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 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.

Bonn, den 28. September 2015, Sally Chau

Abstract

Contents

Chapter 1

Introduction

1.1 Attestor

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 will be. In this section, we will 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 model pointer-manipulating programs, we will introduce the concept of *recursive state machines* that capture the hierarchical (or recursive) nature of method calls in programs.

Since our goal is to model check pointer-manipulating programs, the states of the transition system modeling the input program consist of heap configurations. In order to represent possibly unbounded heap structures, we depict how the heap can represented by *graph grammars* in the second part of this section.

2.1.1 Transition Systems

Transition systems represent the behavior of a system as a model. 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 of a system encodes the information about the system at a certain moment. These pieces of information are formulated as a set of *atomic propositions*. A transition within a system thus depicts that the state of the system changes. 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 2.1 (Transition System [2])**.** *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 propositions $a \in AP$ are 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 2.2 (Terminal State [2])**.** *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 prolongs, is called an *execution* of the transition system T.

Definition 2.3 (Execution)**.** *Let $T = (S, Act, \rightarrow, I, AP, L)$. A* finite execution *of T is an alternating sequence*

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

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*

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

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 $0 \leq i$.*

A state s is called *reachable* if there is an execution that ends in s.

Definition 2.4 (Reachable States [2])**.** *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*

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

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

For our purpose of model checking pointer-manipulating programs, we will only consider the states and the according atomic propositions of the transition system under consideration. Thus, we will focus on the states of a transition system by omitting the actions. Multiple transitions between two states with different actions are thus summarized into a single transition. Therefore, the notion of an execution of a transition will shift towards the notion of *paths* of a transition system that denote sequences of states that are visited throughout a run.

Definition 2.5 (Paths [2])**.** *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*

$$s_0 s_1 s_2 \ldots$$

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 $i > 0$.*

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 state that are traversed throughout a transition system. However, the related sets of atomic propositions of the traversed states, which are relevant for model checking, are not observable. Therefore, we consider the notion of *traces* which are sequences of sets of atomic propositions that are satisfied along a path π.

Definition 2.6 (Trace [2])**.** *Let $T = (S, Act, \rightarrow, I, AP, L)$ be a transition system. 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*

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

Let $Traces(s)$ denote the set of traces of paths starting in state s and let $Traces(T)$ denote the set of traces of the initial states of a transition system T.
The trace $trace(\pi)$ of a path π can be regarded as a word over the alphabet 2^{AP}. Thus, the trace of an infinite path can be interpreted as an infinite word over 2^{AP}. When analyzing a transition system T, requirements are defined for the traces of T. These requirements can be expressed by *linear-time properties.* A linear-time property is a set of (infinite) words over a set AP of atomic propositions, and thus defines a language to be satisfied by the traces of T.

Definition 2.7 (Linear-Time Property [2])**.** *A* linear-time property *over the set of atomic propositions AP is a subset of $(2^{AP})^\omega$.*

The relation between a transition system T and a linear-time property P is captured by the following satisfaction relation $\models$.

Definition 2.8 (Satisfaction Relation for Linear-Time Properties [2])**.** *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 $Traces(s) \subseteq P$.*

From this definition it follows that a transition system T satisfies a linear-time property P if all its traces satisfy P. Thus, for $T \models P$, every trace of T has to be a word in the language induced by $P \subseteq (2^{AP})^\omega$.

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 even recursive) structure of the system, we introduce the notion of *recursive state machines*, that encapsulate each method in an own *component*. A recursive state machine is a transition system that ....

Definition 2.9 (Recursive State Machine [1])**.** *A* recursive state machine *(RSM) 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 relatio*n* 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* $j = Y_i(b)$*.*

Semantics

In order to define the execution of an RSM $A = (A_1, ..., A_k)$, this section describes the global relation between its component state machines $A_i, 1 \leq i \leq k$. A *global state* of an RSM consists of boxes and nodes of its CSMs.

Definition 2.10 ((Global) State [1])**.** *A* (global) state *of an RSM* $A = (A_1, ..., A_k)$ *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* 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* $Y_{j_r}(b_r) = j$*.*

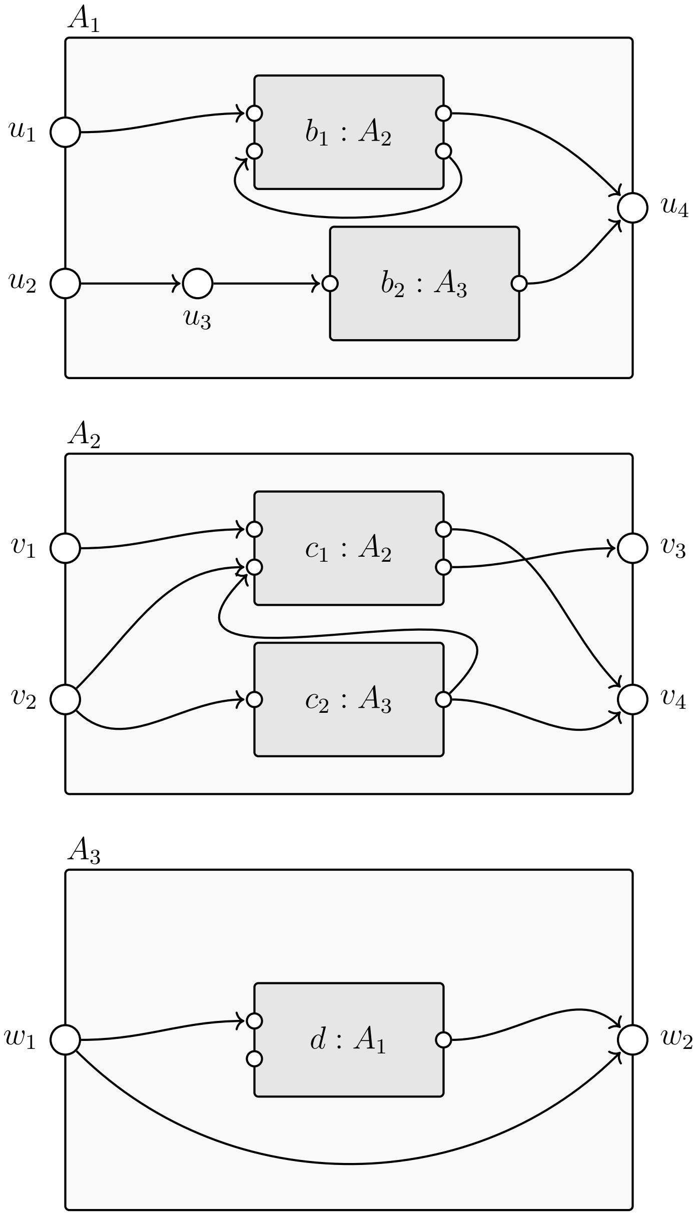


Figure 2.1: A sample recursive state machine. Adopted from [1]. Describe more....

A well-formed state $(b_1, ..., b_r, u)$ of an RSM $A = (A_1, ..., A_k)$ corresponds to a path through the components A_j of A, where we enter component A_j via box b_r of component A_{j_r}.

In order to transition between global states of an RSM 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 2.11 ((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 $A = (A_1, ..., A_k)$. A* (global) transition relation *δ for 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 2.11 defines the possible kinds of transitions between global states $s, s' \in Q$ of an RSM A. Case 1 describes the scenario where the source and the destination states are both within the same component A_j, while case 2 depicts that a new component is entered via a box b' of A_j. Thus, the current node of the destination state s' is the entry-node e. 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.
After defining the terms of global states and the global transition relation for an RSM A, we can summarize these components together with the finite alphabet Σ within the concept of a *labeled transition system* T_A, which encodes the execution of A.

Definition 2.12 (Labeled Transition System [1])**.** *For an RSM $A = (A_1, ..., A_k)$, the* labeled transition system *(LTS) $T_A = (Q, \Sigma, \delta)$ consists of*

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

The LTS of an RSM A is also called the unfolding *of A.*

2.1.3 Heap Representation

After describing recursive state machines as the model for our model checking procedure, we now focus on the representation of the *states* in the transition system. Since we will analyze pointer-manipulating programs, the states under consideration are *heap configurations* holding information on heap objects, program variables, and selectors. Heaps configurations are represented as graphs as described in [4]. The vertices of the graph represent heap objects, while the 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. The representation of heaps as graphs represents pointer-manipulating operations such as `head := tail.prev` as graph transformations.

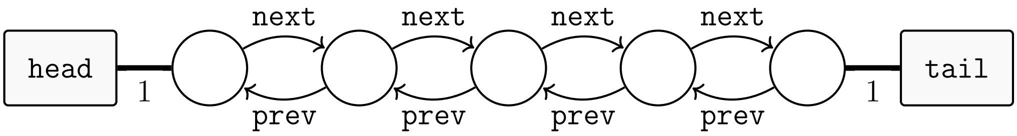


Figure 2.2: A heap as a graph. The heap is a doubly-linked list. Adopted from [4].

Heap configurations change over the course of the program execution so that the size of the heap can become unboundedly large, e.g. when new objects are added to the heap. Since the above mentioned graph representation would result in an unbounded size of the graph, 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 vertices. The number of vertices a hyperedge connects is captured in its *rank*.

Definition 2.13 (Hypergraph [3]). *Given a finite ranked alphabet Σ with associated ranking function $rk : \Sigma \to \mathbb{N}$. A (labelled) 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-labelling 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 $rk(e) = rk(lab(e))$.

An example for a hypergraph is depicted in Figure2.3.

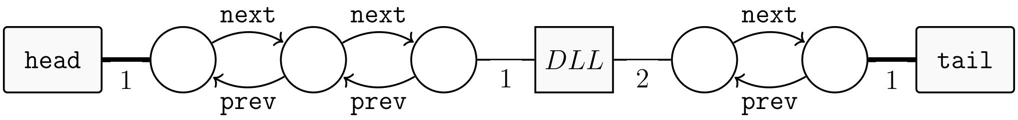


Figure 2.3: A doubly-linked list represented by a hypergraph. Adopted from [4].

2.2 Linear Temporal Logic

First proposed by Pnueli in 1977, *Linear Temporal Logic* (LTL) is a modal temporal logic used to describe properties of paths in a transition system. LTL formulae are composed of three components: the boolean operators *negation* ($\neg$) and *conjunction* ($\wedge$), the temporal operators *next* ($\bigcirc$) and *until* ($\mathbf{U}$), and a set of atomic propositions AP. Atomic propositions are state labels of a transition system, which express properties that hold for a single state, e.g., "$i = 1$". Formally, the set of LTL formulae is defined as follows:

Definition 2.14 (Syntax of LTL [2])**.** *Given a set AP of atomic propositions with $a \in AP$,* LTL formulae *are recursively defined by*

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

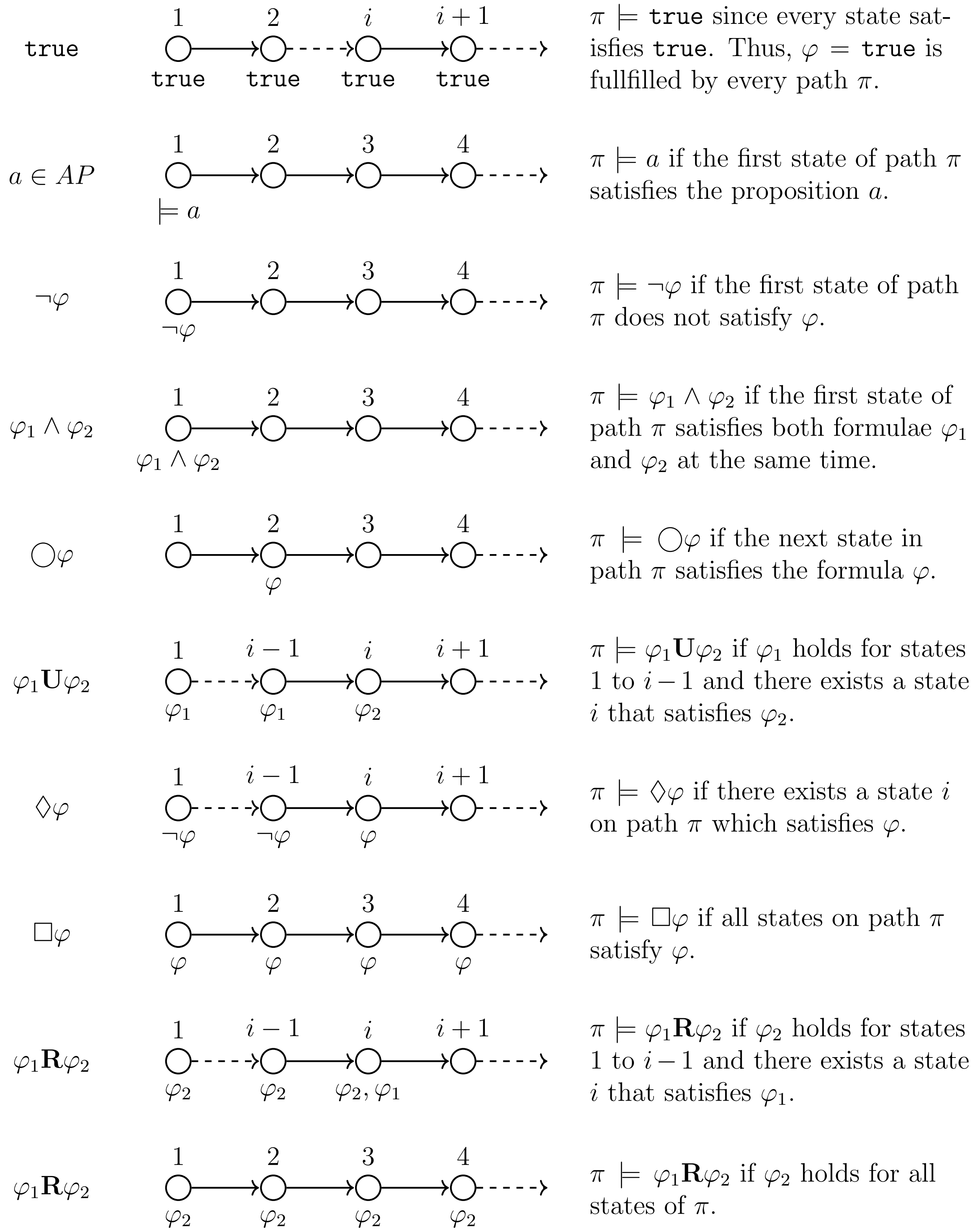
Further temporal operators that are commonly used, but are not included in the definition of LTL formulae, are the temporal modalities *eventually* ($\Diamond$), *globally* ($\Box$), and *release* ($\mathbf{R}$). They can be derived using the operators given in Definition 2.14 as follows:

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

In order to get an intuitive understanding of the semantics of temporal operators, and therefore LTL formulae, we visualize the semantics of temporal operators in Figure 2.4.

Formally, the semantics of LTL is given by the model relation $\models$ that is based on the satisfaction relation over paths and states of a transition system.

Definition 2.15 (Semantics of LTL [2])**.** *Given an LTL formula φ, a concrete transition system S, and a path $\pi \in Paths_S$, the model relation $\models$ for LTL formulae*



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

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

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

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

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

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

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

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

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

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

Figure 2.4: Intuitive semantics of temporal operators.

is defined by

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

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

Definition 2.16 (Positive Normal Form [2])**.** *Given a set AP of atomic propositions with $a \in AP$, LTL formulae in* positive normal form *(PNF) are defined by*

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

2.3 LTL Model Checking Algorithms

2.3.1 Automata-Based Model Checking

2.3.2 Tableaux Construction

Chapter 3

Hierarchical Model Checking with Recursive State Machines

3.1 Algorithm

3.2 Implementation

3.3 Evaluation

Chapter 4

On-The-Fly Hierarchical Model Checking

4.1 Algorithm

4.2 Implementation

4.3 Evaluation

Chapter 5

Benchmarks

5.1 Experimental Setup

Describe Technical details here

5.2 Instances

Describe code examples and properties here

5.3 Result

Table of values

Chapter 6

Conclusion

6.1 Discussion

6.2 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] Baier, Christel and Joost-Pieter Katoen: *Principles of model checking.* MIT press, 2008.

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

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