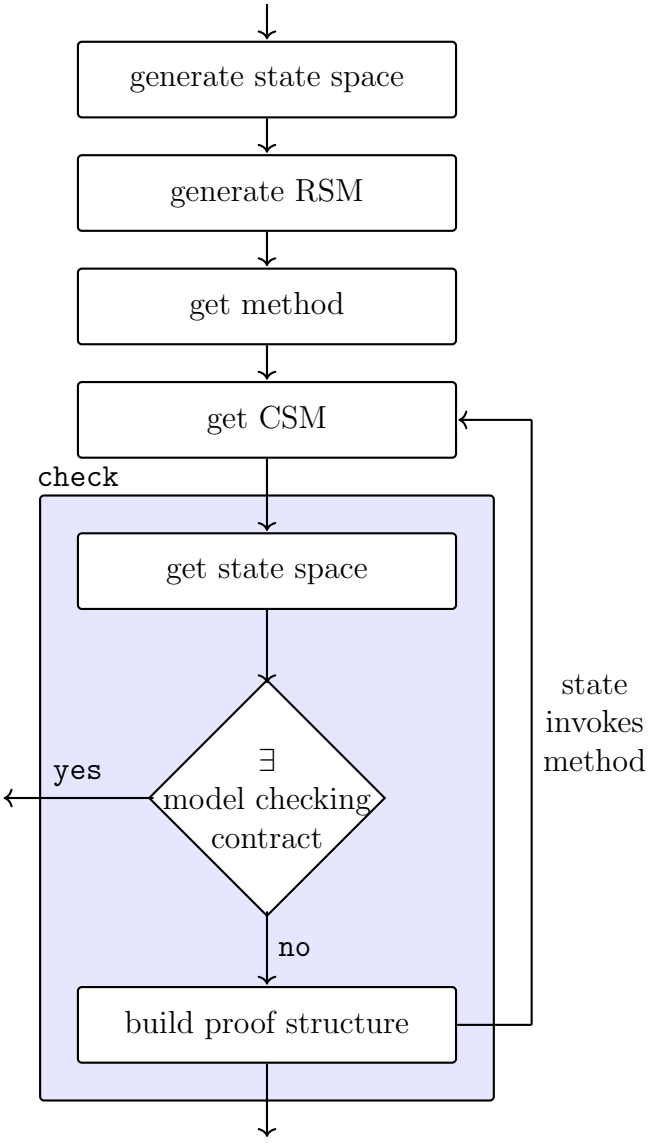


*Figure 5.1:* Model checking RSMs using the tableaux construction.

- Enrich RSM and CSM with proof structures to account for model checking procedure state spaces.

Figure 5.2 sketches the implementation of the required adaptations.

The first adaptation is realized by storing the generated procedure state spaces during ATTESTOR's state space generation phase including information on which state invoked the method call.

Now, given a set of procedure state spaces, we construct an RSM for the input program. The class `RecursiveStateMachine` builds the corresponding RSM by creating an object of the class `ComponentStateMachine` for each method that is called. Since distinct calls to a method originating from two distinct states can result in distinct state spaces, a `ComponentStateMachine` internally maps the calling state to the resulting state space. Furthermore, `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 state $s$. Moreover, a `ComponentStateMachine` stores the model checking results for executed tableaux constructions.

In order to account for model checking procedure state spaces, we require a `ComponentStateMachine` to trigger the tableaux construction for the state space under consideration. Therefore, we implement the class `HierarchicalProofStructure` which adapts ATTESTOR's proof structure to work on `ComponentStateMachine`. Thus, if the proof structure construction finds a state that invokes a method execution, the model checking of the called `ComponentStateMachine` (procedure state space) is started. 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, that is the proof structure is unsuccessful or the proof structure is ultimately successful (there are no more formulae to check), the result can be reported to the dependent proof structures, such that the model checking process can be terminated. Together with a possible failure trace, the model checking result is returned.

In order to avoid repetitive computation of proof structures, we store model checking results as contracts within a `ComponentStateMachine`, similar to the implementation in Chapter 4. However, since the context of the method is unambiguously determined be the `ComponentStateMachine`, we do not require the context of procedure contracts as it is the case for on-the-fly model checking.

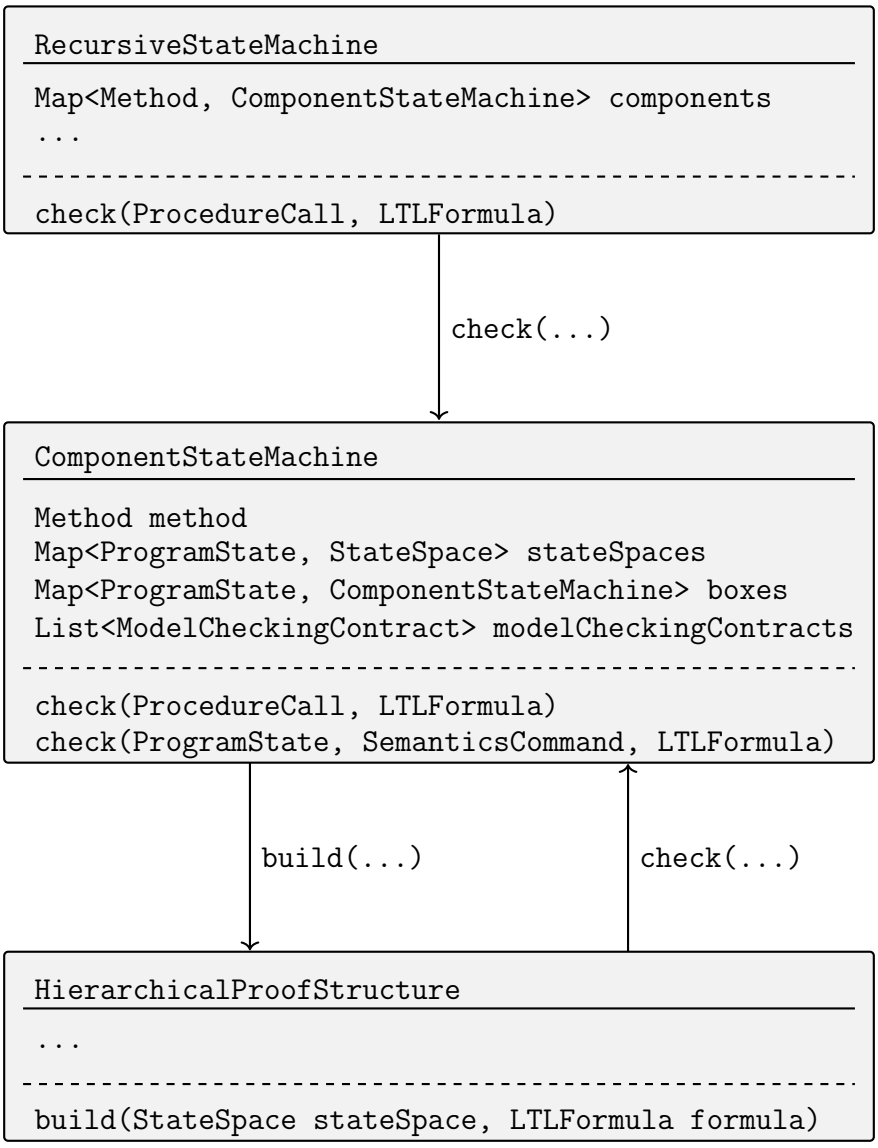


*Figure 5.2:* UML for hierarchical model checking with RSMs.

## 5.3 Evaluation

In this chapter, we presented an algorithm for model checking procedure state spaces represented as an RSM. 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 model checking results can be easily communicated to the state that invoked the method execution.

As procedure state spaces are stored within the component state machine of a method, procedure state spaces only need to be computed once during the state space generation phase. Similarly, model checking of a formula $\varphi$ can be skipped if the results are known due to previous model checking instances.

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

# 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 Comparison regarding different topics: time, memory, ...

# 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 *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] Heinen, Jonathan: *Verifying Java Programs - A Graph Grammar Approach.* PhD thesis, RWTH Aachen University, 2015.

[6] 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.

[7] Heinen, Jonathan, Christina Jansen, Joost-Pieter Katoen and Thomas Noll: *Verifying pointer programs using graph grammars.* Sci. Comput. Program., 97:157–162, 2015.

[8] 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.