RWTH Aachen

Department of Computer Science
Technical Report

Temporal Assertions for Sequential and Concurrent Programs

Volker Stolz

ISSN 0935–3232 · Aachener Informatik Berichte · AIB-2007-15

RWTH Aachen · Department of Computer Science · August 2007

The publications of the Department of Computer Science of RWTH Aachen (*Aachen University of Technology*) are in general accessible through the World Wide Web.

`http://aib.informatik.rwth-aachen.de/`

Temporal Assertions for Sequential and Concurrent Programs

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der Rheinisch-Westfälischen Technischen Hochschule Aachen zur Erlangung des akademischen Grades eines Doktors der Naturwissenschaften genehmigte Dissertation

vorgelegt von

Diplom-Informatiker
Volker Stolz
aus
Würselen

Berichter: Prof. em. Dr. Klaus Indermark
Prof. Bernd Finkbeiner, Ph. D.

Tag der mündlichen Prüfung: 20.7.2006

Diese Dissertation ist auf den Internetseiten der Hochschulbibliothek online verfügbar.

Temporal Assertions for Sequential and Concurrent Programs

Dipl.-Inform. Volker Stolz

Abstract

In this thesis, we present an extension to the well-known concept of assertions: *temporal assertions allow the specification and validation of modal safety properties of an application at runtime.* We see this as a necessary step in enforcing the growing number of implicit requirements of software specifications, which are often only informally defined in the documentation of application program interfaces (API) and are beyond the reach of type checkers, compilers, or model checkers. Also, we show how our techniques can be applied to existing programs without modifying the source code. Although, like assertions, our approach cannot prove the absence of errors, it gives the programmer a more powerful means of automatically checking assumptions about his program at runtime.

It can also be used to look for behaviour that indicates the *potential for problems*, that is, that might be used to predict future errors. An example is the Lock-order Reversal pattern which indicates a potential deadlock in a concurrent program.

Our *parametrised propositions* approach gives us a convenient way to handle *dynamic systems*: in real-world programs, almost always dynamic data is used, for example, new objects are instantiated, or new threads created. While there is already a plethora of work on checking some properties of those systems, they are usually concerned with the boundedness of the number of resources, and not necessarily with the interaction of those objects. Especially, they are often limited to either non-recursive examples or some coarse finite abstraction.

Propositions in our practical examples are certain events which we can observe in the execution of a program: method invocation with caller/callee and arguments, object attribute access, or actions related to concurrency.

If we have a property which has to be checked on a per-object basis (that is, instantiated for each object or set of objects), our template mechanism dynamically instantiates a given formula based on observers, so-called *existence predicates*. In a static approach, this would have to be solved by generating all instances of the property beforehand and checking each of them against the model, leading again to a finitary abstraction.

As application, we see two kinds of properties: firstly, properties over the *universal*

behaviour of data structures. For example, it universally holds that a pop from a stack should not be performed unless something has actually been pushed onto it. Data structures and their respective functionality are usually accumulated in libraries. Thus, a test facility implemented in the library should be available to any application using it. Furthermore, the corresponding formula can be used very much like a design pattern from Software Engineering and new implementations of some previously specified behaviour can be checked against the temporal specification of the pattern.

Secondly, there are *application specific properties*. These can be based, for example, on the specification that defined the application. Or they can be derived from informal requirements, stated in the documentation. Rather than being invariants of a structure like above, they may not be evident and need to be "developed" as much as the source code for the application has to be developed. As a lot of current research also focuses on documenting and enforcing such specifications, we later comment on the necessity and advantages of storing such *semantic annotations* about the application in a machine-readable format.

Naturally, the additional level of possible checks comes at a penalty: *runtime overhead*. In the most general case, for every event, the set of affected state machines (the automaton-based representation of the property) must be determined. This also includes possibly instantiating new ones. Then, in each machine, a state transition must be triggered. As the underlying framework of our mechanism is based on alternating finite automata, we already incur an exponential blow-up in the size of the automaton when moving to nondeterministic automata. The additional determinisation of said nondeterministic automata will make this approach infeasible in practice.

Instead, we will *resolve the nondeterminism at runtime* at the cost of performance: after statically generating an alternating finite automaton from the formula and eventually pre-calculating the (nondeterministic) outgoing edges for each state, we use a breadth-first approach over the branches for checking the acceptance condition of the input. We will see that this solution has double-exponential overhead, but we still consider it as a useful tool, since the average behaviour might prove not to be as harsh as this complexity might suggest. Also, using our algorithm offline, on a recorded trace, is an option that does not slow down the actual application.

Practical examples from object-based and concurrent programs written in Haskell, C, and Java underline the general usefulness of the approach. A proof-of-concept prototype developed in Java confirmed the practicality of our approach.

Acknowledgments

My sincerest thanks go to my supervisor Prof. Dr. Klaus Indermark, who gave me the opportunity as a teaching and research assistant to find my bearings in a rapidly advancing scientific world. His rigorous lectures on subjects that elsewhere are only taught as practical courses will sorely be missed.

I thank Prof. Bernd Finkbeiner, Ph. D., for kindly agreeing to be a member on the examination board.

I am indebted to Prof. Dr. Ir. Joost-Pieter Katoen, who, after taking over I2, provided unquestioned infrastructural support until this thesis could be finished and whose activism provided many additional last-minute insights into formal methods.

Both old and new colleagues at I2 formed an amiable group that made our department a pleasurable place to work on a day to day basis. I shall miss the friendly environment and their helpful contributions.

As for my office mate and invaluable friend Dr. Michael Weber, his suffering for (willingly or unwillingly) participating in my research and software development should not be underestimated. Thank you!

Finally, I also thank my friends in Aachen and abroad, for all their advice and making sure that there is in fact life outside of The Thesis.

Volker Stolz
Aachen, May 2006

Contents

1 Introduction

Assertions are a common feature of modern programming languages. They test assumptions about the program state at specific locations in the source code. An assertion contains a Boolean expression that should be true if execution is to proceed. If the expression is not true at runtime, the system will throw an error. Thus, an assertion confirms the programmers' assumption about the behaviour of the program, increasing confidence that the program is free of errors or at least terminates with an error description instead of returning incorrect results.

As assertions are tied to a specific source code location, they are only verified if execution reaches this point. Common usages for assertions are, for example, tests on the integrity of data structures by checking for null pointers, or out-of-bounds indices.

Some interesting properties of programs however are not limited to a specific location, but rather involve sequences of program points that must not occur. For example, the property that a file must not be used after it has been closed already involves two such locations (unless some state about the file is available). We call such properties spanning more than one source code location *Temporal Assertions*.

An advantage would be if the occurrence of such a sequence of locations was statically decidable. Unfortunately, for non-trivial programs this is not even decidable for a single source location under a specific variable assignment. With static techniques like abstract interpretation and model checking usually only an over-approximation can be computed, giving rise to false positives if the abstraction is to coarse. But still, at runtime, Temporal Assertions can be checked, and appropriate action be taken, like for plain assertions.

In this thesis, we present an extension to the well-known concept of assertions: *temporal assertions allow the specification and validation of modal properties of an application at runtime*. We see this as a necessary step in enforcing the growing number of implicit requirements of software specifications, which are often only informally defined in the documentation of application program interfaces (API) and are beyond the reach of type checkers, compilers, or model checkers. Also, we show how our techniques can be applied to existing programs without modifying the source code.

1.1 Runtime Verification

Our framework was developed in the context of Runtime Verification (for the workshop series see [72, 73, 112, 76]). Runtime Verification is about monitoring, analysing, and guiding the execution of programs, and especially interested in whether formal techniques can be applied at the runtime of a program to improve monitoring techniques such as (performance) monitoring, or distributed debugging.

Another focus is on runtime application of formal methods as a viable complement to the traditional methods that try to prove programs to be correct before their execution, such as model checking and theorem proving.

In *dynamic program analysis*, data is recorded during the execution of a program to subject it to analysis with regard to properties about the program. Often this has a severe impact on performance and is usually only applied in the development phase of a program. It can also be applied conditionally to investigate only certain behaviour.

Finding a *specification language* and suitable semantics for properties goes hand in hand with operational constraints, such as, what kind of trace data can sensibly be generated and reasoned about. Often this will be a formal model where singleton events indicate state transitions in the program. Also, the underlying logic may be closely interwoven with an operational semantics since it may require additional evaluation of predicates in the context of the application. Or the program semantics may be independent from the trace semantics, so that offline analysis on a recorded trace is possible. As a trace is a sequence of events or sets of propositions, most formal models use some temporal logic which often has its origin in model checking, like the Linear Time Logic LTL.

Traces are usually obtained through *program instrumentation*, that is, an existing program, either in source code format or as a compiled executable, must be adapted to generate the relevant data.

This thesis covers the concepts above in varying depths. An additional field of investigation which is out of the scope of our work is *program guidance*, where based on (intermediate) analysis results, the program is directed to, for example, paths which might be prone to violate the specification. In a multi-threaded system, the process scheduler could be of use to produce a certain behaviour by following a certain strategy when choosing the next thread to execute.

For us, an error message and a *witness*, that is, (a part of) the trace up to the position where a violation occurred are sufficient. A related topic is *contributing behaviour*, that is, specifying an action that is to be executed when dynamic analysis detects a failure. Techniques for this range from simply raising an exception to gracefully handling the problem.

In the last five years, popularity of Runtime Verification tools has surged due to the success of some initial prototypes. These tools have been refined to be more efficient and designed in a modular way to support input from various sources and intermediate representations so that different (temporal) formalisms can be implemented on a single back end. This has been pursued by the MaC framework [88] or the expressively rich and efficiently monitorable EAGLE logic [13].

Automata-based techniques have also been used recently in the Intrusion Detection System ORCHIDS [100]. MONID [97] is an Intrusion Detections System that uses the EAGLE logic to check safety formulae of known attack patterns.

Why Runtime Verification? On a first glance, if we have model checking or other static analysis tools being able to verify important properties about a program, why should we have to implement checks at runtime?

Unfortunately, for real-world applications written in real-world programming languages like Java or C, static verification still poses some fundamental problems: the reachability of a specific line of code somewhere in the program in a concrete run is only a semi-decidable property. In many cases, an over-approximation gives adequate results. For example, the flow graph of a program is decidable, but also contains paths which cannot actually occur in a concrete run (see Section 2.4). With respect to verification, every such path might thus trigger a *false positive*, that is, the property at hand to check might indeed not hold on a path in the model, but this does not matter if the path is never taken by an actual execution. Nevertheless, the tool doing the verification would have to emit a warning to the user who has to decide whether it is significant.

In practice, such warnings occur very often, for example, the static C checker Splint [51, 50] literally spews forth warnings on a plain program. To get rid of the warnings, either the code has to be modified, or the source has to be enriched with *annotations* to provide more information to the checker. In fact, it is recommended practice to iterate the process of running Splint and annotating the code until all warnings are fixed and possible errors flushed out. Due to the general undecidability of static program analysis, there will always be a remainder of warnings that cannot be eliminated.

The authors admit that annotating code is in fact the greatest obstacle to adoption of their checker. This holds especially true for legacy code, both in the form of libraries or applications. Clearly, getting spurious warnings that have to be sorted out is not a desirable feature, although in practice, a certain number of false positives can surely by acceptable.

Static verification is marketed with apparent success by Coverity, a Stanford University spin-off [47], despite also suffering from spurious warnings if no additional annotations to the program are provided.

The properties we are interested in in this thesis depend on both the control flow of the application and the values of program variables. This includes, for example, object references in the form of arguments to method invocations (*which object* invokes some method), or primitives for concurrency control, like thread identifiers and semaphores (*which lock* is taken).

1.2 Model Checking

Naturally we are interested in whether such properties can be determined statically. This could be achieved through model checking, a verification technique that explores *all possible* system states. It can be applied to both hardware and software if an appropriate model for the system can be derived. Its origins go back to independent research by Clarke and Emerson [32], and Queille and Sifakis [105]. A survey on model checking can be found in [107]. Monographs covering the subject include [33] and [79].

Properties to check are specified in linear- or branching-time logic like LTL or CTL*. For example, "Is this state reachable from a given start state?", "Is this state *infinitely often* reachable?", and, "Can this multi-threaded program deadlock?". The SPIN model checker by Holzmann [78] is probably the best known implementation of a model checker and has successfully been applied to protocol and software verification. Errors discovered through model checking have led to revisions and improvements in industrial protocols and safety-critical applications like air traffic control and medical appliances.

Even big companies like Microsoft (SLAM/Static Driver Verifier [12]), HP (threading tools [66]), or IBM through its Formal Verification and Testing Technologies group are investing in this trend. Other high-profile users includes NASA, which verified several components of spacecraft software for different Mars missions through model checking techniques [69]. Also, *timed* or *stochastic* properties of systems are studied [3, 9], that is, where properties like "Will a leader in a group be elected in under 1 second", or "What is the probability of electing a leader in n rounds of a probabilistic protocol?" are verified.

We will see later in Chapter 2 that model checking for software programs with dynamic data structures and recursion poses some problems due to the infinite nature of the models, although several fruitful results have been obtained in restricted settings for real-world programming languages like JAVA, C, or C++, for example, by Godefroid [62] or Dwyer *et al.* [67].

1.3 Temporal Assertions

The framework we introduce shall provide the user or developer with a new style of assertions for sequential and concurrent programs: instead of simply asserting a Boolean expression at a specific location, we provide a convenient way of specifying monitors that check **modal safety properties of the dynamic control flow expressed in Linear Time Logic (LTL)**. It can also be used to look for behaviour that indicates the **potential for problems**, that is, that might be used to predict future errors. This is effectively what the Lock-order Reversal pattern captures (see Section 4.2).

Our **parametrised propositions** approach gives us a convenient way to handle **dynamic systems**: in real-world programs, almost always dynamic data is used, for example, new objects are instantiated, or new threads created. While there is already a plethora of work on checking some properties of those systems, they are usually concerned with the boundedness of the number of resources, and not necessarily with the interaction of those objects [40, 41]. Especially, they are often limited to either non-recursive examples or some coarse finite abstraction.

Propositions in our practical examples are certain events which we can observe in the execution of a program: method invocation with caller/callee and arguments, object attribute access, or actions related to concurrency.

If we have a property which has to be checked on a per-object basis (that is, instantiated for each object or set of objects), our template mechanism dynamically instantiates a given formula based on observers, so-called **existence predicates**. In a static approach, this would have to be solved by generating all instances of the property beforehand and checking each of them against the model, leading again to a finitary abstraction.

As application, we see two kinds of properties: firstly, properties over the **universal behaviour of data structures**. For example, it universally holds that a pop from a stack should not be performed unless something has actually been pushed onto it. Data structures and their respective functionality are usually accumulated in libraries. Thus, a check implemented in the library should be available to any application using it. Furthermore, the corresponding formula can be used very much like a design pattern from Software Engineering and new implementations of some previously specified behaviour can be checked against the temporal specification of the pattern.

Secondly, there are **application specific properties**. These can be based, for example, on the specification that defined the application. Or they can be derived from informal requirements, stated in the documentation. Rather than being invariants of a structure like above, they may not be evident and need to be "developed" as much as the source code for the application has to be developed. As a lot of current research also focuses on documenting and enforcing such specifications, we later comment on the necessity and advantages of storing such **semantic annotations** about the application in a machine-readable format in Section 5.5.

Naturally, the additional level of possible checks comes at a penalty: **runtime overhead**. In the most general case, for every event, the set of affected state machines (the automaton-based representation of the property) must be determined. This also includes possibly instantiating new ones. Then, in each machine, a state transition must be triggered. As the underlying framework of our mechanism is based on alternating finite automata, we already incur an exponential blow-up in the size of the automaton when moving to nondeterministic automata. The additional determinisation of said nondeterministic automata will make this approach infeasible in practice.

Instead, we will **resolve the nondeterminism at runtime** at the cost of performance: after statically generating an alternating finite automaton from the formula and eventually pre-calculating the (nondeterministic) outgoing edges for each state, we use a breadth-first approach over the branches for checking the acceptance condition of the input. We will see that this solution has double-exponential overhead, but we still consider it as a useful tool, since the average behaviour might prove not to be as harsh as this complexity might suggest. Also, using our algorithm offline, on a recorded trace, is an option that does not slow down the actual application.

1.4 Outline

First, we introduce a very simple object-based language in Chapter 2, which is expressive enough to be comparable to existing real-world programming languages like Java or C. Using this language, we comment on the current limitations of model checking programs with recursion and dynamic data structures based on results from program analysis.

In Chapter 3, we recapitulate the Linear-time Logic LTL and how LTL properties can be checked on finite paths through alternating finite automata. The conventional framework is then extended by our parametrised propositions which offer an additional degree of conciseness in the representation.

The mapping from events to a trace over sets of propositions is discussed in Chapter 4 which gives examples for the evaluation of traces and also presents several use cases where LTL properties can be used.

Chapter 5 focuses on applying our technique to existing programs, most notably written in Haskell, C, and Java. Advanced programming concepts are discussed that help with efficiently deploying the necessary instrumentation.

Finally, we conclude with related work and a summary in Chapter 6.

2 Reasoning about Programs

In this section, we motivate the need for being able to check certain properties of the program with respect to a specification through some examples from object-oriented and concurrent programming.

We introduce an object-based, dynamically typed language that allows us to reasonably set the stage for a formal framework without having to model complex languages like JAVA or C. After giving its execution semantics, we take a look at different ways of obtaining a model which might be suitable for verification. We point out certain limitations to a static, sound approach.

Given a set of actions occurring during the execution of an object-based program (read/write-access to object attributes and method calls), we show how to obtain a path where each element contains an event. Such a path resembles the observed execution trace of the running program.

2.1 Properties of Programs

To avoid misbehaviour, many software products include assertions which check that certain states on the execution path satisfy given constraints and otherwise either abort execution or execute specific error handling. These assertions are usually limited to testing the values of variables. However, often it would be convenient not only to reason about a *single state* but also about a *sequence of states*.

Frequently it happens that certain functionality is only available at certain points during the time when an application executes, or in other words: at certain times at runtime, certain features like certain methods or objects should *not* be allowed to be accessed for the sake of a safe and stable application.

For example, nothing should be written to a file, if the file has been closed already. Such errors may be documented in the API in the form of comments, but still the user of the file has to remember to obey this rule in order to get a safely working application. We argue that given an appropriate framework of *temporal assertions*, such properties can be checked and enforced automatically at runtime. To further emphasize this dynamic view we give an example.

Safe Iterator Usage

Commonly, a large set of libraries is available to application programmers which offer a variety of often needed functionality. Generic implementations on sets, lists, and other data structures are shipped together with their respective API documentation. They often have dynamic requirements that pose certain obligations on consumers.

For example, we find the following comment in the JAVA 1.5 API documentation for `Iterator.remove()`:

> *"The behaviour of an iterator is unspecified if the underlying collection is modified while the iteration is in progress in any way other than by calling this method."*

Apart from being necessary at all, we find it puzzling that a rather general point about the (temporal) interface is made in a subordinate location in the documentation.

The requirement of the *safe iterator* design pattern above for object-oriented languages such as JAVA can be reformulated in the following way:

> *For each Iterator `i` obtained from a Collection `c`, there must never be an access to the iterator (that is, `i.next()` or `i.remove()`) after the collection has been modified.*

This is a universal statement about all iterators and all collections.

The pattern is in fact enforced in the JAVA5 library implementation: the `Iterator` implementation contains a mechanism to track modifications of the underlying collection by means of a modification counter. If the collection c is updated, the modification-count obtained by the iterator i on instantiation time and the current counter of the collection disagree and lead to an exception on the next access to the iterator. In this case, the specification has crept into the implementation of both the iterator and the collection. There is no way for users to turn the additional check off if they have made sure that they obey the rule. Fortunately in this case, the overhead both in terms of memory and computation is unnoticeable: the overhead amounts to a single integer variable for each collection and iterator, an increment instruction on each collection access and the corresponding test on each iterator access.

Stacks

A similar problem is checking the safe use of stacks:

> *For every stack `s`, there must not be a pop through `s.pop()` until some item `x` has been pushed onto it through `s.push(x)`.*

This statement also makes a general claim about the data structure. An advanced feature would be asserting that never more items are popped from the stack than have been pushed onto it.

Note the subtle difference between these two very similar descriptions of erroneous behaviour: in the first case, we are only talking about some initial access to the stack. The latter property however describes the context-free nature of stack accesses since we must match corresponding push and pop actions.

A More General Approach

Instead of forcing developers to encode such properties in their implementations, we wish to provide a more reasonable way of declaratively specifying safe (or, by complement, unsafe) behaviour of applications.

Such properties over patterns of source code locations in the control flow of an application can be a way to specify safety patterns in the sense of "something bad never happens" which can then be checked at runtime. In the previous examples, the patterns consist of method invocations, for example, `Iterator.next()`, `Iterator.remove()`, `Stack.pop()`, or `Stack.push()`.

Before providing a theoretical framework for checking such properties, we want to set the stage by introducing a small object-based programming language and its semantics. We shall see that we can collect the sequence of events (like method enter and exit) with their respective valuations for arguments/variables induced by an execution of the program. We can then subject such a trace to formal verification techniques.

2.2 An Object-based Programming Language

Our main interest is to be able to handle a large part of programs developed using current programming languages like Java, C, or C++.

We need to choose an appropriate representation of programs that will work for large classes of programs in those languages. Since we are going to observe a running program, we are not interested in high-level control structures of the languages like `while` or `for-next`, but only in some instructions modifying the program state. For an object-based program, the program state consists of a program counter, the current call stack and a heap. Objects reside in the heap, local variables in the stack frame. The stack frame also contains runtime information such as the return address.

Intermediate-level Language $\mathbb{PL}_{int}$

Our object-based intermediate-level language $\mathbb{PL}_{int}$ provides the following features:

- data types containing `void`, `bool`, `nat`, and object references

- atomic operations on said data types
- classes with attributes (fields) and methods
- method invocation with return values
- method-local variables
- conditional and unconditional branching
- object creation

The data type `void` is the unit type with the trivial value `nil`. `bool` is the type of Boolean values TRUE and FALSE, `nat` is the type of natural numbers.

$\mathcal{C}$ is a set of *classes*. Each class $C \in \mathcal{C}$ has an associated set of *attributes* $\mathcal{A}_C$, *methods* $\mathcal{M}_C$, and *objects* $\mathcal{O}_C$. We define the set of all attributes $\mathcal{A}$, methods $\mathcal{M}$, and objects $\mathcal{O}$ accordingly:

$$\mathcal{A} := \bigcup_{C \in \mathcal{C}} \mathcal{A}_C, \qquad \mathcal{M} := \bigcup_{C \in \mathcal{C}} \mathcal{M}_C, \qquad \mathcal{O} := \bigcup_{C \in \mathcal{C}} \mathcal{O}_C.$$

Let $\mathcal{V}$ be a countable set of (untyped) *variables* containing at least the special member `this`. The set of *attribute expressions* is defined as

$$AExps := \{v.a \mid v \in \mathcal{V}, a \in \mathcal{A}\}.$$

F denotes the set of *operator symbols*. Let $F^{(n)}$ denote the functions of arity n ($\mathcal{M}_C^{(n)}$ and $\mathcal{M}^{(n)}$ are defined respectively). $F^{(0)} \subseteq F$ denotes the set of *constants* with at least representations for values of the above data types excluding object references, that is, $\{\texttt{nil}, \text{TRUE}, \text{FALSE}\} \cup \mathbb{N} \subseteq F^{(0)}$.

We assume that high-level language constructs for structured programming like `while` or `for-next` have already been eliminated through the standard techniques of compiler construction. We limit our language to conditional branches and unconditional jumps. Furthermore, we also assume that complex expressions in an original program have already been broken down in code with expressions only containing *at most one* operator with only variables or constants as arguments. Note that this means that we forbid attribute access and method invocation in a function application: values must be assigned to local variables first. The rationale behind this is to obtain an explicit ordering of such accesses in the execution of a program. This allows us to keep the language and latter definitions concise without loosing expressiveness.

Definition 1 (Method)
Let M denote a *method* with $M = \langle MId, \mathcal{V}_M, \texttt{IList}_M \rangle$, where MId is a *method name*, $\mathcal{V}_M = \langle Locs_M, IVars_M \rangle \in 2^{\mathcal{V}} \times 2^{\mathcal{V}}$ disjoint sets of local variables and formal parameters, respectively, and $\texttt{IList}_M \in \mathbb{Instr}^*$ a sequence of instructions (see Table 2.1 for the syntax of instructions). We shall use the assignment operators `:=` (in source code) and $\leftarrow$ (in the semantics) interchangeably for better readability.

x ← o.a	$x \in \mathcal{V}, o.a \in AExps$: load attribute into variable
x ← vc	$x \in \mathcal{V}, vc \in \mathcal{V} \cup F^{(0)}$: assign value to variable
o.a ← vc	$o.a \in AttrExps, vc \in \mathcal{V} \cup F^{(0)}$: assign value to attribute
x ← `new` C	$x \in \mathcal{V}, C \in \mathcal{C}$: create object of class C and bind object to variable/attribute
x ← $f(\text{arg}_1, \ldots, \text{arg}_n)$	Function application $f \in F^{(n)}$, $x \in \mathcal{V}$, $\text{arg}_i \in \mathcal{V} \cup F^{(0)}$
x ← $o.m(\text{arg}_1, \ldots, \text{arg}_n)$	Call n-ary method m on object $o \in \mathcal{V} \cup \{\texttt{this}\}$, $m \in \mathcal{M}^{(n)}$, $\text{arg}_i \in \mathcal{V} \cup F^{(0)}$, store return value in $x \in \mathcal{V}$
`return` vc	Return from method with result $vc \in \mathcal{V} \cup F^{(0)}$
`jmp` n	Unconditional jump to specified instruction, $n \in \mathbb{N}$
`jmf` x n	Conditional jump if $x \in \mathcal{V}$ bound to FALSE, $n \in \mathbb{N}$

Table 2.1: Low-level instructions $\mathbb{Instr}$

Each instruction is prefixed with its line number in the method, starting with 1. For convenience, let a *location* $\alpha = C.m.i$ denote the ith instruction of method m in class C. A location may be used interchangeably with or without the instruction corresponding to this location. Line numbers are omitted in listings if not necessary; we will use symbolic labels as jump-targets for better readability.

Definition 2 (Class)
A class $C = \langle CId, \mathcal{A}_C, \mathcal{M}_C \rangle$ has a unique name CId, a set of attributes $\mathcal{A}_C = \{a_{C,1}, \ldots, a_{C,n}\}$, and a set of methods $\mathcal{M}_C = \{M_{C,1}, \ldots, M_{C,m}\}$.

Definition 3 (Object, Dynamic type)
Objects are unique numbered instances of classes:

$$\mathcal{O} := \{\langle C, n \rangle \mid C \in \mathcal{C}, n \in \mathbb{N}\}$$

The class of an object (its *dynamic type*) can be obtained through

$$\tau : \mathcal{O} \to \mathcal{C}$$
$$\tau(\langle C, n \rangle) := C.$$

Definition 4 (Program)
A *program* $\pi := \langle \mathcal{C}_\pi, (\text{arg}_1, \ldots, \text{arg}_n) \rangle$ consists of a set of classes $\mathcal{C}_\pi = \{C_1, \ldots, C_m\}$ and formal parameters $\text{arg}_1, \ldots, \text{arg}_n$. We require the existence of a class `Main` $\in \mathcal{C}_\pi$ which contains at least a method `main` $\in \mathcal{M}^{(n)}_{\texttt{Main}}$. The formal parameters of a program are defined by those of the distinguished method `Main.main`.

Additionally, we require that the last instruction of each potential execution shall be a `return` statement.

A small application that creates an empty stack, pushes an element of some type O unto it, and then iterates over the contents of the stack is given in Figure 2.2. The implementation uses an empty dummy element for maintaining the current head of the stack. Subsequent elements are kept in a singly linked list. The corresponding $\mathbb{PL}_{int}$ program can be found in the Appendix as Figure 6.1.

$\langle$PROGRAM$\rangle$ ::= $\langle$CLASS$\rangle^+$
$\langle$CLASS$\rangle$ ::= **class** *classname* [**var** *attrname*$^+$] $\langle$METHOD$\rangle^*$
$\langle$VC$\rangle$ ::= *var* | *constant* | **nil**
$\langle$AEXP$\rangle$ ::= (*var* | **this**).*attrname*
$\langle$FEXP$\rangle$ ::= **f (** $\langle$VC$\rangle^*$ **)**
$\langle$METHOD$\rangle$ ::= **method** *methodname*(*var**) [**var** *var*$^+$] [$\langle$INSTR$\rangle^*$]
$\langle$INSTR$\rangle$ ::= *var* $\leftarrow$ $\langle$VC$\rangle$ | $\langle$AEXP$\rangle$ | $\langle$FEXP$\rangle$ | **new** *classname*
| *var* $\leftarrow$ (*var*|**this**).*methodname*($\langle$VC$\rangle^*$)
| $\langle$AEXP$\rangle$ $\leftarrow$ $\langle$VC$\rangle$
| **jmp** $\mathbb{N}$
| **jmf** *var* $\mathbb{N}$
| **return** $\langle$VC$\rangle$

Figure 2.1: Grammar for $\mathbb{PL}_{int}$ programs

```
class Main
  method main
    var stack v o
    stack:=new Stack
    o:=new O
    stack.push(o)
    v:=stack.next // head
    while (v != nil) {
      o:=v.o
      ...
      v:=v.next
    }
    return nil

  // main
// Main
```

```
class Stack
  var o    // Container
      next // Link

  method push(v)
    var t
    t:=new Stack
    t.o:=v
    t.next:=this.next
    this.next:=t
    return nil

  method pop()
    var t
    t:=this.next
    this.next:=t.next
    return t.o
```

Figure 2.2: High-level program using an object-based stack

```
class Main

  method main(bool)
    var t o

    if bool then
      o := new Duck
    else
      o := new Grouse
    t := o.quack()
    return void
  // quack
// Bird
```

Listing 2.1: Static type of variable

```
class Main

  method main(bool)
    var t o

    jmf bool L1
    o := new Duck
    jmp L2
L1: o := new Grouse
L2: t := o.quack()
    return void
  // quack
// Bird
```

Listing 2.2: Static typing in $\mathbb{PL}_{Int}$

Remark 5 (Structural typing)
Note that our language in that sense permits more behaviour than a corresponding Java (bytecode) program. The example in Listing 2.1 is permitted in our language even though `Duck` and `Grouse` cannot have a common superclass, since our language is only object-based, but not object-oriented. In the Java programming language, the type of object o would have to be declared as said superclass.

This behaviour is implemented in many dynamically typed programming languages such as Smalltalk, Ruby, or Python. A similar paradigm is available in the Common LISP Object System CLOS.

2.3 Execution Semantics

We can now establish an execution semantics for a program based on an interpretation for values and function symbols. Program arguments are passed via the `main` method.

Definition 6 (Interpretation)
An *interpretation* $\mathcal{I} := \langle Dom, \xi \rangle$ consists of a value domain Dom and a mapping ξ of the constant and function symbols into functions over Dom:

$$\xi : F \to \bigcup_{i=0}^{\infty} \{\delta \mid \delta : Dom^i \to Dom\} \text{ with}$$
$$\xi(f) : Dom^r \to Dom \text{ for every } f \in F^{(r)}$$
$$\xi(c) \in Dom \text{ for every } c \in F^{(0)}$$

In the following, we assume that Dom contains at least representations for the Boolean truth values TRUE, FALSE and $\perp$ for the undefined value `nil` (for example, on access to uninitialised variables).

Definition 7 (Heap)
Since we use an object-based language, a *heap* is modelled as a partial mapping h from objects $\mathcal{O}$ to valuation functions *Val* of their attributes $\mathcal{A}_C$:

$$\begin{aligned} ODom &:= Dom \cup \mathcal{O} \\ Val &:= \{\lambda \mid \lambda : \mathcal{A}_C \to ODom, C \in \mathcal{C}\} \\ \mathcal{H} &:= \{h : \mathcal{O} \dashrightarrow Val \mid o = \langle C, n\rangle \in Def(h) \text{ such that } h(o) : \mathcal{A}_C \to ODom\} \end{aligned}$$

Let $h(o).[a] := (h(o))(a)$ return the attribute $a \in \mathcal{A}_{\tau(o)}$ from an object $o \in \mathcal{O}$, if it is in the heap $h \in \mathcal{H}$. $h(o).[a/value]$ returns a modified heap where the attribute a of object o has been updated to $value \in ODom$:

$$h(o).[a/value] := h', \text{ with } h'(o') := \begin{cases} h(o'), & \text{if } o \neq o' \\ \lambda', & \text{otherwise} \end{cases}$$

$$\text{where } \lambda'(a') := \begin{cases} value, & \text{if } a = a' \\ (h(o))(a'), & \text{otherwise} \end{cases}$$

Definition 8 (Frame, Active frame, Runtime stack)
The *runtime stack* is a colon delimited sequence of *frames* $f_1 : f_2 : \ldots : f_n \in \mathcal{F}^*$, where each frame contains as first component an object reference $o \in \mathcal{O}$ into the heap for the currently active object, a return address $\alpha \in Loc$, an indirection to a storage location in the previous frame where the return value should be stored, and entries for local variables and arguments of type *ODom*:

$$\mathcal{F} := \mathcal{O} \times Loc \times \mathbb{N} \times \overbrace{ODom \times \ldots \times ODom}^{local\ \ variables} \times \overbrace{ODom \times \ldots \times ODom}^{parameters}$$

The frame stack grows to the left, the left-most frame is the *active frame*.

Often, access to the topmost frame is denoted by $t : st$, where t is the active frame and st is the remainder of frames. Local variables are separated by $\cdot$ from arguments. Storage locations for method arguments and local variables can be accessed by name through $t[var]$, updates to local variables are denoted by $t[var/value]$, where $var \in \mathcal{V}, value \in ODom$. Furthermore, $t[\texttt{this}] := \underline{proj}_1(t)$, where $\underline{proj}_i$ denotes projection to the ith component of a tuple. We shall use *this* : $\mathcal{O}$ as a synonym for the currently active object. An actual implementation would translate variable names to offsets in the stack frame.

For convenience, we define the interpretation function also with respect to variables bound in a stack frame:

$$\xi : \mathcal{F} \times (\mathcal{V} \cup F^{(0)}) \to ODom$$

$$\xi(t, x) := \begin{cases} t[x] \in ODom, \text{if } x \in \mathcal{V} \\ \xi(x), \text{if } x \in F^{(0)}. \end{cases}$$

Size and layout of each object and frame are known at compile time. Many well-known extensions to programming languages such as dynamic data structures can be easily added to the language and its semantics, but have been omitted for conciseness.

Definition 9 (Single-instruction semantics)
The semantics of a single instruction given a program π and interpretation $\mathcal{I}$ yields the next location to execute and a transformation of the current stack (split into the active frame t and the remaining frames st) and heap h:

$$\texttt{exec}^{\pi}_{\mathcal{I}} : \mathit{Loc} \times \mathcal{F}^* \times \mathcal{H} \rightarrow \mathit{Loc} \times \mathcal{F}^* \times \mathcal{H}$$

We assume that variables are used correctly with regard to their visibility, that is, generally $v \in \mathit{Locs}_m$ and $o, x \in \mathit{Locs}_m \cup \mathit{IVars}_m$.

1. Atomic operations over value domain

   $\texttt{exec}^{\pi}_{\mathcal{I}}(C.m.i : v \leftarrow f(x_1, \ldots, x_n),\ t : st, h),$
   with $v \in \mathcal{V},\ f \in F^{(n)},\ x_j \in \mathcal{V} \cup F^{(0)},\ n \geq 0$
   $= (\ \underbrace{C.m.(i+1)}_{\textit{next instruction}}\ ,\ t[v/\ \underbrace{\xi(f)(\xi(t, x_1), \ldots, \xi(t, x_n))}_{\textit{evaluation w.r.t. current stack}}\ ] : st,\ h).$

2. Object instantiation

   $\texttt{exec}^{\pi}_{\mathcal{I}}(C.m.i : v \leftarrow \texttt{new}\ D,\ t : st, h) = (C.m.(i+1),\ t[v/o] : st,\ h'\}),$ where $o = \langle D, j\rangle \in \mathcal{O}$ is a fresh instance of class D, that is, $h(o)$ is undefined, and
   $$h'(o') := \begin{cases} \lambda_\perp, & \text{if } o = o' \\ h(o'), & \text{otherwise} \end{cases}$$
   where $\lambda_\perp$ denotes the entirely undefined attribute mapping where all attributes are mapped to the undefined value.

3. Assignment from attribute to a local variable

   $\texttt{exec}^{\pi}_{\mathcal{I}}(C.m.i : v \leftarrow o.a,\ t : st, h)$, with $v, o \in \mathcal{V},\ a \in \mathcal{A}_D$, and
   $D = \tau(t[o])$ is the dynamic type of the object reference by variable o
   $= (C.m.(i+1),\ t[v/h(t[o]).[a]] : st,\ h)$

4. Assignment to attribute

   $\texttt{exec}^{\pi}_{\mathcal{I}}(C.m.i : o.a \leftarrow x,\ t : st, h),$ with $o \in \mathcal{V},\ D = \tau(t[o]),\ a \in \mathcal{A}_D,\ x \in \mathcal{V} \cup F^{(0)}$
   $= (C.m.(i+1),\ t : st,\ h(t[o]).[a/\xi(t, x)]).$

5. Method invocation

   $\texttt{exec}^{\pi}_{\mathcal{I}}(C.m.i : v \leftarrow o.m'(x_1, \ldots, x_n),\ t : st, h),\ o, v \in \mathcal{V},\ D = \tau(t[o]), m' \in \mathcal{M}^{(n)}_D,$
   $n \geq 0, x_j \in \mathcal{V} \cup F^{(0)},\ t = (\mathit{this}, \mathit{ra}, \mathit{off}, l_1, \ldots, l_{k-1}, l_{[v]}, l_{k+1}, \ldots \cdot \mathit{args})$
   $= (D.m'.1,\ \underbrace{(t[o], C.m.(i+1), k, l'_1 \ldots l'_{|\mathit{Locs}_{D.m'}|} \cdot \xi(t, x_1) \ldots \xi(t, x_n))}_{\textit{new frame}} : t : st,\ h).$

   k denotes the location $l_{[v]}$ of the local variable v in the original stack frame.

6. Return from method

   The kth local variable in the caller's frame t' is updated to the computed value.

   $$\begin{aligned}&\texttt{exec}^{\pi}_{\mathcal{I}}(C.m.i : \texttt{return}\ v,\ t : t' : st, h), v \in \mathcal{V} \cup F^{(0)},\\ &\quad t = (o, \alpha, k, \ldots),\ t' = (o', \alpha', k', l_1, \ldots, l_k, \ldots \cdot args)\\ &= (\alpha, t'[l_k/\xi(t, v)] : st, h)\end{aligned}$$

7. Unconditional jump

   Jumps are only allowed inside of the same method.

   $$\texttt{exec}^{\pi}_{\mathcal{I}}(C.m.i : \texttt{jmp}\ \mathrm{j}, st, h) = (C.m.j, st, h)$$

8. Conditional jump on false

   $$\texttt{exec}^{\pi}_{\mathcal{I}}(C.m.i : \texttt{jmf}\ v\ j,\ t : st, h) = \begin{cases}(C.m.j,\ t : st, h), \text{ if } \xi(t, v) = \textsc{false},\\ (C.m.(i+1),\ t : st, h) \text{ otherwise}\end{cases}$$

Definition 10 (Iteration semantics)
$\texttt{execI}^{\pi}_{\mathcal{I}} : Loc \times \mathcal{F}^* \times \mathcal{H} \to \mathcal{F}^*$ is the *iteration semantics* of an instruction that yields the final stack on termination which may contain the computed values:
$\texttt{execI}^{\pi}_{\mathcal{I}}(\alpha, st, h) :=$

- st iff $\alpha = \texttt{Main.main.0}$ (program termination), or
- $\texttt{execI}^{\pi}_{\mathcal{I}}(\texttt{exec}^{\pi}_{\mathcal{I}}(\alpha, st, h))$ otherwise
  (single step semantics of an instruction)

Definition 11 (Semantics of a program)
Given a program π and an interpretation $\mathcal{I} := \langle Dom, \xi \rangle$, we define the execution semantics $\texttt{Exec}^{\pi}_{\mathcal{I}} : Dom^n \to Dom$ with respect to the arguments as required by `Main.main`. Note that objects can only occur inside the program, but neither be arguments nor the result of a program.

$$\begin{aligned}&\texttt{Exec}^{\pi}_{\mathcal{I}} : Dom^n \to Dom\\ &\texttt{Exec}^{\pi}_{\mathcal{I}}(args) := \underline{proj}_4(\texttt{execI}^{\pi}_{\mathcal{I}}(\texttt{Main.main.1}, initialStack, \{\langle \texttt{Main}, 1\rangle\})),\ \text{with}\\ &initialStack := (\langle \texttt{Main}, 1\rangle, \texttt{Main.main.0}, 1, locs \cdot args) : (\bot, \bot, \bot, \Box)\end{aligned}$$

Accordingly, the initial stack consists of two frames: The active frame for the `main` method where execution starts, and a frame with just a place holder ($\Box$) for the return value of the ultimate `return` statement when execution terminates.

In the active frame, the `this` reference points to the first instance of the `Main` class in the otherwise empty heap. The return address points to a nonexistent instruction to halt execution.

2.4 Static Analysis

In the context of verification, a question naturally arises, namely whether some property can be statically decided for the programming language just presented. Before proceeding with some additional features, we take the time to highlight some properties and problems of checking programs statically.

The properties we wish to reason about relate different points in the execution of a program. A single such dynamic execution point can be split in two parts: firstly, a specific line of code (for example, a method call), and secondly, its context, that is the runtime information, comprising the call stack with variable assignments.

In order to decide whether a specific source code location is reachable under some variable assignment, we need to consider all possible execution paths of the program for every possible input.

Clearly determining the set of *all execution paths* of a non-trivial program is not practicable due to the enormous size of possible inputs. Even for a program with only one integer parameter we would still have to check 2^{32} programs. Real-world programs would even have unlimited input throughout the program, for example, from the disk, or the network.

Consequently, a static analysis of the program would have to abstract from the concrete input. Alas, this means that the analysis must proceed with some arbitrary, possibly symbolic, value for *unknown/"don't know"* for all input parameters. Generally the execution of a non-trivial program will in some way always depend on its input, since otherwise the program (or parts thereof) would represent some constant expression.

Hence, every time an instruction on the execution path encounters such an unknown value, the uncertainty is propagated: for expressions like arithmetic, if one of the arguments is unknown, the analysis can only infer an unknown result.

In Boolean expressions necessary for conditional branches like `if-then-else`, imprecise information gives rise to non-determinism: when we are uncertain which branch to take, we need to proceed in both branches.

With respect to the soundness of the result, up until here, we note the following:

1. By abstracting from the input, we need an appropriate data abstraction to analyse the program with respect to unknown values for variables. If we then determine that we can reach an instruction which resembles the static part of a property we want to check, we may do so with only imprecise information about the variable bindings. Consequently, with respect to the property, we will not be able to determine the exact valuations for the variables in the general case. For the example from the motivation, this may mean that although we are popping an item from *some* stack, we may not know whether this is one of the stacks we previously recorded a `push` action for. Thus, the correctness property cannot be guaranteed.

2. The non-deterministic branches also introduce paths that might never be taken for an actual program run. So even if we reach a program location with a

concrete variable binding, this path might pose a false positive if the analysis leading to this path is to coarse.

Additionally, we must model recursion through method invocation. This is naturally modelled through a context-free set of program paths [99], although for the concrete system there will usually be some fixed upper bound on the stack size because of the available memory. While this would be a precise abstraction, unfortunately it has to be combined with the above result on non-determinism of conditional branches: this augments the set of paths in the model but not in an actual execution and thus may lead to false positives.

Context-free control-flow properties and memory allocation of Java Card Applets have been investigated by Fredlund *et al.* in [31] and [54]. The latter also introduces the technique of adding runtime monitors to the applet in the case that its properties could not be verified statically. Properties are expressed in LTL and verified through pushdown (that is, context-free) systems with the Moped model checker [48]. The abstracted model is a safe over-approximation of context-free call graphs as discussed above.

The temporal logic of calls and returns CaRet [4] allows to check some non-regular properties of pushdown systems with the same complexity as the LTL model checking problem (polynomial in the size of the model and exponential in the size of the formula), although checking context-free properties in general is undecidable for pushdown systems.

In the verification community, the full-featured abstraction of programming languages has proved problematic, for a survey on model checking C see [109]. Some success for JAVA has been achieved through the Java PathFinder [125], although without additional instrumentation the state-space exhausts available memory.

Predicate abstraction [64] (for example used in Microsofts SLAM toolkit [11, 12]) is another promising approach that preserves some information about abstracted input, for example, that although the concrete value for some x is unknown, it may be less than some n on one path of an `if-then-else` for the Boolean expression $x < n$, and greater or equal on the other path.

We conclude that there is no static verification toolkit at the moment that could serve as a "silver bullet" due to the huge variety of necessary techniques, and that the only time when we can reliably decide whether a property is fulfilled is at runtime, although this naturally precludes us from drawing any conclusions about results obtained under other inputs. As Dijkstra already noted [39]:

> *"Program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence".*

We continue our discussion of our programming language with additional features.

2.5 Assertions

Assertions are statements in the program that test assumptions about the program. An assertion contains a Boolean expression that should be true if execution is to proceed. If the expression is not true at runtime, the system will throw an error. Thus, an assertion confirms the programmer's assumption about the behaviour of the program, increasing confidence that the program is free of errors, or at least that it terminates with an error description instead of returning incorrect results.

During debugging, additional output is generated for violated assertions. For example, the JAVA programming language since JDK 1.4 offers such an instruction and will throw an `AssertionError` exception. This output usually contains details like the location of the assertion and an error message, which should preferably contain the values of all variables occurring in the asserted expression.

In our language, we can easily add a new instruction with the desired effect:

Definition 12 (Assertion)
We extend the set of instructions $\mathbb{Instr}$ by `assert` $name, v, x_1, \ldots, x_n$ where $name$ is a unique name to identify the assertion, v a local variable containing a Boolean value, and $x_1, \ldots, x_n$ a set of variables and object attributes.

$$\mathtt{exec}^{\pi}_{\mathcal{I}}(C.m.i : \mathtt{assert}(name, v, x_1, \ldots, x_n),\ t : st, h)$$
$$= \begin{cases} (C.m.(i+1),\ t : st, h), \text{ if } \xi(t, v) = \text{TRUE}, \\ (\mathtt{Main.main.0}, (\bot, \bot, \bot, \bot), h) \text{ otherwise, which will stop the execution.} \end{cases}$$

The arguments $x_1, \ldots, x_n$ are only informational.

Note that due to our definition of the execution semantics `Exec`, we can only communicate an assertion error by returning an undefined value. A concrete implementation should output the name of the assertion and all arguments when encountering a failed assertions before terminating.

2.6 Extension to Concurrency

Our operational setting is still that of a sequential application. As one of the major sources for potential bugs is the interaction of threads in a concurrent setting, we now modify our environment to accommodate interleaved execution of threads, so that we can investigate their behaviour. We will see that on the level of events not much will change, since a trace obtained from the interleaved execution of a multi-threaded application does not differ from a trace of a sequential program, except that each event is augmented with the thread identifier of the thread currently executing the instruction. The runtime system requires some more elaborate extension, though.

A *thread* has little nonshared state. According to [110], "[an] individual thread has at least its own register state, and usually its own stack".

Here, each thread has its own program counter and control stack with local variables. The heap is globally visible to all threads. We use an interleaving execution

```
...
t := o.a

t := t+1
o.a := t
...
```

Thread 1

```
...

o.a := 5


...
```

Thread 2

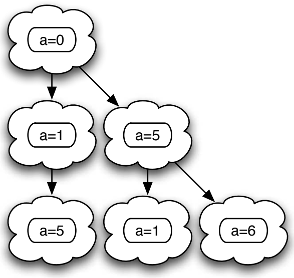


Figure 2.3: Interleaved object access

`thread` $o.m(args)$	The method invocation $o.m(args)$ is started concurrently, the return value is ignored
`lock` o	Try to lock object o, suspend if already locked
`unlock` o	Unlock object o. Threads already waiting on this object will become eligible for scheduling

Figure 2.4: Concurrency primitives

semantics where non-deterministically a thread is chosen by the scheduler and one instruction executed. This also means that the value of an attribute which is stored in a local variable in step n may no longer reflect the actual value of the attribute after step $n+1$ if another thread wrote a different value into the attribute (see Figure 2.3; liberally white-spaced to highlight the important detail). The diagram shows all possible evolutions of the object's attribute `a` through the different schedulings, and starts out with a value of 0. We conclude that for concurrency, some primitive for mutual exclusion is strongly desired.

Concurrency Primitives

A new thread is created by invoking a method call concurrently. The return value shall be ignored. An implementation might consider only allowing calls to methods return `void`. Figure 2.4 summarises the new instructions.

Apart from thread creation, we also introduce two primitives for locking and unlocking objects. Locks are *advisory* locks, that is, if a thread does not explicitly use locking, it can still interfere with a thread which has a lock on the object by modifying its attributes. Locking an already locked object will suspend the thread trying to acquire the lock, until the previous lock is released. A thread may hold more than one lock at a time. If more than one thread is waiting on a lock, after unlocking, a thread to activate will be chosen non-deterministically.

Concurrent Execution Semantics

The threaded runtime needs to implement two additional features: firstly, the mapping $\lambda_{\mathcal{T}}$ from threads to their respective stacks, and secondly, management of locks. In the following, *ts* is the *thread list*, that is, the set of all threads. *ls* is the *lock set* of a thread and contains all currently locked objects.

Definition 13 (Function space)
We introduce *function spaces* both for total and for partial functions:

$$\begin{aligned}[A \to B] &:= \{f \mid f : A \to B\} \\ [A \dashrightarrow B] &:= \{f \mid f : A \dashrightarrow B\}\end{aligned}$$

Definition 14 (Concurrent single-instruction semantics)
Let $\mathcal{T} \subseteq \mathbb{N}$ be the set of *thread identifiers*. We define the *concurrent single instruction semantics*

$$\mathtt{exec}^{\pi}_{\mathcal{I}} : \mathcal{T} \times \underbrace{[\mathcal{T} \dashrightarrow (Loc \times \overbrace{2^{\mathcal{O}}}^{ls} \times \mathcal{F}^*)]}_{\lambda_{\mathcal{T}}} \times \underbrace{2^{\mathcal{T}}}_{ts} \times \mathcal{H} \to [\mathcal{T} \dashrightarrow (Loc \times 2^{\mathcal{O}} \times \mathcal{F}^*)] \times 2^{\mathcal{T}} \times \mathcal{H}$$

as follows:

1. Lifted sequential instruction
   $\mathtt{exec}^{\pi}_{\mathcal{I}}(tId, \lambda_{\mathcal{T}}, ts, h) = (\lambda'_{\mathcal{T}}, ts, h')$,
   if for $\lambda_{\mathcal{T}}(tId) = (\alpha, ls, st)$: α is a sequential instruction, and
   $\lambda'_{\mathcal{T}}(u) := \begin{cases} (\alpha', ls, st'), \text{ where } (\alpha', st', h') = \mathtt{exec}^{\pi}_{\mathcal{I}}(\alpha, st, h) \text{ if } u = tId, \\ \lambda'_{\mathcal{T}}(u) := \lambda_{\mathcal{T}}(u) \text{ otherwise.} \end{cases}$

2. Thread creation
   $\mathtt{exec}^{\pi}_{\mathcal{I}}(tId, \lambda_{\mathcal{T}}, ts, h) = (\lambda'_{\mathcal{T}}, ts \cup \{tId'\}, h)$, if
   $\lambda_{\mathcal{T}}(tId) = (C.m.i : \mathtt{thread}\ o.m'(args), ls, t : st)$, where $D = \tau(t[o])$,
   $tId' \in \mathcal{T}$ is a fresh thread identifier, that is, $tId' \notin ts$, and
   $\lambda'_{\mathcal{T}}(u) := \begin{cases} (C.m.(i+1), ls, t : st) \text{ if } u = tId, \\ (D.m'.1, \varnothing, newStack) \text{ if } u = tId', \text{ with} \\ \quad newStack := (t[o], \mathtt{Main.main.0}, 1, locs_{D.m'} \cdot args) : (\bot, \bot, \bot, \Box), \\ \lambda_{\mathcal{T}}(u) \text{ otherwise.} \end{cases}$

   A new stack has to be initialised for the new thread like on program startup, see Definition 11.

3. Obtaining a lock
   $\mathtt{exec}^{\pi}_{\mathcal{I}}(tId, \lambda_{\mathcal{T}}, ts, h) = (\lambda'_{\mathcal{T}}, ts, h)$, if $\lambda_{\mathcal{T}}(tId) = (C.m.i : \mathtt{lock}\ o, ls, t : st)$,
   where $\lambda'_{\mathcal{T}}(u) := \begin{cases} (C.m.(i+1), ls \cup \{t[o]\}, t : st) \text{ if } u = tId, \\ \lambda_{\mathcal{T}}(u), \text{ otherwise.} \end{cases}$

4. Releasing a lock
   $\mathtt{exec}^{\pi}_{\mathcal{I}}(tId, \lambda_{\mathcal{T}}, ts, h) = (\lambda'_{\mathcal{T}}, ts, h)$, $\lambda_{\mathcal{T}}(tId) = (C.m.i : \mathtt{unlock}\ o, ls, t : st)$,
   where $\lambda'_{\mathcal{T}}(u) := \begin{cases} (C.m.(i+1), ls \setminus \{t[o]\}, t : st) \text{ if } u = tId, \\ \lambda_{\mathcal{T}}(u), \text{ otherwise.} \end{cases}$
   (The behaviour of unlocking an already unlocked object could also remain undefined.)

Note that the single-instruction semantics only implements the maintenance component of locking, that is, handling of the per-thread sets of locked objects. The actual scheduling based on whether a lock is available or not, is done in the following iteration semantics.

Definition 15 (Concurrent iteration semantics)
The scheduler non-deterministically selects a thread to run. Here, only one instruction is executed. Threads wanting to acquire a lock are not eligible to scheduling, if the respective object is already locked. To check this, the scheduler can peek into all threads' lock sets.

$$\mathtt{execI}^{\pi}_{\mathcal{I}} : [\mathcal{T} \dashrightarrow Loc \times 2^{\mathcal{O}} \times \mathcal{F}^*] \times 2^{\mathcal{T}} \times \mathcal{H} \to \mathcal{F}^*$$

$\mathtt{execI}^{\pi}_{\mathcal{I}}(\lambda_{\mathcal{T}}, ts, h) :=$

- st, iff $\lambda_{\mathcal{T}}(1) = (\alpha, ls, st)$, and $\alpha = \mathtt{Main.main.0}$
  (program termination by termination of the initial thread), or
- $\mathtt{execI}^{\pi}_{\mathcal{I}}(\mathtt{exec}^{\pi}_{\mathcal{I}}(tId, \lambda_{\mathcal{T}}, ts, h))$ otherwise, where $\lambda_{\mathcal{T}}(tId) = (\alpha, ls, t : st)$ for some $tId \in ts$ with either
  - $\alpha = C.m.i : \mathtt{lock}\ o$, and $t[o] \notin \bigcup_{i \in ts \setminus \{tId\}} \underline{proj}_2(\lambda_{\mathcal{T}}(i))$
    (object not locked by another thread), or
  - $\alpha \neq C.m.i : \mathtt{lock}\ o$ (regular instruction)

Observe that no runnable process might exist and that this function behaves non-deterministically.

Definition 16 (Concurrent program semantics)
For the *concurrent program semantics*, we now only have to set up an initial thread. Again, refer to Definition 11 to compare this to the sequential version and calculation of the initial stack.

$$\begin{aligned}&\mathtt{Exec}^{\pi}_{\mathcal{I}}(args) := \underline{proj}_4(\mathtt{execI}^{\pi}_{\mathcal{I}}(\lambda_{\mathcal{T}}, \{1\}, \{\langle \mathtt{Main}, 1\rangle\})), \text{ with}\\ &\lambda_{\mathcal{T}}(u) := (\mathtt{Main.main.1}, \varnothing, initialStack), \text{ if } u = 1.\end{aligned}$$

2.7 Obtaining a Trace Model

The execution of a program traverses a sequence of instructions of the program. Each instruction modifies the state of the program, either by modifying the stack (assignments to local variables, method call or return), or the heap (object creation, attribute assignment).

As our logic will be concerned with control flow properties (method invocation, but not, for example, iteration), and values of arguments and object attributes, the trace contains events from the dynamic control flow, that is, some action together with a valuation for any variables in the expression. Local computations involving only local variables remain invisible on the trace.

Definition 17 (Events)
An *event* is either

- τ on an invisible event, that is, function application, inter-variable assignment, or jumps inside a single method,
- $\texttt{method enter}(tId, this, Class.method, o, args)$ on method invocation,
- $\texttt{method exit}(tId, this, Class.method, o, args, x)$ on return,
- $\texttt{set}(tId, this, Class.attrib, o, x)$ on attribute access (write),
- $\texttt{get}(tId, this, Class.attrib, o, x)$ on attribute access (read),
- $\texttt{new}(tId, this, Class, o)$ on object creation,
- $\texttt{assert}(tId, this, name, args)$ on a *passed* assertion,
- $\texttt{lock}(tId, this, o)$ on obtaining a lock, or
- $\texttt{unlock}(tId, this, o)$ on releasing a lock,

where $tId \in \mathcal{T}$, $Class \in \mathcal{C}$, $method \in \mathcal{M}^{(n)}_{Class}$, $attrib \in \mathcal{A}_{Class}$, $this, o \in \mathcal{O}$, $x \in ODom$, $args \in ODom^n$. We denote the set of these events by $\textsc{Evt}$.

this is always the object performing the action, while o is the target of some action, that is, the object a method is invoked on, or the object whose attribute is accessed either for reading or writing.

Failed assertions are not made explicit in the trace, since execution stops anyway. Although the class can always be determined based on the target object o dynamically, it is encoded in the event label. Otherwise we would always need to dereference the object in the heap to obtain its class.

Also note that events do not contain any information about where it originates except from the *this* reference: neither the exact source code location nor the currently executing method are available. In Chapter 5 we will see that this very closely models the reality when instrumenting applications, as usually only arguments are available, but no easily accessible information about the context.

Execution Model

Next, we extend our execution semantics to generate those events. All arguments will either be local variables or arguments of the currently executing method and thus have their valuation available in the active stack frame.

Definition 18 (Execution trace)
The single-instruction semantics is augmented through the *trace component*:

$$\begin{aligned}\texttt{execT}^{\pi}_{\mathcal{I}}: \;& \mathcal{T} \times [\mathcal{T} \dashrightarrow Loc \times 2^{\mathcal{O}} \times \mathcal{F}^*] \times 2^{\mathcal{T}} \times \mathcal{H} \times \textsc{Evt}^* \\ & \to [\mathcal{T} \dashrightarrow Loc \times 2^{\mathcal{O}} \times \mathcal{F}^*] \times 2^{\mathcal{T}} \times \mathcal{H} \times \textsc{Evt}^*\end{aligned}$$

Let $(\alpha', ls', st') = \lambda'_{\mathcal{T}}(tId)$, with $(\lambda'_{\mathcal{T}}, ts', h') = \texttt{exec}^{\pi}_{\mathcal{I}}(tId, \lambda_{\mathcal{T}}, ts, h)$ and

$$\texttt{execT}^{\pi}_{\mathcal{I}}(tId, \lambda_{\mathcal{T}}, ts, h, \rho) := (\texttt{exec}^{\pi}_{\mathcal{I}}(tId, \lambda_{\mathcal{T}}, ts, h), (\rho; ev(tId, \alpha, st)))$$

where $(\alpha, ls, st) = \lambda_{\mathcal{T}}(tId)$ and $ev : \mathcal{T} \times (Loc \times \mathcal{F}^*) \to \textsc{Evt}$ a mapping from the dynamic semantics to an event:

1. Invisible action
   $ev(tId, (\alpha, t : st)) = \tau$,
   if instruction α involves only local variables, constants and/or functions.

2. Object instantiation
   $ev(tId, (\alpha : o \leftarrow \texttt{new}\ D, t : st))$
   $= \texttt{new}(tId, t[\texttt{this}], D, t[o])$,

3. Assignment from attribute to temporary variable
   $ev(tId, (\alpha : v \leftarrow o.a, t : st)),\ o \in \mathcal{V},\ D = \tau(t[o]),\ a \in \mathcal{A}_D,\ v \in \mathcal{V}$
   $= \texttt{get}(tId, t[\texttt{this}], D.a, t[o], \xi(t, o.a))$

4. Assignment to attribute
   $ev(tId, (\alpha : o.a \leftarrow v, t : st)),\ o \in \mathcal{V},\ D = \tau(t[o]),\ a \in \mathcal{A}_D,\ v \in \mathcal{V} \cup F^{(0)}$
   $= \texttt{set}(tId, t[\texttt{this}], D.a, t[o], \xi(t, v))$

5. Method invocation
   $ev(tId, (\alpha : v \leftarrow o.m'(args), t : st))$,
   $v, o \in \mathcal{V},\ D = \tau(t[o]),\ m' \in \mathcal{M}_D,\ x_j \in \mathcal{V} \cup F^{(0)}$
   $= \texttt{method enter}(tId, t[\texttt{this}], D.m', t[o], \bar{\xi}(t, args))$,

   where $\bar{\xi} : \mathcal{F} \times \mathcal{V}^n \to ODom^n$ is the natural extension of ξ. Note that v takes no part in the event.

6. Return from method
   $ev(tId, (C.m.i : \texttt{return}\ v, \overbrace{(this, D.m'.j, k, lvars \cdot args)}^{t} : t' : st)),\ v \in \mathcal{V} \cup F^{(0)}$
   $= \texttt{method exit}(tId, t'[\texttt{this}], C.m, t[\texttt{this}], \bar{\xi}(t, args), \xi(t, v))$
   Observe that here, $t'[\texttt{this}] \neq t[\texttt{this}] = this$.

7. Assertion
   $ev(tId, (\alpha : \texttt{assert}(name, v, args), t : st))$, $args \in (\mathcal{V} \cup F^{(0)})^n$, $v \in \mathcal{V}$
   $= \texttt{assert}(tId, t[\texttt{this}], name, \bar{\xi}(t, args))$ where *name* is some identifier

8. Lock/Unlock
   $ev(tId, (\alpha : \texttt{lock}\ o, t : st))$, $o \in \mathcal{V}$
   $= \texttt{lock}(tId, t[\texttt{this}], t[o])$ ($\texttt{unlock}$ respectively)

Definition 19 (Trace semantics of a program)
The *trace iteration semantics* is then defined along the same lines as the concurrent iteration semantics:

$$\texttt{execI}^{\pi}_{\mathcal{I}} : [\mathcal{T} \dashrightarrow Loc \times 2^{\mathcal{O}} \times \mathcal{F}^*] \times 2^{\mathcal{T}} \times \mathcal{H} \times \textsc{Evt}^* \to \textsc{Evt}^*$$

$\texttt{execI}^{\pi}_{\mathcal{I}}(\lambda_{\mathcal{T}}, ts, h, \rho) :=$

- ρ , iff $\lambda_{\mathcal{T}}(1) = (\alpha, ls, st)$, and $\alpha = \texttt{Main.main.0}$
- $\texttt{execI}^{\pi}_{\mathcal{I}}(\texttt{execT}^{\pi}_{\mathcal{I}} tId, \lambda_{\mathcal{T}}, ts, h, \rho)$ otherwise, for some $tId \in ts$ under the same condition for blocked threads as in Definition 15.

For the *trace program semantics*, we obtain:

$\texttt{ExecT}^{\mathcal{I}}_{\pi} : Dom^n \to \textsc{Evt}^*$

$\texttt{ExecT}^{\mathcal{I}}_{\pi}(args) := \texttt{execI}^{\pi}_{\mathcal{I}}(\lambda_{\mathcal{T}}, \{\mathit{1}\}, \{\langle \texttt{Main}, \mathit{1} \rangle\}, \epsilon)$, with

$\lambda_{\mathcal{T}}(tId) := (\texttt{Main.main.1}, \varnothing, initialStack)$, if $tId = 1$, $n = |IVars_{\texttt{Main.main}}|$,
and the initially empty trace ϵ.

Again, note that both semantics are non-deterministic due to multiple threads.

Summary

We conclude the discussion of our simple object-based language. We have introduced concepts of method invocation and concurrency. For the trace model, each instruction together with the dynamic information about the current thread, object, and eventual arguments or return values are accumulated. Concurrent execution is resolved in an interleaving manner, that is, the scheduler non-deterministically repeatedly selects a thread which is allowed to execute a single instruction.

3 Parametrised LTL Formulae

In the following we start with an introduction to the Linear Time Logic LTL. After giving a motivation for parametrised propositions based on a real-world example, the logic is then extended to handle dynamic bindings for parameters and filtering expressions that quantify over the current state.

As Temporal Assertions are concerned with a single execution path, we do not need the more general approach of branching time logic which is often used in the context of Model Checking. Also, we will look into some subtle differences between the usual notion of checking infinite paths in finite models and the approach we chose here.

In this section we introduce a finite-path variant of LTL. This semantics has two distinct features: on the one hand, we handle *finite*, non-empty paths. On the other hand, for practical purposes, we will consider the so-called *next-free subset of LTL*, where we only permit formulae using the temporal operators Release and Until (and the derived operators Globally and Finally), but not the explicit Next operator.

The automaton construction we subsequently provide will accept exactly the language of paths that satisfy the respective formula. It requires that any formula is *normalized* first into an equivalent representation, where negation has been pushed down to the propositional level, that is, the “leaves” of a formula.

While negation can be pushed through the Release and Until operator and to the quantifiers we will introduce, following some basic principle, this solution is not sound for the Next operator, as we will illustrate with Example 27 after introducing the temporal operators. Thus, we do not consider formulae with an explicit Next-operator.

In practice, we have not found this to be a limitation: Next is often not used explicitly and thus this is no big limitation. In fact, our examples like the Lock-order Reversal in Section 4.2 do not require the Next operator. The EAGLE logic (see Section 6.1) permits a minimal and a maximal interpretation of temporal operators at the end of the path, somewhat alleviating the situation.

3.1 LTL

Linear-time temporal logic LTL [104] extends propositional logic over a set of *atomic propositions AP* with operators which describe events along a computation path.

$$
\begin{array}{rcl}
LTL & ::= & \mathbf{tt} \mid \mathbf{ff} \mid p \in AP \\
& \mid & \mathbf{F}\ LTL \\
& \mid & \mathbf{G}\ LTL \\
& \mid & LTL\ \mathbf{U}\ LTL \\
& \mid & LTL\ \mathbf{R}\ LTL \\
& \mid & LTL\ \vee\ LTL \\
& \mid & LTL\ \wedge\ LTL \\
& \mid & \neg LTL
\end{array}
$$

Figure 3.1: LTL syntax

Finite Path Semantics for LTL

Figure 3.1 gives the syntax of next-free LTL formulae. The temporal operators have the following meaning for formulae φ, ψ:

- "Eventually" ($\mathbf{F}\ \varphi$): φ will hold at some state now or in the future (also: "in the future","finally")
- "Globally" ($\mathbf{G}\ \varphi$): at every state on the path ϕ holds
- "Until" ($\varphi\ \mathbf{U}\ \psi$): combines two properties in the sense that φ has to hold until finally ψ holds.
- "Release" ($\varphi\ \mathbf{R}\ \psi$): dual of $\mathbf{U}$; expresses that the second property holds along the path up to and including the first state where the first property holds, although the first property is not required to hold eventually.

Definition 20 (Finite path semantics of LTL)
Let AP be a set of atomic propositions and $w = w[0]...w[n-1] \in (2^{AP})^n$ for $n > 0$ a finite path. For each path position $w[j]$ $(0 \leq j < n)$, a proposition $p \in AP$, and formulae φ, ψ:

$$
\begin{array}{llll}
w[j] & \models \mathbf{tt}, & w[j] \not\models \mathbf{ff}, & \\
w[j] & \models p & \text{iff} & p \in w[j] \\
 & \models \neg\varphi & \text{iff} & w[j] \not\models \varphi \\
 & \models \mathbf{F}\ \varphi & \text{iff} & \exists k\ (j \leq k < n) \text{ s.th. } w[k] \models \varphi \\
 & \models \mathbf{G}\ \varphi & \text{iff} & \forall k\ (j \leq k < n) \rightarrow w[k] \models \varphi \\
 & \models \varphi\ \mathbf{U}\ \psi & \text{iff} & \exists k\ (j \leq k < n) \text{ s.th. } w[k] \models \psi \\
 & & & \quad \wedge\ \forall l\ (j \leq l < k) \rightarrow w[l] \models \varphi \\
 & \models \varphi\ \mathbf{R}\ \psi & \text{iff} & \forall k\ (j \leq k < n) \rightarrow w[k] \models \psi \\
 & & & \quad \vee\ \exists l\ (j \leq l < k) \text{ s.th. } w[l] \models \varphi \\
 & \models \varphi \oplus \psi & \text{iff} & w[j] \models \varphi \oplus w[j] \models \psi, \oplus \in \{\vee, \wedge\}
\end{array}
$$

We write $w \models \varphi$ iff $w[0] \models \varphi$.

Definition 21 (LTL-induced language)
The language of paths accepted by an LTL formula φ is defined by

$$L_\varphi := \{w \mid w \models \varphi, w \in (2^{AP})^+\}.$$

Definition 22 (Equivalence of LTL formulae)
Equivalence of formulae is defined through equality of the set of accepted paths, that is, the respective languages:

$$\varphi \equiv \psi :\Longleftrightarrow L_\varphi = L_\psi.$$

Theorem 23 (Relation between temporal operators)
The following well-known equivalences to express *Finally* and *Globally* hold:

$$\mathbf{F}\ \varphi \equiv \mathbf{tt}\ \mathbf{U}\ \varphi \qquad \mathbf{G}\ \varphi \equiv \mathbf{ff}\ \mathbf{R}\ \varphi$$

As an example we consider the formula $p\ \mathbf{U}\ q$ over propositions $AP := \{p, q\}$. It is valid on the path $\{p\}\{p\}\{q\}$ but it is neither valid on the path $\{p\}\{p\}\varnothing\{q\}$ nor on the path $\{p\}\{p\}\{p\}$

Remark 24 (Finite path semantics)
Regarding the end of a path we observe the following behaviour: due to the quantification in the $\models$ relation for a Release formula, such a formula is satisfied if ψ holds on the last state. Conversely, since the Until formula depends on existential quantification over positions in the trace, it fails if ψ does not hold.

Definition 25 (Positive form)
An LTL formula is converted to *positive form* by pushing down negations by repeatedly applying the following rewriting rules (see [93]):

$$\begin{array}{lcl}
\neg\neg\varphi & \longrightarrow & \varphi \\
\neg(\varphi \vee \psi) & \longrightarrow & \neg\varphi \wedge \neg\psi \\
\neg(\varphi \wedge \psi) & \longrightarrow & \neg\varphi \vee \neg\psi \\
\neg(\mathbf{G}\ \varphi) & \longrightarrow & \mathbf{F}\ (\neg\varphi) \\
\neg(\mathbf{F}\ \varphi) & \longrightarrow & \mathbf{G}\ (\neg\varphi) \\
\neg(\varphi\ \mathbf{U}\ \psi) & \longrightarrow & (\neg\varphi)\ \mathbf{R}\ (\neg\psi) \\
\neg(\varphi\ \mathbf{R}\ \psi) & \longrightarrow & (\neg\varphi)\ \mathbf{U}\ (\neg\psi)
\end{array}$$

In the following, we assume that φ^+ refers to the positive form for an arbitrary LTL formula φ if this distinction is required.

Lemma 26 (Equivalence of positive form)
For all $\varphi \in LTL$ we have $\varphi \equiv \varphi^+$ (without proof).

Remark 27 (Soundness of positive form of formulae with Next operator)
For certain paths, the semantics of LTL formulae containing the Next operator is not preserved with regard to normalisation to positive form where negation is pushed down to the propositional level. Assume the usual notion of the finite path semantics of the Next operator for a path w of length n:

$$w[j] \models \mathbf{X}\ \varphi \quad \text{iff} \quad j < n-1 \text{ and } w[j+1] \models \varphi$$

When we consider a path w with $|w| = 1$ and some formula φ, we observe the following behaviour:

$$\begin{aligned} w \models \neg\mathbf{X}\ \varphi &\equiv w \not\models \mathbf{X}\ \varphi \\ &\equiv w \not\models \mathbf{ff} \\ &\equiv \mathbf{tt} \end{aligned}$$

Yet on the other hand, for the normal form, we apply the following rewriting rule:

$$\begin{aligned} w \models \neg\mathbf{X}\ \varphi &\longrightarrow w \models \mathbf{X}\ \neg\varphi \\ &\equiv \mathbf{ff} \end{aligned}$$

Remark 28 (Complexity of LTL model checking on infinite words)
LTL formulae model star-free ω-regular properties [119]. The space and time complexity of LTL model checking is linear in the size of the model, but exponential in the size of the formula (PSPACE-hard) [111].

3.2 Extension to Parametrised Propositions and their Semantics

Already in the introduction to Chapter 2 we gave examples where some property should hold *for each instance* of data structure (iterators and stacks), that is, objects of a specific type. We complete the motivation for parametrised propositions by the following example: for a software product, the specification might require that every opened file is eventually closed.

The number of files a program opens or closes can be dynamic. Since references to files may be stored in variables, any static verification would have already to deal with aliasing. Dynamic data-structures further complicate the picture, as we have seen previously. The program could, for example, contain an array of file handles and access the entries based on offset or pointer arithmetic. Without data structures, that is, where file handles are only referenced through variables, the problem could be solved statically by showing that every program path that opens a file passes through its corresponding closing instruction with respect to variable aliasing.

To further illustrate this point, we show some source code from the Apache web server (`http://httpd.apache.org`). The web server supports several backends to distribute incoming requests to a set of *processes* which in turn may pass it to one of several *threads*. A configurable amount of processes/threads can be initialised at startup, with a maximum cap on additionally created processes/threads at runtime. On UNIX systems, threads in the Apache library are implemented in an abstraction layer on top of `pthreads` (see Section 5.3 for details on the instrumentation). Source code used in examples refers to version 2.0.49 of the original Apache software distribution.

Since the limits for the amount of concurrency are configurable, data pertaining to concurrency control is stored in dynamic data structures. For example, in the `worker` module, processes and threads are stored in arrays, which are dynamically allocated. Listing 3.1 gives parts of the source code, where `start_threads` shows how the thread identifiers are stored in an array, and `startup_children` invokes via `make_child` the system call `fork()` to generate processes, whose process identifiers are likewise stored in arrays (not shown), as is the dynamic storage allocation for said data. Semaphores for concurrent access are also allocated dynamically on a per-queue basis (see Listing 3.2).

Coming back to our file example, let's assume that opening or closing triggers the events `method exit open(`$name, i$`)` or `method exit close(`j`)`, with $i, j \in \mathbb{N}$ and *name* the file name. For clarity, we will omit the file name argument in the following since it does not contribute to the motivation. It can be assumed as universally quantified.

As for the file handles, for example, in the C runtime system, they are simply consecutive integers starting from zero, where the first three usually refer to the standard input, standard output, and standard error output of the application. When using the higher-level abstraction of *streams*, file references are pointers to data structures in memory, but can nevertheless be interpreted as integer values. Given a model where states contain propositions indicating which files have been opened or closed, in a first attempt, we can capture the specification as follows (we do not consider reusing the same file handle on the trace):

$$
\begin{aligned}
& \forall x : \mathbf{G}\ [\mathit{open}(x) \rightarrow \mathbf{F}\ \mathit{close}(x)] \\
\equiv\ & \forall x : \mathbf{G}\ [\neg \mathit{open}(x) \vee \mathbf{F}\ \mathit{close}(x)]
\end{aligned}
$$

To complete the picture, for the iterator and stack examples, the formulae asserting correct use might look as follows:

$$
\begin{aligned}
\varphi_1 &:= \forall c, i : \mathbf{G}\ [\mathit{makeIterator}(c, i) \rightarrow\ \mathbf{G}\ (\mathit{modify}(c) \rightarrow \mathbf{G} \neg \mathit{next}(i))] \\
\varphi_2 &:= \forall s \exists e : \mathit{push}(s, e)\ \mathbf{R}\ \neg \mathit{pop}(s)
\end{aligned}
$$

This brings us to the question of what actually the domain of a variable is, so that we can syntactically expand the formula into a clause of instantiated formulae. Again, for the C runtime system, values could stretch over the whole range of integers. But in fact for Runtime Verification, we only need to consider those values which we actually observe in a run.

```
static void * APR_THREAD_FUNC start_threads
                        (apr_thread_t *thd, void *dummy)
{
    thread_starter *ts = dummy;
    apr_thread_t **threads = ts->threads;
    ...
    rv = apr_thread_create(&threads[i], thread_attr,
                        worker_thread, my_info, pchild);
    ...
}

/* start up a bunch of children */
static void startup_children(int number_to_start)
{
    int i;

    for (i = 0; number_to_start && i < ap_daemons_limit; ++i) {
        if (ap_scoreboard_image->parent[i].pid != 0) {
            continue;
        }
        if (make_child(ap_server_conf, i) < 0) {
            break;
        }
        --number_to_start;
    }
}
```

Listing 3.1: httpd-2.0.49/server/mpm/worker/worker.c

```
/**
 * Initialize the fd_queue_t.
 */
apr_status_t ap_queue_init(fd_queue_t *queue,
                        int queue_capacity, apr_pool_t *a)
{
    int i;
    apr_status_t rv;
    char *lbl;

    if ((rv = apr_thread_mutex_create(&queue->one_big_mutex,
              APR_THREAD_MUTEX_DEFAULT, a)) != APR_SUCCESS) {
        return rv;
    }
```

Listing 3.2: httpd-2.0.49/server/mpm/worker/fdqueue.c

3.2.1 Variables and their Domains

From a purely mathematical logic point of view, the domain of a variable needs to be clearly defined. Then, formulae using quantification can be expanded to conjunctions and disjunctions where the variables have been instantiated with concrete values. This holds in the modal setting if we consider that, for example, all possible file handles are known in advance.

For practical means, it would be very inefficient even for the formulae above for file access, with only one variable corresponding to the file handle, to instantiate this formula for all possible values in 2^{32}, or, for more recent processors even 2^{64}, as the domain of quantification. The same holds for object references which will usually be pointers to memory locations, thus in the same range.

What could we do to limit the domain to the subset of values actually necessary to validate or invalidate a formula on a run? For programs, the objects we want to reason about do not appear out of the blue sky. In object-oriented languages, objects are explicitly created. Even for C programs, there is usually some initialisation. In the above case, this would be the `open` event which makes the object "interesting" for us.

This also has the advantage that we do not have to know the entire domain beforehand, but can rather rely on quantification just working on the *current state*. Another possibility would be making quantification work over all *known* objects up until now. After all, for example in Java, we could iterate over the objects in the heap. Of course, this will lead to similar scalability problems like above for large programs. To efficiently solve the problem of quantification in our Runtime Verification approach, we need quantification to be as concise (in the sense that we pick up just the valuations interesting to us and not more) as possible.

As change in a program is driven by statements, which in turn can be considered as events or propositions, we shall associate quantifiers and their variables directly to a parametrised proposition in the formula.

Instead of using some previously known domain, we will observe the program (the trace), for *events* which will indicate that an object "has become interesting" for us, that is, that it should be subject to an instantiation of a formula. Although all our use cases are in fact from event-based systems, our formal framework permits states with more than one proposition/event per state.

In the example above, the event making a file handle "interesting" for us can be clearly identified: although many other unused file handles may exist, we want to observe this property for *actually opened* files. Thus we wish to reformulate the property in the way the we only need to contemplate the trace from state to state. This is easily accomplished through the formula

$$\mathbf{G}\ [\forall x : \neg open(x) \vee \mathbf{F}\ close(x)].$$

Unfortunately, this formula also has no explicit constructive proposition which would tell us only about the currently opened file and still requires us to know all possible files: it only explicitly mentions files *not being opened* and files which get closed.

An ideal framework would allow us to select only pertinent values from the current state and assert some property which should hold on the rest of the trace. Thus, we need to make the event which binds the value explicit. This goes hand in hand with changing the scope of the quantifier, which now only needs to reason about the current state:

$$\mathbf{G}\ [\forall x : open(x) \land \mathbf{F}\ close(x)]$$

This also already hints at the fact that there exists a special relationship between the quantified proposition which *binds* the value and the remainder of the formula which only *uses* it. We will see that this is solved by introducing a new (non-commutative) operator which ties together both sides.

State-based Quantification

One of the foundations of our framework is that *each instantiation of a variable must be a consequence of an observed state*. That is, *the domain of a variable is only defined through the trace*. This has thorough implications for the handling of negated propositions containing unbound variables. In fact, we will limit negation to ground propositions in our framework, but also show that negation can be pushed through, thus not imposing any limitation on the user (see Theorem 46).

In first-order temporal logic, the quantifiers also give us yet another degree of freedom: quantification may also be used to *enumerate entities present in the current state only* through an *explicit existence predicate*. This view has been put forth in [122]. We will use this approach by splitting formulae in two parts: one part contains the (positive) existence predicate with the quantifiers for variables occurring free in the parametrised proposition. The second part is a formula where those variables may only be referenced, but no new valuations for the quantified variables may be induced. In a state where a proposition p does not hold for any valuation, universal quantification will short-cut evaluation to $\mathbf{tt}$, while existential quantification will result in $\mathbf{ff}$.

State-based quantification thus means that quantified variables inside temporal operators will have potentially disjoint domains. Without presuming too much about the actual semantics of our extension to LTL that we will define shortly, we shall consider a small example to illustrate this point. Although the two propositions in the following formula use the same constructor p, the domains for the variables x and y are distinct on the trace:

$$\begin{aligned} w &:= \{p(1)\}\ \{p(2)\} \\ \varphi &:= (\forall x : p(x)) \land \mathbf{G}\ (\exists y : p(y)) \end{aligned}$$

If we consider both states separately, we clearly see that in each state, there is exactly one valuation for the variables: $x/1, y/1$ in state $w[0]$ and $x/2, y/2$ in $w[1]$. As evaluation of temporal formula proceeds along the trace, a possible approach is to look at a formula and generate the corresponding instances of quantified formulae, in this case based on the initial state:

$$\varphi \equiv p(1) \wedge \mathbf{G}\ p(1)$$

However, this would neglect the fact that the existential quantifier is contained *within* the temporal operator. We might as well consider all possible valuations for a variable looking at the entire path: in the first step, we accumulate all propositions occurring on the path, obtaining $\{p(1), p(2)\}$. Then, under the same consideration as above, the instantiation of the quantified variables with respect to this step yields:

$$\varphi \equiv (p(1) \wedge p(2)) \wedge \mathbf{G}\ (p(1) \vee p(2))$$

This approach has two drawbacks: on the one hand, it requires collecting the entire path before starting evaluation, which, in the scope of Runtime Verification, is problematic as we want to detect possible misbehaviour of an application with regard to some temporal specification immediately, and we might also want to check continuously running programs. On the other hand, during evaluation, it is hard to find a suitable meaning for a temporal formula with instantiations of values that have not occurred yet and will only manifest themselves in future computations.

Instead of the previous all-or-nothing approach, there is also another alternative to treat quantifiers within temporal operators. Just as the finite path semantics for the Globally operator refers to the contained subformula, which in this case contains the quantification, we see that the possible substitutions are applied repeatedly on subsequent states, that is, using this approach we obtain:

$$\begin{aligned} w \models \varphi \iff & \quad w[0] \models p(1) \wedge \mathbf{G}\ p(1) \\ & \wedge \ \ w[1] \models \mathbf{G}\ p(2) \end{aligned}$$

As a last remark, let us point out that the result of verifying a quantified formula against a path yields a **tt**/**ff** result with a single valuation for the state which (in)validated the formula. On each state of the trace however, potentially *different valuations* have been used if quantification occurs inside temporal operators. For example, the formula

$$(\exists x : p(x))\ \mathbf{U}\ q$$

requires at least one proposition $p^{(1)}$ to hold in each state up to the state where the right-hand side q holds. If this formula fails or succeeds, there may be *no value for x* available. Previous valuations for x may have been recorded in the *trace*, but are not accessible to any posterior state due to inner quantification (also see Example 45).

Next, we will define the underlying mechanisms of our parametrised framework and then illustrate the provided functionality with examples.

3.2.2 Parametrised Propositions

In the extended framework, *a proposition occurring in a formula* consists of a constructor with a given arity and the corresponding number of variables as arguments.

In a *state*, the arguments to the constructor are values from some fixed object domain. A state defines a mapping of a parametrised proposition to a set of currently defined valuations. When matching against a parametrised proposition which contains quantified and already bound variables, each unbound variable in the proposition may get a value from the underlying domain. This mechanism is closely related to unification in the Prolog system [114], although we only handle constants, and not arbitrary terms.

Syntactically, we will enforce by construction that a quantifier is tied to the proposition that *binds* valuations. In the remainder of the formula, the variable is only *used* but does not generate any new bindings.

Definition 29 (Parametrised proposition, Ground proposition)
Let $\mathcal{PN}$ be a set of *proposition names*, where $p \in \mathcal{PN}^{(n)}$ denotes a constructor of arity $n \in \mathbb{N}$. Then, the set of *all propositions* $\mathcal{P}$ given a fixed value domain $\mathcal{D}$ and a set of variables $\mathcal{V}$ is defined as:

$$\mathcal{P} := \bigcup_{n\in\mathbb{N}} \bigcup_{p\in\mathcal{PN}^{(n)}} \{p(v_1, \dots, v_n) \mid v_k \in \mathcal{D} \cup \mathcal{V}, 1 \leq k \leq n\}$$

The set of *ground propositions* $\mathcal{P}_\perp$ is the subset of all propositions where each position is instantiated with an element of $\mathcal{D}$, that is, no position contains a variable:

$$\mathcal{P}_\perp := \{p(d_1, \dots, d_n) \in \mathcal{P} \mid \forall i : d_i \in \mathcal{D}, n \in \mathbb{N}\}$$

The operators of LTL formulae in the extended framework are virtually the same as in the previous section. But since propositions may now contain variables, we must introduce quantification. Moreover, we will permit quantifiers also inside subformulae shadowed by temporal operators. We will limit ourselves to *sentences*, where every variable is quantified. Syntactically, we restrict formulae to the form where quantifiers occur only together with a *positive* proposition.

The introduction of a special operator makes it easier for us to specify and enforce semantic constraints already on a syntactical level: a quantified *existence predicate* (selector) may be the left-hand side of a special type of non-commutative implication denoted by $\dot{\rightarrow}$, where the right-hand side is another temporal formula that refers to the bound values. The following definition of pLTL formulae adheres to the syntax definition of LTL (see Figure 3.1), where we replace the atomic propositions by binding expressions.

Definition 30 (Existence predicate, Selector, pLTL formula)
The set pLTL of parametrised LTL formulae over a set of variables $\mathcal{V}$ and a value domain $\mathcal{D}$ is defined by the following attribute grammar where quantified variables are propagated top-down:

$$
\begin{array}{rcl}
p\text{LTL} & ::= & LTL(\varnothing) \\
LTL(V \subset \mathcal{V}) & ::= & \mathbf{tt} \mid \mathbf{ff} \\
& \mid & p(u_1, \ldots, u_n) \in \mathcal{P}^{(n)},\ u_1, \ldots, u_n \in V \cup \mathcal{D} \\
& \mid & Q_1 x_1 \ldots Q_m x_m : p(u_1, \ldots, u_n) \dot{\rightarrow} LTL(V'), \\
& & \ \forall i : x_i \in \mathcal{V},\ \exists j : x_i = u_j,\ Q_i \in \{\forall, \exists\}, \\
& & \ V' := V \uplus \{x_1, \ldots, x_m\},\ \forall k : u_k \in V' \cup \mathcal{D} \\
& \mid & \mathbf{F}\ LTL(V) \\
& \mid & \mathbf{G}\ LTL(V) \\
& \mid & LTL(V)\ \mathbf{U}\ LTL(V) \\
& \mid & LTL(V)\ \mathbf{R}\ LTL(V) \\
& \mid & LTL(V)\ \vee\ LTL(V) \\
& \mid & LTL(V)\ \wedge\ LTL(V) \\
& \mid & \neg LTL(V)
\end{array}
$$

For clarity, we require that quantifiers always use fresh variables, that is, variables which are not yet contained in V. We use $LTL(V)$ for some $V \subset \mathcal{V}$ to denote the set of pLTL formulae with free variables V. Also, examples may omit the implication and the right-hand side in the leaf of a formula if it implies $\mathbf{tt}$. Note that each quantified variable *must occur at least once* in the existence predicate to avoid vacuous quantification.

Remark 31 (Existence predicate)
A formula $Q_1 x_1 \ldots Q_m x_m : p(u_1, \ldots, u_n) \dot{\rightarrow} \psi$ can be understood as:

> *"There exists a set of valuations derived from the current state (depending on the quantifiers) satisfying both $p(u_1, \ldots, u_n)$ and ψ with the bound values.*
>
> *Otherwise the outermost quantifier indicates whether evaluation should succeed (universal quantification over the empty domain) or fail (existential quantification)."*

The formal definition will be given below. We define the finite paths semantics for a pLTL formula over a set of variables with respect to a given *valuation (binding)* $\beta : \mathcal{V} \dashrightarrow \mathcal{D}$ and a path $w = w[0] \ldots w[n-1] \in (2^{\mathcal{P}_\perp})^n$.

Definition 32 (Valuation function, Binding)
Let $\hat{\beta}$ be the natural extension of $\beta : U \to \mathcal{D}$ over pLTL propositions and formulae $\varphi \in LTL(V)$, $U, V \subset \mathcal{V}$, $bound(\varphi) \cap U = \varnothing$:

$$\hat{\beta} : LTL(V) \to LTL(V)$$
$$\hat{\beta}(p(v_1, \ldots, v_m)) := p(u_1, \ldots, u_m), \text{ where}$$
$$u_i := \begin{cases} \beta(v_i), & \text{if } v_i \in \mathcal{V} \text{ and } \beta(v_i) \text{ defined} \\ v_i, & \text{otherwise} \end{cases}$$

$$\hat{\beta}(\mathbf{tt}) := \mathbf{tt},\ \hat{\beta}(\mathbf{ff}) = \mathbf{ff}$$
$$\hat{\beta}(\neg\varphi) := \neg\hat{\beta}(\varphi)$$
$$\hat{\beta}(Q_1 x_1 \ldots Q_m x_m : p(u_1, \ldots, u_n) \dot{\to} \psi) := Q_1 x_1 \ldots Q_m x_m : \hat{\beta}(p(u_1, \ldots, u_n)) \dot{\to} \hat{\beta}(\psi)$$
$$\hat{\beta}(\mathbf{F}\ \varphi) := \mathbf{F}\ \hat{\beta}(\varphi),\ \hat{\beta}(\mathbf{G}\ \varphi) := \mathbf{G}\ \hat{\beta}(\varphi)$$
$$\hat{\beta}(\varphi \oplus \psi) := \hat{\beta}(\varphi) \oplus \hat{\beta}(\psi),\ \oplus \in \{\mathbf{R}, \mathbf{U}, \wedge, \vee\} \text{ (binary operators)}$$

We use $\beta_\varnothing$ to denote the empty valuation which does not assign a value to any variable. When representing valuations in the text, we will use a set of tuples consisting of variable/value pairs, for example, $\{x/3, y/4\}$, resembling the function that maps x to 3 and y to 4.

Definition 33 (Specialisation of valuations)
The $\odot$ operator *specialises a valuation*. We will later only use it in an environment where the two bindings are disjoint and thus one binding *extends* the previous one.

$$\begin{array}{rcl} \odot & : & (\mathcal{V} \dashrightarrow \mathcal{D}) \times (\mathcal{V} \dashrightarrow \mathcal{D}) \to (\mathcal{V} \dashrightarrow \mathcal{D}) \\ \beta_2 \odot \beta_1 & := & \lambda x. \begin{cases} \beta_1(x), & \text{if } x \in \mathcal{V} \text{ bound in } \beta_1 \\ \beta_2(x), & \text{otherwise} \end{cases} \end{array}$$

Corollary 34 (Composition of valuations)
It is easy to see that $\hat{\gamma} \circ \hat{\beta} \equiv (\widehat{\gamma \odot \beta})$. (Without proof.)

Definition 35 (Unifiable parametrised propositions)
Two parametrised propositions with the same proposition name are called *unifiable*, iff there exist substitutions for all variables occurring in them such that both propositions are identical under these substitutions:

$$\downarrow : \mathcal{P} \times \mathcal{P} \to \mathbb{B}$$

$$p(u_1, \ldots, u_n) \downarrow p(v_1, \ldots, v_n)\ (u_i, v_i \in \mathcal{V} \cup \mathcal{D}, 1 \le i \le n) :\Longleftrightarrow$$
$$\exists \sigma_1, \sigma_2 : \mathcal{V} \dashrightarrow \mathcal{D} \text{ such that } \hat{\sigma}_1(p(u_1, \ldots, u_n)) = \hat{\sigma}_2(p(v_1, \ldots, v_n))$$

Definition 36 (Valuations of a parametrised proposition)
Given a (partially instantiated) parametrised proposition $p(u_1, \ldots, u_n) \in \mathcal{P}$ and a state $a \in 2^{\mathcal{P}_\perp}$, we obtain all *possible valuations* for a variable $x \in \{u_1, \ldots, u_n\}$:

$$\begin{array}{rcl} vals : \mathcal{P} \times 2^{\mathcal{P}_\perp} \times \mathcal{V} \to 2^{\mathcal{D}} & & \\ vals(p(u_1, \ldots, u_n), a, x) & := & \{d \in \mathcal{D} \mid \exists p(d_1, \ldots, d_n) \in a : \\ & & (\widehat{\{x/d\}})(p(u_1, \ldots, u_n)) \downarrow p(d_1, \ldots, d_n)\} \end{array}$$

Definition 37 (Extended finite path semantics)
We define the *extended finite path satisfaction relation* $(w[j], \beta) \models \varphi$ for a non-empty path $w = w[0] \dots w[n-1] \in (2^{\mathcal{P}_\perp})^n$, $0 \leq j < n$, $\beta : \mathcal{V} \dashrightarrow \mathcal{D}$, and a formula $\varphi \in LTL(V)$, $V \subset \mathcal{V}$ by induction on the structure of φ.

First, we define the model satisfaction relation where no existence predicates occur. The only difference to the plain LTL semantics (Definition 20) is that the already accumulated valuation β is threaded through:

$$
\begin{array}{llll}
(w[j], \beta) & \models \mathbf{tt}, \quad (w[j], \beta) & \not\models \mathbf{ff}, & \\
 & \models p(u_1, \dots, u_m) & \text{iff} & \hat{\beta}(p(u_1, \dots, u_m)) \in w[j], \\
 & \models \neg\varphi & \text{iff} & w[j] \not\models \varphi \\
 & \models \mathbf{F}\ \varphi & \text{iff} & \exists k\ (j \leq k < n) \text{ s.th. } (w[k], \beta) \models \varphi \\
 & \models \mathbf{G}\ \varphi & \text{iff} & \forall k\ (j \leq k < n) \rightarrow (w[k], \beta) \models \varphi \\
 & \models \varphi\ \mathbf{U}\ \psi & \text{iff} & \exists k\ (j \leq k < n) \text{ s.th. } (w[k], \beta) \models \psi \\
 & & & \qquad \wedge\ \forall l\ (j \leq l < k) \rightarrow (w[l], \beta) \models \varphi \\
 & \models \varphi\ \mathbf{R}\ \psi & \text{iff} & \forall k\ (j \leq k < n) \rightarrow (w[k], \beta) \models \psi \\
 & & & \qquad \vee\ \exists l\ (j \leq l < k) \text{ s.th. } (w[l], \beta) \models \varphi \\
 & \models \varphi \oplus \psi & \text{iff} & (w[j], \beta) \models \varphi \oplus (w[j], \beta) \models \psi, \oplus \in \{\vee, \wedge\}
\end{array}
$$

In the presence of existence predicates with quantifiers, we first derive the set of all possible valuations for each variable and are obliged to prove the remaining formula with respect to these bindings. We need to be aware that quantification over an empty set D' results in either **ff** or **tt**, depending on the quantifier. This is achieved through the disjunctive normal form (DNF), where we calculate the *set of sets of valuations*, where for at least one item in the outer set, all valuations of the inner set must satisfy the formula on the remaining path:

$$
\begin{array}{l}
(w[j], \beta) \models Q_1 x_1 \dots Q_k x_k : p(u_1, \dots, u_m) \dot{\rightarrow} \psi, \\
\text{iff } \exists \theta \in \Theta \text{ such that } \forall \sigma \in \theta : (w[j], \sigma \odot \beta) \models \psi
\end{array}
$$

which we can reformulate to

$$
\bigvee_{\theta \in \Theta} \bigwedge_{\sigma \in \theta} (w[j], \sigma \odot \beta) \models \psi
$$

where

$$
\begin{array}{rcl}
\Theta & := & \mathit{valid}(\mathit{vals}_{x_1} \otimes_1 (\ldots(\mathit{vals}_{x_k} \;\otimes_k\; \{\{\beta_\varnothing\}\})\ldots)), \\
& & \text{or } \{\{\beta_\varnothing\}\} \text{ if } k = 0 \text{ (no quantifiers)} \\
\mathit{vals}_{x_i} & := & \mathit{vals}(\hat{\beta}(p(u_1,\ldots,u_m)), w[j], x_i) \\
\otimes_i & : & 2^{\mathcal{D}} \times 2^{2^{\mathcal{V} \dashrightarrow \mathcal{D}}} \to 2^{2^{\mathcal{V} \dashrightarrow \mathcal{D}}} \\
D \otimes_i \Theta' & := & \begin{cases} \displaystyle\bigcup_{\theta' \in \Theta'} \left\{ \bigcup_{\sigma' \in \theta'} \{\ \{x_i/d\} \odot \sigma'\} \middle| d \in D \right\}, \text{if } Q_i = \exists \\ \displaystyle\bigotimes_{d \in D} \{\{\{\ \{x_i/d\} \odot \sigma' \mid \sigma' \in \theta'\} | \, \theta' \in \Theta'\}\}, \text{if } Q_i = \forall \end{cases}
\end{array}
$$

$$
\begin{array}{l}
\text{with } \bigotimes\{\Theta_1,\ldots,\Theta_n\} := \Theta_1 \otimes \ldots \otimes \Theta_n, \text{ where} \\
S \otimes T := \{s \cup t \mid s \in S,\ t \in T\} \\
\text{and } \mathit{valid}(\Omega) := \left\{ \{\sigma \mid \sigma \in \theta, (\widehat{\sigma \odot \beta})(p(d_1,\ldots,d_m)) \in w[j]\} \middle| \theta \in \Omega \right\}
\end{array}
$$

The above calculation is thus the extension of the *vals* function (Definition 36) to multiple quantifiers and shall be referred to in the future as the function

$$\mathit{spec} : (\mathcal{V} \dashrightarrow \mathcal{D}) \times 2^{\mathcal{P}_\perp} \times \mathit{LTL}(V) \to 2^{2^{\mathcal{V} \dashrightarrow \mathcal{D}}},$$

that is, here

$$\Theta = \mathit{spec}(\beta, w[j], Q_1 x_1 \ldots Q_k x_k : p(u_1,\ldots,u_m)).$$

Observe that although β is passed in to the function, the resulting valuations are calculated modulo the pre-existing binding. Old and new bindings are composed through the expression $\sigma \odot \beta$ in the consumer. The right-hand side of the existence-predicate passed into the evaluation of *spec* takes no part in the result and is therefore omitted where convenient.

We define two shorthands for convenience. Like for plain LTL, for a given path w we consult the above semantics starting at the first item in the path:

$$(w, \beta) \models \varphi :\Longleftrightarrow\ (w[0], \beta) \models \varphi.$$

A path is a model for a pLTL formula, if the formula can be proved for the initially empty valuation $\beta_\varnothing$:

$$w \models \varphi :\Longleftrightarrow\ (w, \beta_\varnothing) \models \varphi.$$

$\hat{\beta}_\varnothing$ is thus the identity function id.

Theorem 38
For all non-empty paths $w \in (2^{\mathcal{P}_\perp})^+$, $\varphi \in \mathit{LTL}(V)$ in positive form, $V \subset \mathcal{V}$, and valuations $\beta : \mathcal{V} \dashrightarrow \mathcal{D}$ it holds that:

$$(w, \beta) \models \varphi \iff (w, \beta_\varnothing) \models \hat{\beta}(\varphi)$$

Proof:

This property clearly holds on the propositional level. For the Boolean and temporal operators, evaluation just proceeds into the respective branches. Next, we look at the proof for the case where a valuation is actually applied to a proposition with an existence predicate. For the left-hand side with $\vec{u} = u_1, \ldots, u_m$, we obtain:

$$\begin{array}{rl} & (w,\beta) \models Q_1x_1 \ldots Q_kx_k : p(\vec{u}) \dot{\rightarrow} \psi \\ \equiv & (w[0],\beta) \models Q_1x_1 \ldots Q_kx_k : p(\vec{u}) \dot{\rightarrow} \psi \\ \equiv & \exists \theta \in \Theta \text{ s.th. } \forall \sigma \in \theta : (w[0], \sigma \odot \beta) \models \psi \\ & \text{with } \Theta := \mathit{spec}(\beta, w[0], Q_1x_1 \ldots Q_kx_k : p(\vec{u})) \end{array}$$

Looking at the right-hand side of the theorem according to the definition:

$$\begin{array}{rl} & (w,\beta_\varnothing) \models \hat{\beta}(Q_1x_1 \ldots Q_kx_k : p(\vec{u}) \dot{\rightarrow} \psi) \\ \equiv & (w[0],\beta_\varnothing) \models \hat{\beta}(Q_1x_1 \ldots Q_kx_k : p(\vec{u}) \dot{\rightarrow} \psi) \\ \equiv & \exists \theta \in \Theta' \text{ s.th. } \forall \sigma \in \theta : (w[0], \sigma \odot \beta_\varnothing) \models \hat{\beta}(\psi), \\ & \text{with } \Theta' := \mathit{spec}(\beta_\varnothing, w[0], Q_1x_1 \ldots Q_kx_k : \hat{\beta}(p(\vec{u}))) \end{array}$$

As $\beta_\varnothing$ is the empty valuation function, it holds that

$$\equiv \quad \exists \theta \in \Theta' \text{ s.th. } \forall \sigma \in \theta : (w[0], \sigma) \models \hat{\beta}(\psi)$$

Since by the induction above $(w[0], \sigma \odot \beta) \models \psi \equiv (w[0], \beta_\varnothing) \models \widehat{(\sigma \odot \beta)}(\psi) \equiv (w[0], \sigma) \models \hat{\beta}(\psi)$ (ψ will eventually be a leaf of a formula), it remains to be shown that $\Theta = \Theta'$. For both, the derivations are obtained with respect to the same state and the same set of free variables. For $\Theta := \mathit{spec}(\beta, w[0], Q_1x_1 \ldots Q_kx_k : p(\vec{u}))$ the per-variable instantiation is by definition:

$$\mathit{vals}^{\Theta}_{x_i} = \mathit{vals}(\hat{\beta}(p(\vec{u})), w[0], x_i),$$

while $\Theta' := \mathit{spec}(\beta_\varnothing, w[0], Q_1x_1 \ldots Q_kx_k : \hat{\beta}(p(\vec{u})))$ applies the current valuations first:

$$\begin{array}{rcl} \mathit{vals}^{\Theta'}_{x_i} & = & \mathit{vals}((\hat{\beta_\varnothing} \circ \hat{\beta})(p(\vec{u})), w[0], x_i) \\ & = & \mathit{vals}(\hat{\beta}(p(\vec{u})), w[0], x_i). \end{array}$$

We conclude that for each quantified variable x_i on a state $w[0]$ the same instances are computed. Thus we are free to move substitutions from either side to the other.

Corollary 39
It follows that

$$(w,\beta) \models \varphi \iff (w,\beta_\varnothing) \models \hat{\beta}(\varphi) \iff w \models \hat{\beta}(\varphi).$$

(Without proof.)

Remark 40 (Fallback to propositional logic)
Observe that in the absence of quantifiers and thus parametrised propositions with variables, always the empty valuation will be propagated. Applying it (or any other) valuation to ground propositions results simply in the proposition again. The $\dot{\rightarrow}$ operator is then equivalent to the Boolean "and".

Corollary 41 (Semantics of non-ground propositions)
From the declarative semantics it follows that partially instantiated propositions, that is, propositions which have at least one free variable, never satisfy any state.

Theorem 42 (Upwards compatibility of disjoint bindings)
For independently quantified formulae $\varphi, \psi \in p\mathrm{LTL}$ with $\mathit{bound}(\varphi) \cap \mathit{bound}(\psi) = \varnothing$, the bindings obtained from both branches of a binary $\vee$ operator can be combined by set union.

Proof:

$$\begin{array}{ll} & (w, \beta) \models \varphi \vee \psi \\ \equiv & (w, \beta) \models \varphi \text{ or } (w, \beta) \models \psi \\ \equiv & \quad (\exists \theta \in \mathit{spec}(\beta, w[0], \varphi) : \forall \sigma \in \theta : (w[0], \sigma \odot \beta) \models \bar{\varphi}) \\ & \text{or } (\exists \theta \in \mathit{spec}(\beta, w[0], \psi) : \forall \sigma \in \theta : (w[0], \sigma \odot \beta) \models \bar{\psi}) \\ \equiv & \exists \theta \in (\mathit{spec}(\beta, w[0], \varphi) \cup \mathit{spec}(\beta, w[0], \psi)) : \forall \sigma \in \theta : (w[0], \sigma \odot \beta) \models \bar{\varphi} \vee \bar{\psi} \end{array}$$

with

$$\begin{array}{rcl} \overline{Q_1 x_1 \ldots Q_m x_m : p(u_1, \ldots, u_n) \dot{\rightarrow} \psi} & := & p(u_1, \ldots, u_n) \wedge \psi \\ \overline{\psi} & := & \psi, \text{ otherwise} \end{array}$$

as bindings obtained from separate branches involve disjoint sets of variables and thus can never affect the evaluation of the opposite branch.

Since the namespaces of φ and ψ are disjoint and therefore any branch will fail under a substitution obtained from the opposite branch because of Corollary 41.

A similar result can be obtained for the temporal **U** operator, while $\wedge$ requires the combination of bindings through the crossproduct as *both* branches must find valuations. **R** is special since it is a combination of both: satisfaction requires both sides to hold, but evaluation can also proceed with bindings obtained only from the left-hand side.

This property gives us an alternative means of deriving the set of valuations that are inferred for a parametrised formula from the current state. As in predicate logic, if both branches use the same bound variable, a renaming can first be applied to obtain disjoint sets without changing the semantics of the formula.

Also, this is a convenient location to remind the reader that any obtained valuation can still be refuted on the current state, as evaluation of the right-hand side may still fail for that binding. Consider for example the trivial case of $\exists x : p(x) \dot{\rightarrow} \mathbf{ff}$ where many valuations can be derived, but all will be dropped after evaluation of the right-hand side.

Example 43 (Valuations)
We exemplify the mechanism of valuations: let

$$\begin{aligned}\varphi &:= \forall x \exists y : p(x,y) \dot{\rightarrow} \psi, \text{ and}\\ w &:= \{p(1,1), p(1,2), p(2,1), p(2,3)\},\ |w| = 1.\end{aligned}$$

We obtain the following valuations for $p(x,y)$:

$$\beta_1 = \{x/1, y/1\},\ \beta_2 = \{x/1, y/2\},\ \beta_3 = \{x/2, y/1\},\ \beta_4 = \{x/2, y/3\}$$

Observe that neither $\{x/1, y/3\}$ nor $\{x/2, y/2\}$ are inferred bindings, since no valuations are generated which do not produce instances of the active proposition in the current state. The resulting proof obligations are:

$$\begin{aligned} w \models \varphi \iff & \quad [(w,\beta_1) \models \psi \vee (w,\beta_2) \models \psi]\\ \wedge & \quad [(w,\beta_3) \models \psi \vee (w,\beta_4) \models \psi]\end{aligned}$$

Resolving this into Disjunctive Normal Form, we get:

$$\begin{aligned}\Theta \quad &= \{ \quad \{\{x/1,y/1\},\{x/2,y/1\}\},\ \{\{x/1,y/1\},\{x/2,y/3\}\},\\ &\qquad\ \{\{x/1,y/2\},\{x/2,y/1\}\},\ \{\{x/1,y/2\},\{x/2,y/3\}\}\ \}\\ w \models \varphi \quad &\iff \quad \bigvee_{\theta\in\Theta}\bigwedge_{\beta\in\theta}(w,\beta) \models \psi\end{aligned}$$

For further clarify the mechanism, we show the effect of two other quantifications:

$$\begin{aligned}\forall x \forall y \rightsquigarrow \ \Theta = &\ \{\ \{\{x/1,y/1\},\{x/1,y/2\},\{x/2,y/1\},\{x/2,y/3\}\}\ \}\\ \exists y \forall x \rightsquigarrow \ \Theta = &\ \{\ \{\{x/1,y/1\},\{x/2,y/1\}\},\ \{\{x/1,y/2\}\},\ \{\{x/2,y/3\}\}\ \}\end{aligned}$$

In the case of the same quantified variable occurring several times in the existence predicate, on the same state as above, $\{x/1\}$ is the only derived valuation:

$$\varphi \ := \ Qx : p(x,x) \dot{\rightarrow} \psi$$

Remark 44 (Temporal assertions in J-LO)
The formal model of the "Java Logical Observer" J-LO [18, 115] (see Section 5.4) only permitted quantification over events. It did not contain overlapping propositions, that is, in each state, exactly one proposition did hold. Quantification was implicit: the Globally-operator induced universal quantification, while a Finally resulted in existential quantification.

Example 45 (Error witness, path of valuations)
In case a formula is refuted, we are naturally interested in the concrete bindings obtained during its evaluation, for example, as a debugging aid. Consider the formula:

$$\varphi := \mathbf{G}\ \forall x : p(x) \dot{\rightarrow} \psi.$$

On every state, we need to find a fresh instantiation σ for the variable x such that $\widehat{\sigma \odot \beta}(\psi)$ holds. Because of the unwinding when Globally is expressed through Release (see Definition 20), we may obtain a new valuation for x in each state on a path w of length n given some initial binding β:

$$(w, \beta) \models \mathbf{G}\ \forall x : p(x) \dot{\rightarrow} \psi \iff \forall k < n \rightarrow (w[k], \beta) \models \forall x : p(x) \dot{\rightarrow} \psi.$$

If we eventually refute the formula, we do so for a specific binding, which we can consider as a *witness to the error*. This binding will of course contain the previous valuations β. But in the case of the Globally formula above, in some state $w[i]$, the obtained binding will not include any valuations bound previously to x, for example, on state $w[i-1]$, which might be helpful information to a user tracking down a particular problem. The semantics of the Globally operator requires us to "restart" evaluation based on β in each state.

As a formula refuted in state $w[j]$ will only contain bindings β obtained before starting evaluation of w and some value of x obtained at some state $w[i], i < j$, the combined valuation $\sigma \odot \beta$ does not contain information about the bound values that allowed evaluation to proceed thus far, that is, states $w[0], \ldots, w[i-1]$.

If such information should be provided to the user, the implementation can provide a record of all instantiations in the past or compute them again on demand from the beginning of the trace. As an example, assume that $\psi = \texttt{p(2)}$, which should hold when x is bound to the value of 2. On the trace

$$w = \{p(2)\}\varnothing\varnothing\{p(1)\}\{p(3)\}$$

the formula will be refuted on state $w[3] = \{p(1)\}$, yielding only the valuation $\{x/1\}$ (the previous states have been accepted because of quantification over the empty set). It holds no indication of the valuations obtained in previous states ($\{x/2\}$ in $w[0]$, the empty valuation in states $w[1]$ and $w[2]$), although they are material for not making the formula fail earlier. We also note that the path up to $w[3]$ is the *shortest trace* which refutes the formula.

3.2.3 Negation in pLTL

In a general framework of first order logic, negation is usually pushed through quantifiers, toggling them as it passes them. In temporal formulae, a similar behaviour occurs for temporal operators, that is, $\mathbf{F}/\mathbf{G}$ or $\mathbf{R}/\mathbf{U}$, respectively (see Definition 25 for positive form of plain LTL formulae). This is especially necessary for the automaton based construction as it assures that a normal form is attained.

In our setting with existence predicates, though, quantifiers are closely tied to the selector which enumerates the current objects that the right-hand side should affect. Thus, we do not have the notion of a negated selector. Accordingly, negations can only appear on the right-hand side of the existence predicate.

Let's consider the pLTL formula

$$\varphi := \neg \forall x : p(x) \dot{\rightarrow} \mathbf{F}\ q(x).$$

Figure 3.2 shows the result of evaluating the formula on three different traces. Naively pushing down the negation, we obtain the formula

$$\begin{aligned}\varphi' &:= \exists x : p(x) \dot{\rightarrow} \neg\mathbf{F}\ q(x)\\ &\equiv \exists x : p(x) \dot{\rightarrow} \mathbf{G}\ \neg q(x)\end{aligned}$$

Clearly, φ' shows the desired behaviour.

Trace	*Result for* φ/φ'
{p(1),p(2)};{q(1),q(2)}	**ff**
{p(1),p(2)};{q(1)}	**tt**
∅;{q(1),q(2)}	**ff**

Figure 3.2: Negation in pLTL

Definition 46 (Positive form of a pLTL formula)
Given a pLTL formula φ, we extend the rewriting rules for plain *LTL* formulae from Definition 25 through the rule:

$$\begin{aligned}&\neg Q_1 x_1 \dots Q_n x_n : p(y_1, \dots, y_m) \dot{\rightarrow} \psi\\ \longrightarrow\quad &\overline{Q}_1 x_1 \dots \overline{Q}_n x_n : p(y_1, \dots, y_m) \dot{\rightarrow} \neg\ \psi,\end{aligned}$$

where

$$\overline{Q}_i := \begin{cases}\exists, & \text{iff } Q_i = \forall\\ \forall, & \text{iff } Q_i = \exists.\end{cases}$$

Theorem 47 (Equivalence of positive form of a pLTL formula)
For each path w, valuation β, and formula φ, we show that

$$(w, \beta) \models \neg\varphi \iff (w, \beta) \models (\neg\varphi)^+.$$

Proof:

As for formulae that do not begin with an existence predicate, we know that we can apply the substitution simply to the right-hand side and obtain a partially instantiated formula. For this the regular finite path semantics holds on each subformula, so we restrict our consideration to the newly introduced quantified existence predicate with a single quantifier. By construction, the result can be lifted. Let

$$\varphi := Qx : p(y_1, \dots, y_n) \dot{\rightarrow} \psi$$

and

$$\Theta_Q := spec(\beta, w[0], Qx : p(y_1, \dots, y_n)).$$

Since in both the original and the normalised cases the bindings are calculated with respect to the same initial binding on the same state, on the one hand bindings (if any) are conjoined, while on the other hand they are disjoint:

$$\begin{array}{rcl}\Theta_\forall & := & \{\ \{\sigma_1, \ldots, \sigma_k\}\ \} \\ \Theta_\exists & := & \{\ \{\sigma_1\}, \ldots, \{\sigma_k\}\ \}\end{array}$$

We branch over the quantifier:

1. $Q = \forall$:

$$\begin{array}{rl} & (w, \beta) \models \neg\varphi \\ = & (w, \beta) \not\models \forall x : p(y_1, \ldots, y_n) \dot{\rightarrow} \psi \\ \equiv & \neg[\exists \theta \in \Theta_\forall \text{ s.th. } \forall \sigma \in \theta : (w, \sigma \odot \beta) \models \psi]\end{array}$$

This gives rise to two further cases: either there are some $p^{(n)}$ in the current state which allow specialisation of the current binding or not. For the latter, by definition of $\Theta_\forall$, we know that $\forall$ succeeds on no valuation, yielding an overall result of **ff** due to the negation in this case.

If valuations exist, by definition we only obtain conjuncts, that is, $|\Theta_\forall| = 1$.

$$\begin{array}{rl} & \neg\left[\bigvee_{\theta \in \Theta_\forall} \bigwedge_{\sigma \in \theta} (w, \sigma \odot \beta) \models \psi\right] \\ \equiv & \neg\left[\bigwedge_{\sigma \in \theta} (w, \sigma \odot \beta) \models \psi, \text{ with } \Theta_\forall = \{\theta\}\right] \\ \equiv & \bigvee_{\sigma \in \theta} (w, \sigma \odot \beta) \not\models \psi, \text{ with } \Theta_\forall = \{\theta\} \\ \equiv & \bigvee_{\sigma \in \{\sigma_1, \ldots, \sigma_k\}} (w, \sigma \odot \beta) \not\models \psi \end{array}$$

We will now take a look at the positive expansion and see that we indeed obtain the same result.

$$\begin{array}{rll} & (w, \beta) \models (\neg\varphi)^+ & \\ = & (w, \beta) \models (\neg\forall x : p(y_1, \ldots, y_n) \dot{\rightarrow} \psi)^+ & \\ = & (w, \beta) \models \exists x : p(y_1, \ldots, y_n) \dot{\rightarrow} (\neg\psi)^+ & \text{(rewriting rule)} \\ \equiv & \exists \theta \in \Theta_\exists \text{ s.th. } \forall \sigma \in \theta : (w[0], \sigma \odot \beta) \models (\neg\psi)^+ & \\ \equiv & \bigvee_{\theta \in \Theta_\exists} \bigwedge_{\sigma \in \theta} (w, \sigma \odot \beta) \models (\neg\psi)^+ & \\ \equiv & \bigvee_{\sigma \in \{\sigma_1, \ldots, \sigma_k\}} (w, \sigma \odot \beta) \models (\neg\psi)^+ & (|\Theta_\exists| = k,\ |\theta_i| = 1) \\ \equiv & \bigvee_{\sigma \in \{\sigma_1, \ldots, \sigma_k\}} (w, \sigma \odot \beta) \not\models \psi & \text{(induction)} \end{array}$$

By induction, the right-hand side will eventually reach an unquantified subformula. We conclude that positive normal form does not affect overall evaluation.

2. $Q = \exists$:
   A similar argument can be given for existential quantification.

3.3 Predicates

As propositions in the trace allow access to values from the underlying domain, we might as well make use of them: values are bound to variables, and given an interpretation, we can easily introduce *relations* over values into our logic. Furthermore, allowing also functions, we can construct Boolean *terms* over variables which can be evaluated.

Definition 48 (Terms)
Given a signature Σ of n-ary function symbols (including constants)

$$\Sigma := \bigcup \{F^{(n)} \mid n \in \mathbb{N}\}$$

and variables $V \subseteq \mathcal{V}$, we define the set of *terms* $T_\Sigma(V)$ inductively:

1. $F^{(0)} \subseteq T_\Sigma(V)$
2. $V \subseteq T_\Sigma(V)$
3. $f(t_1, \ldots, t_n) \in T_\Sigma(V)$ if $t_1, \ldots, t_n \in T_\Sigma(V)$ and $f \in F^{(n)}$.

Definition 49 (Predicate syntax)
Syntactically, a *predicate* has the same structure as a proposition, the arguments are terms with variables. Let $\mathcal{PrN}^{(n)}$ denote the *predicate names* of arity $n \geq 1$. The arguments are variables which must already have been bound by an existence predicate and its corresponding quantifier:

$$\begin{aligned}
&LTL(V \subset \mathcal{V}) ::= \ldots \\
&\quad \mid q(t_1, \ldots, t_n),\ q \in \mathcal{PrN}^{(n)},\ t_1, \ldots, t_n \in T_\Sigma(V) \\
&\quad \mid \ldots
\end{aligned}$$

Observe that variables and functions can only occur within a predicate and not directly in a formula.

We shall use the same notion of *all predicates* $\mathcal{Pr}$ and *ground predicates* $\mathcal{Pr}_\perp$ as we do for propositions, that is, that no variables may occur. Also, we deliberately confuse the constants with the elements from $\mathcal{D}$, that is $F^{(0)} = \mathcal{D}$.

Definition 50 (Predicate semantics)
Given an interpretation $\mathcal{I} := \langle \mathcal{D}, \xi \rangle$ (see Definition 6) extended to predicate names by $\xi(q) : \mathcal{D}^{(n)} \to \{\mathbf{tt}, \mathbf{ff}\}$ for each $q \in \mathcal{PrN}^{(n)}$ and $n \geq 1$, the finite path semantics of pLTL for a predicate $q(t_1, \ldots, t_n) \in \mathcal{Pr}$, where $t_i \in T_\Sigma(V)$ is extended through the following rules for predicates

$$\begin{aligned}
(w, \beta) &\models q(t_1, \ldots, t_n) && \text{iff } (\xi, \beta)(q(t_1, \ldots, t_n)) = \mathbf{tt} \\
(w, \beta) &\models \neg q(t_1, \ldots, t_n) && \text{iff } (\xi, \beta)(q(t_1, \ldots, t_n)) = \mathbf{ff}
\end{aligned}$$

where $\mathcal{I}$ yields with regard to $\beta : V \to \mathcal{D}$ the usual interpretation (ξ, β) of a term $t \in T_\Sigma(V)$ with

1. $(\xi, \beta)(c) = \xi(c)$, for $c \in F^{(0)}$
2. $(\xi, \beta)(v) = \beta(v)$ for $v \in \mathcal{V}$,
3. $(\xi, \beta)(f(u_1, \ldots, u_n)) = \xi(f)((\xi, \beta)(u_1), \ldots, (\xi, \beta)(u_n))$, for $f \in F^{(n)}$,

and in the same way for a predicate

$$(\xi, \beta)(q(t_1, \ldots, t_n)) = \xi(q)((\xi, \beta)(t_1), \ldots, (\xi, \beta)(t_n)).$$

Example 51 (Predicates over natural numbers)
Let $\mathcal{D} = \mathbb{N}$ and ξ be the standard semantics of functions and relations on natural numbers. Let

$$\varphi := \mathbf{G}\ [\forall o, y : set(o, y) \dot{\to} \mathbf{G}\ \forall z : set(o, z) \dot{\to} (y \leq z)]$$

Here we have $set \in \mathcal{PN}^{(2)}$, $(\leq) \in \mathcal{P}r\mathcal{N}^{(2)}$. Consider now that *set* might correspond to an event where an object (or object attribute) o gets assigned some value (object reference could be represented as a natural number). Then this formula states that the value must only monotonically be incremented, otherwise it fails.

Remark 52 (Predicates and temporal operators)
Since predicates are evaluated with respect to bound values and functions are side-effect free, certain invariants hold which may be used to optimise formulae in the sense of eliminating temporal sub-formulae in favour of a simple predicate: for all instances of x, y it holds that

$$\begin{array}{rcl} \mathbf{G}\ q(x, y) & \iff & q(x, y) \\ \mathbf{G}\ \neg q(x, y) & \iff & \neg q(x, y) \end{array}$$

since without rebinding, which can only be caused through occurrence of quantification within the temporal operator, the result cannot change. This is similar to simplifying temporal formulae such as $\mathbf{G}\ \mathbf{tt}$.

Next, we show that for *ground* formulae, we can first evaluate the predicates and then apply the finite path semantics.

Definition 53 (Evaluation of predicates in formulae)
Given a pLTL formula over ground propositions $\mathcal{P}_\perp$ and predicates $\mathcal{P}r_\perp$, we define the evaluation of predicates with regard to an interpretation $\mathcal{I} = \langle \mathcal{D}, \xi \rangle$ as follows:

$$
\begin{array}{rcl}
eval^{\mathcal{I}} & : & p\mathrm{LTL} \to p\mathrm{LTL} \\
eval^{\mathcal{I}}(q(\vec{d})) & := & \xi(q)(\vec{d}),\ q(\vec{d}) \in \mathcal{P}r_{\perp} \\
eval^{\mathcal{I}}(\neg q(\vec{d})) & := & \neg\xi(q)(\vec{d}),\ q(\vec{d}) \in \mathcal{P}r_{\perp} \\
eval^{\mathcal{I}}(\varphi \ \vee \ \psi) & := & eval^{\mathcal{I}}(\varphi) \ \vee \ eval^{\mathcal{I}}(\psi) \\
eval^{\mathcal{I}}(\varphi \ \wedge \ \psi) & := & eval^{\mathcal{I}}(\varphi) \ \wedge \ eval^{\mathcal{I}}(\psi) \\
eval^{\mathcal{I}}(\varphi \ \mathbf{U} \ \psi) & := & eval^{\mathcal{I}}(\varphi) \ \mathbf{U} \ eval^{\mathcal{I}}(\psi) \\
eval^{\mathcal{I}}(\varphi \ \mathbf{R} \ \psi) & := & eval^{\mathcal{I}}(\varphi) \ \mathbf{R} \ eval^{\mathcal{I}}(\psi) \\
eval^{\mathcal{I}}(\varphi) & := & \varphi,\ \text{otherwise } (\mathbf{tt},\ \mathbf{ff},\ \text{propositions})
\end{array}
$$

The temporal formula remains unchanged except that all predicates have been evaluated to their Boolean results. It can then be subjected to checking with respect to the finite path semantics.

Theorem 54 (Correctness of evaluation)
Given a ground formula $\phi \in p\mathrm{LTL}$, an interpretation $\mathcal{I}$, and a path w,

$$w \models \phi \iff w \models eval^{\mathcal{I}}(\phi).$$

(Without proof.)

Example 55 (Singleton pattern)
Consider a variant of the *Singleton design pattern* [57]:

> *"Ensure a class has only one instance, and provide a global point of access to it"*

The informal contract for such a class prescribes that, after creating an object with *new*, the consumer must additionally invoke an explicit instantiation method and use the returned handle instead of the object created by *new*. The instantiation method would, for example, use a class variable (opposed to an instance variable) to look up a previously created instance or create a new one if none exists yet.

We want to observe whether a class adheres to this principle. Every invocation of `Singleton.instantiate` should return the same object identifier. The event `method exit`$(\mathit{Singleton.instantiate}, \mathit{this}, C.m, \mathit{target}, \mathit{args}, r)$ captures this method call and its return value. As the only relevant information in this event is the instance returned r (the return value of the method call), for convenience we abstract this into a proposition $\mathit{inst}(r)$. This abstraction will later be formalised in Chapter 4. We can express this pattern in an event-based setting without overlapping propositions as follows:

$$
\begin{array}{rl}
 & \mathbf{G}\ \forall x : \mathit{inst}(x) \dot{\rightarrow} \mathbf{G}\ (\mathit{inst}(x) \vee \nexists y : \mathit{inst}(y) \dot{\rightarrow} \mathbf{tt}) \\
\equiv & \mathbf{G}\ \forall x : \mathit{inst}(x) \dot{\rightarrow} \mathbf{G}\ (\mathit{inst}(x) \vee \forall y : \mathit{inst}(y) \dot{\rightarrow} \mathbf{ff})
\end{array}
$$

Observe that this is equivalent to

$$\mathbf{G}\ \forall x : inst(x) \dot{\rightarrow} \mathbf{G}\ (\forall y : inst(y) \dot{\rightarrow} inst(x))$$

After seeing any instantiation, either no subsequent instantiation $inst(y)$ occurs (universal quantification yields $\mathbf{tt}$, thus accepting that state), or, if it occurs, we assert that the returned handle is the same one we have observed previously (there will only be a single instantiation for the universally quantified variable y due to the event-based setting): the inner Globally will fail for an event $inst(y)$ that has $x \neq y$.

Remark 56 (Predicates and equality)
Of course relations may also be specified statically: for example, equality can be expressed as a binary predicate $(=)$ with the natural semantics:

$$\xi(=) := \lambda\ x\ y. \begin{cases} \mathbf{tt}, & \text{if } x = y \\ \mathbf{ff}, & \text{otherwise} \end{cases}, \forall x, y \in \mathcal{D}$$

that is, each member of the domain is in this relation with only itself. That way, it can be used to compare bound variables.

3.4 Translating LTL Formulae into Alternating Finite Automata

In the following, we introduce alternating finite automata and recapitulate how LTL formulae can be translated into them such that the language induced by an LTL formula φ is exactly the language accepted by an appropriately constructed alternating automaton A_φ. That way, we can use A_φ to decide if an input w (a path) is a model for φ.

The automaton construction is then extended in the next section to collect the bindings of our parametrised formalism.

Alternating finite automata are due to Chandra, Kozen, and Stockmeyer [25, 26]. In the classical, that is, propositional setting, alternating finite automata generated from LTL formula consume finite words and produce as result a run in the form of a tree; we paraphrase [124].

Definition 57 (Alternating finite automaton)
An alternating finite automaton is a tuple $A := \langle \Sigma, Q, q_0, \delta, F \rangle$. Σ is the finite nonempty input alphabet, Q a finite nonempty set of states, $q_0 \in Q$ a unique initial state, a transition function $\delta : Q \times \Sigma \rightarrow 2^{2^Q}$ where we consider 2^{2^Q} as a Boolean formula over states in DNF, and a set $F \subseteq Q$ of accepting states. Due to universal choice, a run of an alternating automaton yields a *tree*. Existential choice allows non-deterministic choice between *all trees*.

A run tree of A for a finite word $w = a_0, a_1, \ldots, a_{n-1}$ is a tree where each node x is assigned a *label* $\lambda(x) \in Q$. Every node in a *run tree* of the automaton has a unique predecessor, except for the root of the tree, denoted by ϵ, which has no parent. The *level* of a node $|x|$ is its distance from the root node, where $|\epsilon| = 0$.

The labelling has the following properties: $\lambda(\epsilon) = q_0$, and a node x with $|x| = i$ has a set $\{x_1, \ldots, x_k\}$ of successor nodes, if $\{\lambda(x_1), \ldots, \lambda x_k\} \in \delta(\lambda(x), a_i)$. Consequently, there are two possibilities for a sink node, that is, a node without successor nodes: a true node if $\varnothing \in \delta(\lambda(x), a_i)$ and a false node if $\delta(\lambda(x), a_i) = \varnothing$.

The run of an alternating finite automaton is *accepting*, if all nodes on depth n are in the automaton's acceptance set and it does not contain any **ff**-nodes.

We say $w \in A_\varphi$, iff there exists an accepting run tree for A_φ on w.

For a node x of a run tree with $|x| = i$ and a non-empty set of successors $\{x_1, \ldots, x_k\}$ with $\varnothing \in \delta(\lambda(x), a_i)$, without loss of generality, we can treat x as a true-node and eliminate the subtree with root x. Accordingly, *delta* shall have the following property:

$$\varnothing \in \delta(q, a) \Rightarrow \delta(q, a) = \{\varnothing\}$$

Next, based on a formula φ, we construct an alternating automaton A_φ that accepts the same language L_{A_φ} as the language of the formula L_φ (see Definition 21). For the construction of the alternating automaton from a formula, we need the closure of a formula where negations only occur on the propositional level (see Lemma 26).

For the remainder of this section, we also assume that all occurrences of the unary temporal operators **F** and **G** have been rewritten into their counterparts using **U** and **R** (see Definition 23).

Definition 58 (Closure of a formula)
The *closure* $cl : LTL \rightarrow 2^{LTL}$ of a formula ϕ in positive form, is defined through all (syntactical) subformulae of ϕ.

- $\phi \in cl(\phi)$
- $\mathbf{tt}, \mathbf{ff} \in cl(\phi)$
- if $\varphi \vee \psi \in cl(\phi)$ then $\varphi, \psi \in cl(\phi)$
- if $\varphi \wedge \psi \in cl(\phi)$ then $\varphi, \psi \in cl(\phi)$
- if $\varphi\ \mathbf{R}\ \psi \in cl(\phi)$ then $\varphi, \psi \in cl(\phi)$
- if $\varphi\ \mathbf{U}\ \psi \in cl(\phi)$ then $\varphi, \psi \in cl(\phi)$

Definition 59 (Alternating finite automaton for an *LTL* formula)
By induction on the structure of a positive *LTL* formula ϕ, we obtain the corresponding alternating automaton, where each element of the *closure* of ϕ is a state

in $A_\phi = \langle 2^{AP}, cl(\phi), \phi, \delta, F\rangle$. The set of final states contains the $\mathbf{tt}$ state and all Release nodes:

$$F := \{\mathbf{tt}\} \cup \{q \in cl(\phi) \mid q = \varphi\ \mathbf{R}\ \psi\}$$

The transition relation is defined as follows.

– $\delta(\mathbf{tt}, a) = \{\varnothing\}$

– $\delta(\mathbf{ff}, a) = \varnothing$ ($\mathbf{tt}$ and $\mathbf{ff}$ can be considered sinks)

For propositions $p \in AP$:

– $\delta(p, a) = \{\{\mathbf{tt}\}\}$, if $p \in a$

– $\delta(\neg p, a) = \{\{\mathbf{ff}\}\}$, if $p \in a$

– $\delta(p, a) = \{\{\mathbf{ff}\}\}$, if $p \notin a$

– $\delta(\neg p, a) = \{\{\mathbf{tt}\}\}$, if $p \notin a$

Binary Boolean operators, $\varphi, \psi \in LTL$:

– $\delta(\varphi \wedge \psi, a) = \delta(\varphi, a) \otimes \delta(\psi, a)$

– $\delta(\varphi \vee \psi, a) = \delta(\varphi, a) \cup \delta(\psi, a)$

Temporal operators:

– $\delta(\varphi\ \mathbf{U}\ \psi, a) = \delta(\psi, a) \cup (\{\{\varphi\ \mathbf{U}\ \psi\}\} \otimes \delta(\varphi, a))\ \}$

– $\delta(\varphi\ \mathbf{R}\ \psi, a) = (\delta(\varphi, a) \otimes \delta(\psi, a)) \cup (\{\{\varphi\ \mathbf{R}\ \psi\}\} \otimes \delta(\psi, a))$

We note that the explicit Boolean disjunction and the disjunction implicit in the Until and Release definitions introduce non-determinism into the automaton.

Theorem 60 (Equivalence of A_φ and φ)
Given a formula $\varphi \in LTL$ in positive form, it holds that

$$A_\varphi \equiv \varphi.$$

Proof:

We show equivalence of the formula and the respective automaton construction by showing equality of their corresponding languages, that is

$$L_{A_\varphi} = L_\varphi.$$

We outline the proof based on the inductive construction of the transition function: on the propositional level, it is easy to see that the automaton accepts an input corresponding to the semantics of propositional formula by transiting from the initial state to a Boolean *formula of conjunctions* where propositions have been evaluated to **tt** based on the input. *Non-determinism* resolves the choice between *disjuncts*.

On the temporal level, we take a look at the Until operator. We recall the finite path semantics from Definition 20 for a path w of length n:

$$\begin{array}{lll} w[j] & \models \varphi\ \mathbf{U}\ \psi & \text{iff} \quad \exists k\ (j \leq k < n) \text{ s.th. } w[k] \models \psi \\ & & \qquad \wedge\ \forall l\ (j \leq l < k) \rightarrow w[l] \models \varphi \end{array}$$

The induced language is

$$L_{\varphi \mathbf{U} \psi} := \bigcup_{k \in \mathbb{N}} ((\Sigma^k \cdot L_\psi) \cap (\bigcap_{0 \leq l < k} (\Sigma^l \cdot L_\varphi)))$$

($\cdot$ is language concatenation; for $k = l = 0$ we assume the right-hand side of the intersection to return the neutral element of intersection, that is, Σ^* here). We call the language for a given k $L^{(k)}_{\varphi \mathbf{U} \psi}$.

On the automaton side, we obtain by construction

$$L_{A_{\varphi \mathbf{U} \psi}} \quad := \quad L_{A_\psi} \cup (L_{A_\varphi} \cap \Sigma \cdot L_{A_{\varphi \mathbf{U} \psi}})$$

Unwinding the recursive definition, we obtain

$$\begin{array}{rlll} L_{A_{\varphi \mathbf{U} \psi}} = & & L_{A_\psi} & (0) \\ & \cup & L_{A_\varphi} \cap (\Sigma \cdot L_{A_\psi}) & (1) \\ & \cup & L_{A_\varphi} \cap (\Sigma \cdot (L_{A_\varphi} \cap (\Sigma \cdot L_{A_\psi}))) & (2) \\ & \cup & \ldots & \end{array}$$

The term on row (2) can be transformed into

$$L_{A_\varphi} \cap (\Sigma \cdot (L_{A_\varphi} \cap (\Sigma \cdot L_{A_\psi}))) = L_{A_\varphi} \cap (\Sigma \cdot L_{A_\varphi}) \cap (\Sigma \cdot \Sigma \cdot L_{A_\psi})$$

Generalising this, for each disjunction i, we obtain i conjuncts of the form $\Sigma^j \cdot L_{A_\varphi}$, $0 \leq j < i$ and a final conjunct $\Sigma^i \cdot L_{A_\psi}$. We define:

$$L^{(i)}_{A_{\varphi \mathbf{U} \psi}} := (\bigcap_{0 \leq j < i} (\Sigma^j \cdot L_{A_\varphi})) \cap (\Sigma^i \cdot L_{A_\psi})$$

Forming the disjunction over all rows, and assuming that $L_\varphi = L_{A_\varphi}$ and $L_\psi = L_{A_\psi}$ for the induction, we obtain:

$$\begin{array}{rcl} L_{A_{\varphi \mathbf{U} \psi}} & := & \bigcup_{k \in \mathbb{N}} L^{(k)}_{A_{\varphi \mathbf{U} \psi}} \\ & = & \bigcup_{k \in \mathbb{N}} ((\bigcap_{0 \leq j < k} (\Sigma^j \cdot L_{A_\varphi})) \cap (\Sigma^k \cdot L_{A_\psi})) \\ & = & \bigcup_{k \in \mathbb{N}} ((\bigcap_{0 \leq j < k} (\Sigma^j \cdot L_\varphi)) \cap (\Sigma^k \cdot L_\psi)), \end{array}$$

thus completing the proof for $L_{\varphi \mathbf{U} \psi} = L_{A_{\varphi \mathbf{U} \psi}}$. The corresponding proof for Release formulae is left as an exercise to the reader.

Remark 61 (Size of the construction)
Observe that there is a fixed number of states of the automaton depending only on the formula. Since the transition function of an alternating automaton is a power set construction over the closure of the formula, this limits also the maximum size of those successor formulae with respect to standard Boolean optimisations (for example, $\varphi \wedge \varphi = \varphi$). Because we handle these formulae as sets, we get part of those optimisations for free.

Some issues like having the states for the same proposition both in positive and negative form in the same clause are not covered by the construction above but rather deferred to evaluation of a run. An implementation of the algorithm above might want to short-cut those right at construction time, as might be done with conjoined edges leading to the **tt** node.

Example 62 (Sample construction)
We illustrate the construction with an example taken from [53]. Let

$$\phi := \mathbf{G}\ (\neg a \rightarrow \neg b\ \mathbf{U}\ c)$$

and the input alphabet $\Sigma := 2^{\{a,b,c\}}$. First, we eliminate the implication and express the globally using Release:

$$\phi = \mathbf{ff}\ \mathbf{R}\ (a \vee \neg b\ \mathbf{U}\ c)$$

As the formula is already in positive form, we can directly derive the closure of the formula:

$$cl(\phi) = \{\ \mathbf{ff}\ \mathbf{R}\ (a \vee \neg b\ \mathbf{U}\ c), a \vee \neg b\ \mathbf{U}\ c, a, \neg b\ \mathbf{U}\ c, \neg b, c, \mathbf{tt}, \mathbf{ff}\ \}$$

The automaton will thus have two accepting states: the initial state since it is a Release state and of course **tt**.

$$F := \{\ \mathbf{tt}, \mathbf{ff}\ \mathbf{R}\ (a \vee \neg b\ \mathbf{U}\ c)\ \}$$

Instead of giving the transition table as an eight by eight matrix (2^3 input characters and $|cl(\phi)| = 8$ states), we illustrate the resulting automaton graphically in Figure 3.3 (conjoined edges with a dot indicate a conjunction). Observe that in the absence of a Next operator only temporal nodes, that is, nodes containing a temporal outermost operator, and **tt**/**ff** are reachable. The other nodes, like $a \vee \neg b\ \mathbf{U}\ c$, are only used during construction, but take no part in a run. As a further simplification for presentation, we do not label edges with the entire input letter, but only give the propositional formulae which have to hold in DNF, treating non-specified propositions as wildcards. Since no conjunctions with atomic propositions are present, all labels have only one proposition set—multiple labels also indicate disjunctions. That is, the label $\{a\}, \{c\}$ stands for all inputs where either the proposition a or the proposition c is set. The complete construction will be discussed in detail in Section 3.5.2 in the context of parametrised automata.

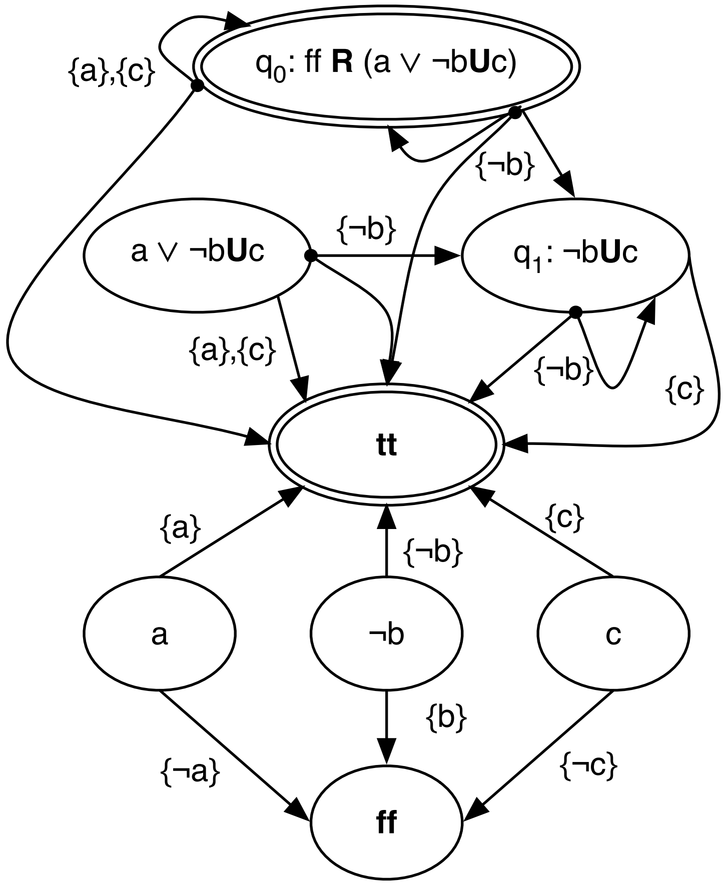


Figure 3.3: Sample alternating finite automaton

Example 63 (Sample run)
Consider the following input

$$w := \{a, b\}\varnothing\{a\}\{b, c\}\{a\}$$

to the automaton in Figure 3.3 where we label the two reachable non-trivial states (that is, excluding propositional formulae) as follows:

$$\begin{array}{rcl} q_0 & = & \mathbf{ff}\ \mathbf{R}\ (a \vee\ \neg b\ \mathbf{U}\ c) \\ q_1 & = & \neg b\ \mathbf{U}\ c \end{array}$$

The automaton starts in the initial state q_0, yielding successively the following accepting runtree where each set lists the labels corresponding to the conjoined nodes on the corresponding level:

$$\begin{array}{lccr} & & \{q_0\} & \textit{accepting} \\ \{a, b\} & \longrightarrow & \{q_0, \mathbf{tt}\} & \textit{accepting} \\ \varnothing & \longrightarrow & \{q_0, q_1, \mathbf{tt}\} & \textit{rejecting} \\ \{a\} & \longrightarrow & \{q_0, q_1, \mathbf{tt}\} & \textit{rejecting} \\ \{b, c\} & \longrightarrow & \{q_0, \mathbf{tt}\} & \textit{accepting} \\ \{a\} & \longrightarrow & \{q_0, \mathbf{tt}\} & \textit{accepting} \end{array}$$

Every clause containing q_1 is rejecting since this state is not an accepting state.

Complexity

The membership problem for an arbitrary alternating finite automaton A with input alphabet Σ and an arbitrary word $w \in \Sigma^*$ is *P-complete* [82]. Conversion of an alternating finite automaton to a nondeterministic automaton incurs an exponential blowup. We conclude this recapitulation by stating that alternating automata are as expressive as nondeterministic automata, but exponentially more succinct.

3.5 Parametrised Automaton Construction

We now proceed to extend the automata with respect to valuations. In the following, we first present a naive expansion with regard to an arbitrary but fixed domain. This expansion will also illustrate the worst-case behaviour of our construction. Then, we proceed with an implementation which statically calculates the abstract successor states, and instantiates concrete successors on the fly.

3.5.1 Handling Quantified Propositions

When we take a look at the structure of the extension from propositional LTL to parametrised pLTL in Definition 37, we can observe that the basic formalism for checking the trace remains unchanged from the moment on when we instantiate an existence predicate: we take a ground (parametrised) proposition as an atomic proposition.

The set of all ground propositions is countable for a countable domain, as parametrised propositions can be seen as vectors of fixed size n over countable sets. Since we bind variables to, for example, object references, which are defined through a countable set of classes and a countable set of instances, the principle is sound.

Depending on their position in the automaton, states may also be labelled with (possibly negated) unquantified parametrised propositions. By construction, due to the syntactical limits already imposed on the pLTL formula, such a state can only be reached through a path where all occurring variables have been matched previously by existence predicates.

When leaving a node that is labelled with an existence predicate, we have to branch into the sub-automata corresponding to the respective instantiations from the current input. For an n-ary predicate, there can be at most $|\mathcal{D}|^n$ instances in the current state. Depending on whether we have existential or universal quantification, these successor states are either disjoint or conjoined. This effect is independent of the quantification: conjoined edges have to be resolved within the same run, while disjoint edges are resolved by non-deterministically picking an accepting run-tree. As we have to treat the non-determinism algorithmically to find an accepting run, we have to visit *all* successors states, whether resulting from universal or existential children.

Remember from the introduction to parametrised propositions (see Section 3.2) that only a potentially minimal number of valuations actually occur on the trace, although the underlying domain is bigger. Even then, not all input states may contribute to new bindings. This holds especially true for event-based system, that is, where the underlying model corresponds to a program, where only a singleton proposition holds in each state. We will see that an explicit, flat construction based on the set of ground propositions is not practicable and will later provide an on-the-fly construction.

Example 64 (Sample derivation of flat states)
Let $\mathcal{V} = \{x, y\}$, $\mathcal{D} = \{1, 2, 3\}$, $\mathcal{PN} = \{p^{(1)}, q^{(2)}\}$, and $\{p(x), q(x, y)\} \subseteq cl(\phi)$. It follows that

$$\begin{aligned}\mathcal{P}_\perp \;=\;& \{p(1), p(2), p(3)\} \cup \\ & \{q(1,1), q(1,2), q(1,3), q(2,1), q(2,2), q(2,3), q(3,1), q(3,2), q(3,3)\}\end{aligned}$$

The size of $\mathcal{P}_\perp$ is thus 12, yielding already $2^{12} = 4096$ possible inputs that the flattened automaton should accept. For an existence predicate $\forall x \exists y : q(x, y) \dot{\rightarrow} \psi$ we have at most 3^3 successors if all q-propositions above are set in a state.

Definition 65 (Ground closure $cl_\perp^\mathcal{I}$)
Let $cl_\perp^\mathcal{I} : p\mathrm{LTL} \rightarrow 2^{p\mathrm{LTL}}$ denote the *ground closure* of ϕ with respect to some interpretation $\mathcal{I}$ of predicates. It includes formulae with quantifiers and all ground

instances of subformulae with free variables:

$$\begin{aligned} cl^{\mathcal{I}}_{\perp}(\phi) &:= \bigcup_{\varphi \in cl'(\phi)} \{\ (eval^{\mathcal{I}} \circ \hat{\sigma})(\varphi) \mid \sigma \in [\mathcal{V} \to \mathcal{D}]\} \\ free(\hat{\sigma}(\varphi)) &= \varnothing \text{ since } \sigma \text{ is a total function} \end{aligned}$$

where $cl' : p\text{LTL} \to 2^{LTL(V)}$ for $V = bound(\phi)$ is the extension of cl to pLTL through rule $(*)$:

- $\phi \in cl'(\phi)$
- $\mathbf{tt}, \mathbf{ff} \in cl'(\phi)$
- if $\varphi \vee \psi \in cl'(\phi)$ then $\varphi, \psi \in cl'(\phi)$
- if $\varphi \wedge \psi \in cl'(\phi)$ then $\varphi, \psi \in cl'(\phi)$
- if $\varphi \mathbf{R} \psi \in cl'(\phi)$ then $\varphi, \psi \in cl'(\phi)$
- if $\varphi \mathbf{U} \psi \in cl'(\phi)$ then $\varphi, \psi \in cl'(\phi)$
- if $Q_1 x_1 \ldots Q_n x_n : p(u_1, \ldots, u_m) \dot{\to} \psi \in cl'(\phi)$ then $\psi \in cl'(\phi)$ $(*)$

The construction of cl' can be seen as a purely syntactical extension of cl.

Predicates will have been eliminated in that phase. Note that contrary to fully instantiated propositions, the left-hand side of a binding expression is not an explicit state in the construction since it is not a syntactically valid sub-formula.

Definition 66 (Expanded alternating automaton)
The *expanded (flat) alternating automaton* $A_{\perp}(\phi) := \langle 2^{\mathcal{P}_{\perp}}, Q, q_0, \delta_{\perp}, F\rangle$ for a pLTL formula ϕ and an interpretation $\mathcal{I}$ is constructed as follows:

$$\begin{aligned} Q &:= cl^{\mathcal{I}}_{\perp}(\phi) \\ q_0 &:= \phi \\ F &:= \{\mathbf{tt}\} \cup \{q \in cl^{\mathcal{I}}_{\perp}(\phi) \mid q = \varphi \mathbf{R} \psi\} \end{aligned}$$

The behaviour of the transition function must handle the case when we encounter an *empty set of bindings for the parametrised proposition* at hand: for existential quantification, this means we move to the sink, thus rejecting this branch of the run. In the case of universal quantification, we short-cut to the accepting **tt** state.

The transition relation $\delta_{\perp}$ behaves like that of the plain alternating automaton introduced in Definition 59. For an existence predicate, it is defined as follows for $\{u_1, \ldots, u_m\} \subseteq \{x_1, \ldots, x_n\} \cup \mathcal{D}$ and $\{x_1, \ldots, x_n\} \subseteq \{u_1, \ldots, u_m\}$ (the u_i not in the set of quantified variables have been bound previously; all quantified variables x_i

must occur in the selector):

$$\delta_\perp(Q_1x_1 \ldots Q_nx_n : p(u_1, \ldots, u_m) \dot{\rightarrow} \psi, a)$$

$$:= \begin{cases} \{\{\mathbf{tt}\}\}, & \text{iff } Q_1 = \forall \text{ and } \nexists p(d_1, \ldots, d_m) \in a, \\ & \text{such that } p(d_1, \ldots, d_m) \downarrow p(u_1, \ldots, u_m) \\ \{\{\mathbf{ff}\}\}, & \text{iff } Q_1 = \exists \text{ and } \nexists p(d_1, \ldots, d_m) \in a, \text{ ditto} \\ \bigcup_{\theta \in \Theta} \left\{ \bigotimes \{\delta_\perp(\hat{\sigma}(\psi), a) \mid \sigma \in \theta\} \right\}, & \text{otherwise} \end{cases}$$

$$\Theta := spec(\beta_\varnothing, a, Q_1x_1 \ldots Q_nx_n : p(u_1, \ldots, u_m))$$

$\bigotimes$ is again the conjunction of clauses (product, see Definition 37) applied as a folding over sets (the inner set will never be empty: it contains at least the empty set). Θ is derived like in the extended finite-path semantics, where the valuations of the variables $x_1, \ldots, x_n$ are based on the input a instead of $w[j]$. Remember that $\hat{\sigma}$ will only replace the now free occurrences of the quantified variables on the right-hand side.

For all other nodes (propositions, Boolean functions, $\mathbf{U}/\mathbf{R}$), $\delta_\perp$ structurally behaves like the transition function δ of the normal alternating automaton (recursive invocations call to $\delta_\perp$, naturally).

Theorem 67 (Equivalence of expanded automaton and finite path semantics)
For a pLTL formula ϕ and the corresponding expanded automaton $A_\perp(\phi)$, it holds that:

$$\phi \equiv A_\perp(\phi).$$

Proof:

Again, we prove the equivalence by way of the languages of the formula and the automaton and show that $L_\phi \equiv L_{A_\perp(\phi)}$. Having previously discussed formulae without parametrised propositions (see Theorem 60), we now only need to consider the new rules in both formalisms. Let $\phi = Q_1x_1 \ldots Q_nx_n : p(u_1, \ldots, u_m) \dot{\rightarrow} \psi$. The finite path semantics from Definition 37 was given as:

$$\begin{aligned} & \bigvee_{\theta \in \Theta} \bigwedge_{\sigma \in \theta} (w[j], \sigma \odot \beta) \models \psi \\ \equiv \; & \bigvee_{\theta \in \Theta} \bigwedge_{\sigma \in \theta} (w[j], \beta_\varnothing) \models (\widehat{\sigma \odot \beta})(\psi) && (*) \\ \equiv \; & \bigvee_{\theta \in \Theta} \bigwedge_{\sigma \in \theta} w[j] \models (\widehat{\sigma \odot \beta})(\psi) && (Cor.\ 39) \end{aligned}$$

with

$$\begin{aligned} \Theta &= spec(\beta\ \ , w[j], Q_1x_1 \ldots Q_kx_k : p(u_1, \ldots, u_m)) \\ &= spec(\beta_\varnothing, w[j], Q_1x_1 \ldots Q_kx_k : \hat{\beta}(p(u_1, \ldots, u_m))) \end{aligned}$$

Both for the finite path semantics and the automaton, we know that all variables on the left-hand side of the existence predicate are already instantiated apart from variables quantified in ϕ, which also implies $\beta = \beta_\varnothing$ and $j = 0$ on the start of the trace, and, due to $(*)$ on the subsequent trace (see Theorem 38). Thus for every valuation σ in $\bigcup \Theta$: $\hat{\sigma}(p(u_1, \ldots, u_m)) \in \mathcal{P}_\perp$. We remind the reader that *spec* will yield $\varnothing$ if there is no valuation for outermost existential quantification and $\{\varnothing\}$ for a universal quantifier. We obtain for the finite paths semantics with $\beta = \beta_\varnothing$:

$$L_\phi := \bigcup_{\theta \in \Theta} \bigcap_{\sigma \in \theta} L_{\hat{\sigma}(\psi)}$$

For the three distinct cases of $\delta_\perp(\phi, w[0])$ above we may obtain either:

1. $\{\{\mathbf{tt}\}\}$: accepting configuration, thus any word with prefix $w[0]$ in $L_{A_\perp(\phi)}$ and also in L_ϕ due to $\Theta = \{\varnothing\}$.

2. $\{\{\mathbf{ff}\}\}$: rejecting configuration, thus words with prefix $w[0]$ not in $L_{A_\perp(\phi)}$ and also not in L_ϕ due to $\Theta = \varnothing$.

3. Otherwise, we obtain $\Theta' = \Theta = \mathit{spec}(\beta_\varnothing, w[0], Q_1 x_1 \ldots Q_n x_n : p(u_1, \ldots, u_m))$. The automaton yields:
$$L_{A_\phi} := \bigcup_{\theta \in \Theta} \bigcap_{\sigma \in \theta} L_{A_{\hat{\sigma}(\psi)}}$$

Size of the Construction

Clearly, neither calculating the ground closure with a worst-case size of $|cl^{\mathcal{I}}_\perp(\phi)| \leq |cl'(\phi)| \times |\mathcal{D}|^{|V|}$ (exponential in the number of variables, since each state is replicated once for each instantiation of its free variables) nor explicitly defining the transition function for $2^{|\mathcal{P}_\perp|}$ possible inputs for each such state is practicable for a sufficiently large set of parametrised propositions.

Next, we will give a bottom-up approach (in the sense that evaluation first descends into the non-temporal leaves and propagates propositions upwards) which statically calculates parametrised edges to successor states. Variable bindings will be instantiated on the fly.

3.5.2 Parametrised Automaton

If we take a closer look at the flat automaton $A_\perp$, it is easy to see that the sub-nodes hanging below an existence predicate all share the same structure. After all, they have been *generated from* the corresponding pLTL formula.

While the above expansion is instructive to get a grasp of the capabilities of our approach, operationally we want to tackle this differently. Especially, since the above construction relies on knowing the domain beforehand for the expansion. This is not actually a problem since at least for integer values real-world programs have a fixed range. Even seemingly continuous ranges like floating point numbers have a finitary representation in a program. But choosing the domain too broad would create many states which will never be visited at all in a concrete run.

The following construction preserves the abstract structure of the original formula and stores the active bindings in a separate component instead of branching into an instantiated sub-automaton on each quantification.

Definition 68 (Parametrised automaton)
The *parametrised automaton* of a pLTL formula ϕ is an AFA $A^{\mathcal{I}}(\phi):=\langle 2^{\mathcal{P}_{\perp}}, Q, q_0, \delta, F\rangle$ where the states of the automaton are pairs where the first component is an element of the (unflattened) closure of a corresponding pLTL formula ϕ in positive form, and has bindings in its second component:

$$\begin{aligned} Q &:= cl'(\phi) \times [\mathcal{V} \dashrightarrow \mathcal{D}] \\ q_0 &:= (\phi, \beta_{\varnothing}) \end{aligned}$$

Again, the final states are the $\mathbf{tt}$ state and all Release nodes:

$$F := (\{\mathbf{tt}\} \cup \{q \in cl'(\phi) \mid q = \varphi \ \mathbf{R}\ \psi\}) \times [\mathcal{V} \dashrightarrow \mathcal{D}]$$

The major difference now is that we do not generate the entire flattened automaton statically. Rather, we statically determine the parametrised, static structure, and explore it through instantiation on the fly. The number of states has shrunk again to only linear in the size of the formula, independently from the domain of quantification, as we only use the syntactic closure of the formula. However, the binding function now carries the burden of dynamically tracking the valuations. The transition function is then defined through:

$$\begin{aligned}
\delta((\mathbf{tt},\beta),a) &:= \{\varnothing\} \\
\delta((\mathbf{ff},\beta),a) &:= \varnothing
\end{aligned}$$

Propositions:

$$\begin{aligned}
\delta((p(\vec{u}),\beta),a) &:= \begin{cases} \{\{(\mathbf{tt},\beta)\}\}, & \text{iff } \hat{\beta}(p(\vec{u})) \in a \\ \{\{(\mathbf{ff},\beta)\}\}, & \text{otherwise} \end{cases} \\
\delta((\neg p(\vec{u}),\beta),a) &:= \begin{cases} \{\{(\mathbf{tt},\beta)\}\}, & \text{iff } \hat{\beta}(p(\vec{u})) \notin a, \mathit{free}(\hat{\beta}(p(\vec{u}))) = \varnothing \\ \{\{(\mathbf{ff},\beta)\}\}, & \text{otherwise} \end{cases}
\end{aligned}$$

Predicates:

$$\begin{aligned}
\delta((q(u_1,\ldots,u_k),\beta),a) &:= \begin{cases} \{\varnothing\}, & \text{iff } \hat{\xi}(q)(\beta(u_1),\ldots,\beta(u_k)) = \mathbf{tt} \\ \varnothing, & \text{otherwise} \end{cases} \\
\delta((\neg q(u_1,\ldots,u_k),\beta),a) &:= \begin{cases} \{\varnothing\}, & \text{iff } \hat{\xi}(q)(\beta(u_1),\ldots,\beta(u_k)) = \mathbf{ff} \\ \varnothing, & \text{otherwise} \end{cases} \\
\delta((\varphi \vee \psi,\beta),a) &:= \delta((\varphi,\beta),a) \cup \delta((\psi,\beta),a) \\
\delta((\varphi \wedge \psi,\beta),a) &:= \delta((\varphi,\beta),a) \otimes \delta((\psi,\beta),a) \\
\delta((\varphi \ \mathbf{U}\ \psi,\beta),a) &:= \delta((\psi,\beta),a) \cup (\{\{(\varphi \ \mathbf{U}\ \psi,\beta)\}\} \otimes \delta((\varphi,\beta),a)) \\
\delta((\varphi \ \mathbf{R}\ \psi,\beta),a) &:= (\delta((\varphi,\beta),a) \otimes \delta((\psi,\beta),a)) \cup (\{\{(\varphi \ \mathbf{R}\ \psi,\beta)\}\} \otimes \delta((\psi,\beta),a))
\end{aligned}$$

As before, the only occasion that we modify the bindings is on states prefixed with an existence predicate. For correct behaviour with respect to universal quantification over an empty domain we introduce a separate short-cut rules. Observe that we only have to consider the first quantifier, as due to the definition of *spec*, it is not possible to have valuations for *some* of the quantified variables in the same parametrised proposition but not for others. Existential quantification coincides with the general case as Θ will be empty, thus producing no successor states.

$$\delta((Q_1 x_1 \dots Q_n x_n : p(u_1, \dots, u_m) \dot{\rightarrow} \psi, \beta), a)$$

$$:= \begin{cases} \{\varnothing\}, \text{ iff } Q_1 = \forall \text{ and } \Theta = \varnothing \\ \bigcup_{\theta \in \Theta} \left\{ \bigotimes \{\delta((\psi, \sigma \odot \beta), a) \mid \sigma \in \theta\} \right\}, \end{cases}$$

$$\Theta := \mathit{spec}(\beta, a, Q_1 x_1 \dots Q_n x_n : p(u_1, \dots, u_m))$$

A run of the *parametrised automaton* is a tree where nodes are labelled with tuples from $(Q \times [\mathcal{V} \dashrightarrow \mathcal{D}])$. The run is accepting, if all leaves are labelled with tuples where the state component is in F. Note that each leaf may be labelled with a different binding function. The incremental nature of the bindings is still visible in the tree: The valuation of a child node is always *at least as specific* (in the sense that it binds at least the same variables) as its parent.

In [53], a variety of algorithms for checking finite paths (breadth-first, depth-first, and backwards) with alternating finite automata has been presented. Since we are interested in checking a trace as it grows, backwards traversal is not interesting to us. Depth-first search needs to traverse the same trace several times. We will first give an implementation of the single-step evaluation very much like the finite path semantics but based on the automaton construction, and then a breadth-first algorithm.

Definition 69 (Implementation of a parametrised automaton)
For each state in the parametrised automaton $A^{\mathcal{I}}(\phi) = \langle 2^{\mathcal{P}_\perp}, Q, q_0, \delta, F \rangle$, we split evaluation of the current subformula into two steps: firstly, we calculate the *propositional part* of the formula that must be satisfied for the formula to be satisfied in the current state *or in the future*. If the propositional part is refuted, no further evaluation is necessary. Only then we consider the future temporal part of the formula: if it is empty, this means the formula is completely satisfied iff the propositional formula from the previous step was satisfied. If it contains a singleton formula, this formula must either be proved on the remaining trace, or, if this is the last state of the trace, must be in the accepting set of the automaton. It is sufficient to consider only the static part of the automaton, without the bindings, since whether a state is accepting only depends on the formula, and not a specific binding. Existence predicates in the first component indicate that new bindings have to be obtained from the current state.

We will then show that this is just an alternative view on the previous automaton construction. During evaluation, we may need to combine results from different branches of the alternating automaton, as each branch produces an intermediate result. The outer set maintains the distinct run-trees, while the inner sets are conjoined children in a single tree. As a shorthand notation for the following definitions, we introduce

$$C := cl'(\phi)$$

for the static part of the automaton.

$$\begin{array}{rcll}
\mathit{split} & : & C \to 2^{2^{(C\times 2^{C})}} & \\
\mathit{split}(\mathbf{tt}) & := & \{\{(\mathbf{tt}, \varnothing)\}\} & \\
\mathit{split}(\mathbf{ff}) & := & \{\{(\mathbf{ff}, \varnothing)\}\} & \\
\mathit{split}(p(\vec{u})) & := & \{\{(p(\vec{u}), \varnothing)\}\}, \quad p(\vec{u}) \in \mathcal{P} & \text{(propositions)} \\
\mathit{split}(\neg p(\vec{u})) & := & \{\{(\neg p(\vec{u}), \varnothing)\}\}, \; p(\vec{u}) \in \mathcal{P} & \\
\mathit{split}(q(\vec{u})) & := & \{\{(q(\vec{u}), \varnothing)\}\}, \quad q(\vec{u}) \in \mathcal{P}r & \text{(predicates)} \\
\mathit{split}(\neg q(\vec{u})) & := & \{\{(\neg q(\vec{u}), \varnothing)\}\}, \; q(\vec{u}) \in \mathcal{P}r & \\
\mathit{split}(\varphi \vee \psi) & := & \mathit{split}(\varphi) \cup \mathit{split}(\psi) & \\
\mathit{split}(\varphi \wedge \psi) & := & \mathit{split}(\varphi) \otimes \mathit{split}(\psi) & \\
\mathit{split}(\varphi \; \mathbf{U} \; \psi) & := & \mathit{split}(\psi) \cup \mathit{combine}(\mathit{split}(\varphi), \varphi \; \mathbf{U} \; \psi) & \\
\mathit{split}(\varphi \; \mathbf{R} \; \psi) & := & (\mathit{split}(\varphi) \otimes \mathit{split}(\psi)) \cup \mathit{combine}(\mathit{split}(\psi), \varphi \; \mathbf{R} \; \psi) &
\end{array}$$

where

$$\begin{array}{l}
\mathit{combine} : 2^{2^{(C\times 2^{Q})}} \times C \to 2^{2^{(C\times 2^{C})}} \\
\mathit{combine}(s, \phi) := s \otimes \{\{(\mathbf{tt}, \{\phi\})\}\}
\end{array}$$

augments the successor state set by another conjoined formula. This is designed to avoid that any bindings spill from the left-hand side of an Until (right-hand side of Release) into the evaluation of the recurrent part of the formula.

For quantified propositions, we need to collect the current quantification and any quantification on the temporal top-level from the right-hand side (for example, for $\forall x : p(x) \dot{\to} \exists y : q(x, y) \dot{\to} \psi$, where both x and y are quantified in the current state, albeit in two subformulae):

$$\begin{array}{l}
\mathit{split}(Q_1 x_1 \ldots Q_m x_m : p(u_1, \ldots, u_n) \dot{\to} \psi) \\
:= \big\{\{(Q_1 x_1 \ldots Q_m x_m : p(u_1, \ldots, u_n) \dot{\to} \mathit{now}_\psi, \; \mathit{next}_\psi) \\
\qquad \mid (\mathit{now}_\psi, \mathit{next}_\psi) \in \mathit{qs}\} \mid \mathit{qs} \in \mathit{split}(\psi)\big\}
\end{array}$$

In the *split* function for propositions, predicates, and **tt**/**ff** we use an *empty set* as successor state (the second component), indicating that no future obligations have to hold.

By construction, the first component of the result of *split* will always contain singletons, that is **tt**/**ff** or a (negated) proposition unless quantifiers are involved.

For that case, we need a function which descends the chain of existence predicates with quantifiers and *collect*s the resulting sets of bindings, for formulae like above:

$$
\begin{aligned}
& collect : 2^{\mathcal{P}_\perp} \times p\mathrm{LTL} \to 2^{2^{[\mathcal{V} \dashrightarrow \mathcal{D}]}} \\
& collect(a, Q_1 x_1 \ldots Q_m x_m : p(\vec{u}) \dot{\to} \psi) \\
& \quad := \bigcup \{ \bigotimes \{ collect(a, \hat{\sigma}(\psi)) \mid \sigma \in \theta \} \mid \theta \in spec(\beta_\varnothing, a, Q_1 x_1 \ldots Q_m x_m : p(\vec{u})) \} \\
& collect(a, \varphi) := \{\{\beta_\varnothing\}\}, \text{ otherwise (propositional formulae)}
\end{aligned}
$$

Theorem 70 (Equivalence of finite path semantics and successive evaluation)
We obtain a way to check the acceptance of an input $w = w[0] \ldots w[n-1] \in (2^{\mathcal{P}_\perp})^n$ with $w[1{>} = w[1], \ldots, w[n-1]$ based on a propositional formula which is checked against the current state only and obligations on the remainder of the path:

$$
\begin{aligned}
(w, \beta) \models \phi \iff \; & (|w| > 1 \to \exists qs \in split(\phi) : \forall (\varphi, cs) \in qs : \\
& \quad \exists \theta \in collect(w[0], \hat{\beta}(\varphi)) : \forall \sigma \in \theta : \\
& \quad\quad (w[0], \sigma \odot \beta) \models eval^{\mathcal{I}}(\underline{\varphi}) \wedge \forall c \in cs : (w[1{>}, \sigma \odot \beta) \models c) \\
\wedge \; & (|w| = 1 \to \exists qs \in split(\phi) : \forall (\varphi, cs) \in qs : \\
& \quad \exists \theta \in collect(w[0], \hat{\beta}(\varphi)) : \forall \sigma \in \theta : \\
& \quad\quad (w[0], \sigma \odot \beta) \models eval^{\mathcal{I}}(\underline{\varphi}) \wedge \forall c \in cs : (c, \sigma \odot \beta) \in F
\end{aligned}
\qquad (*)
$$

where $\underline{\varphi}$ removes the quantifiers from a chain of nested existence predicate if present (no name clashes can occur since we required the input formula to use fresh variables for every quantification):

$$
\begin{aligned}
\underline{Q_1 x_1 \ldots Q_m x_m : p(u_1, \ldots, u_n) \dot{\to} \psi} \; &:= \; \underline{\psi} \\
\underline{\psi} \; &:= \; \psi, \text{ otherwise} \\
& \quad\; \text{(propositions, predicates, } \mathbf{tt}/\mathbf{ff}\text{)}
\end{aligned}
$$

We remind the reader that the test $(c, \sigma \odot \beta) \in F$ above can be answered by simply looking at the static component c.

Proof:

We prove the equivalence of successive evaluation **(*)** with regard to the finite path semantics (Definition 37) for the end of the path, that is, for a path of length 1.

Singleton Path $|w| = 1$

1. Boolean formulae ($\mathbf{tt}, \mathbf{ff}$) correspond in both semantics as they remain untouched in the first component of *split*.
2. A (negated) *unquantified* proposition or predicate is completely instantiated through β ($\sigma = \beta_\varnothing$ as there are no new bindings) by construction and evaluated in both semantics through the non-temporal rules: in the stepwise evaluation, this means *split* will move them into the first component, while the second component will be accepted by default since it is empty.

3. Disjunction and conjunction are split into their respective DNF form by *split* and evaluation proceeds in both branches recursively, just like in the finite path semantics.

4. $\varphi\ \mathbf{U}\ \psi$ succeeds at the end of the path in the finite path semantics iff $w[0] \models \psi$ since there is no $l < k$ for $k = 0$. This implies that during evaluation ψ must be satisfied which is reflected in the recursive invocation of $split(\psi)$. Since the second component of an **U**-formula will always contain the same **U**-formula in the *combine*d part, it can never be an accepting configuration as it is not contained in F.

5. A similar argument holds for $\varphi\ \mathbf{R}\ \psi$, where at the end of the path acceptance of ψ is sufficient due to the fact that the second component in each *combine*d subformula will be an accepting **R**-state in F.

6. If ϕ is a quantified existence predicate, recursively for each subformula of the right-hand side a separate instance prefixed with the quantification is generated in the first component of *split*; the chain of existence predicates is then traversed by *collect* to accumulate the corresponding bindings based on the current state. The finite path semantics traverses the chain in the same order. The ground propositional formula in the first component is evaluated, and any temporal formula stemming from the right-hand side of ϕ in the second component is checked against the set of accepting states.

Path $w = w[0]w[1] \ldots w[n-1]$

1. Boolean formula $\mathbf{tt}/\mathbf{ff}$: trivial result from *split*; *collect* returns the identity valuation; the successor relation is trivially satisfied by universal quantification over the empty set and thus corresponds to the finite path semantics.

2. (Negated) unquantified propositions/predicates do not contribute new bindings in *collect*; same argument as for $\mathbf{tt}/\mathbf{ff}$ after the state-part in the first component of *split* is satisfied.

3. The Boolean operations generate distinct branches in *split*, for which the new valuations are collected from the propositional part. As different branches represent different, disjoint namespaces for quantified variables, they are both checked independently (*collect* is applied separately to each branch).

4. For an Until, we obtain two components: the first one is a set of obligations that might satisfy the right-hand side, the second one tries to prove the left-hand side with the additional implication of having to prove the recurrent formula from the next state on (second component of *split*). Due to the introduction of the separate clause with the $\mathbf{tt}$ in the first component of *split*, no new variables will be instantiated and thus the recurrent formula will be proved with unchanged valuation.

5. A similar argument holds for Release, where proof obligations for satisfaction on the current state are conjoined, each with its own bindings derived from the current state. As for Until, the recurrent part is in a separate disjunct with unpolluted bindings.

6. Quantified propositions on "temporal top level", that is, in the first component of *split*, contribute their bindings through the first and the last rule of *collect* in the same way that the declarative semantics for a propositional formula does (like in the previous argument for a singleton path). Also, any non-temporal components are accumulated and evaluated with respect to the bindings.

   Future behaviour can only have been contributed by the three temporal operators above and is distributed between the two components accordingly.

This concludes our proof of correctness of the operational semantics. Algorithm 1 gives the breadth-first algorithm derived from the operational semantics for checking a path against a pLTL formula. Before proceeding with a sample application, we point out that the *split* function does not depend on the actual input and can thus be precomputed and the results stored in a table for quick lookup through the algorithm.

Algorithm 1 Parametrised automaton, breadth-first search up to last state

Check input trace w breadth-first against a positive pLTL formula ϕ by keeping a list of configurations.

```
config := {{(q0, β0)}}; i := 0
while i < |w| and config ≠ ∅ do
  tempConfig := ∅
  for each S ∈ config do
    S' := {∅}   // all successors of a state
    for each (q, β) ∈ S do
      T := ∅
      for each qs ∈ split(q) do
        t' := {∅}
        for each (φ, cs) ∈ qs do
          Θ := collect(w[i], β̂(φ))
          if Θ = ∅ then
            t' := ∅; break   // refuted: no bindings at all
          else
            t := ∅   // conjunction
            for each θ ∈ Θ do
              if θ ≠ ∅ then
                temp := ∅
                for each σ ∈ θ do
                  if ([i], σ ⊙ β) ⊨ eval^I(φ̲) then
                    temp := temp ∪ {(c, σ ⊙ β) | c ∈ cs}
                  else
                    temp := ∅; break   // refuted: bindings no good
                  end if
                end for  // end for σ
                if temp ≠ ∅ then
                  t := t ∪ {{temp}}
                end if
              else
                t := {∅}   // ∀ satisfied by absence
              end if
            end for  // end for θ
            t' := t' ⊗ t
          end if
        end for  // end for (φ, cs))
        T := T ∪ t'
      end for  // end for qs
      S' := S' ⊗ T
    end for  // end for (q, β)
    tempConfig := tempConfig ∪ S'
  end for
  config := tempConfig; i := i + 1
end while
acceptFinal(config)
```

Algorithm 2 `acceptFinal` : $2^{2^{(\mathtt{C},[\mathcal{V} \dashrightarrow \mathcal{D}])}}$: evaluation of parametrised automaton, on the last state only

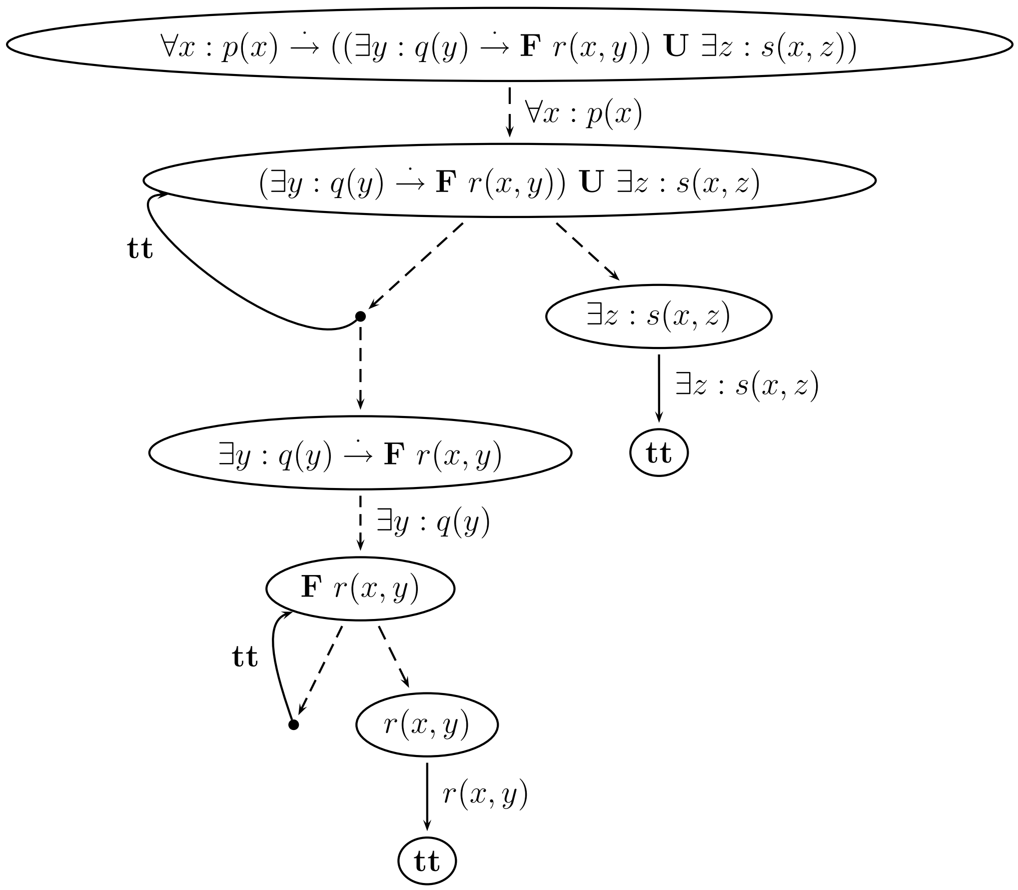
$T := \mathbf{ff}$ // input non-empty?
if $i > 0$ **then**
  for each $S \in$ *config* **do**
    $T' := \mathbf{tt}$
    for each $(q, \beta) \in S$ **do**
      $T' := T' \wedge ((q, \beta) \in F)$ // accepting state?
    end for
    $T := T \vee T'$ // at least one accepting configuration?
  end for
end if
return T

Example 71 (Run of parametrised automaton)
To illustrate the behaviour of the implementation, we take a look at an example.

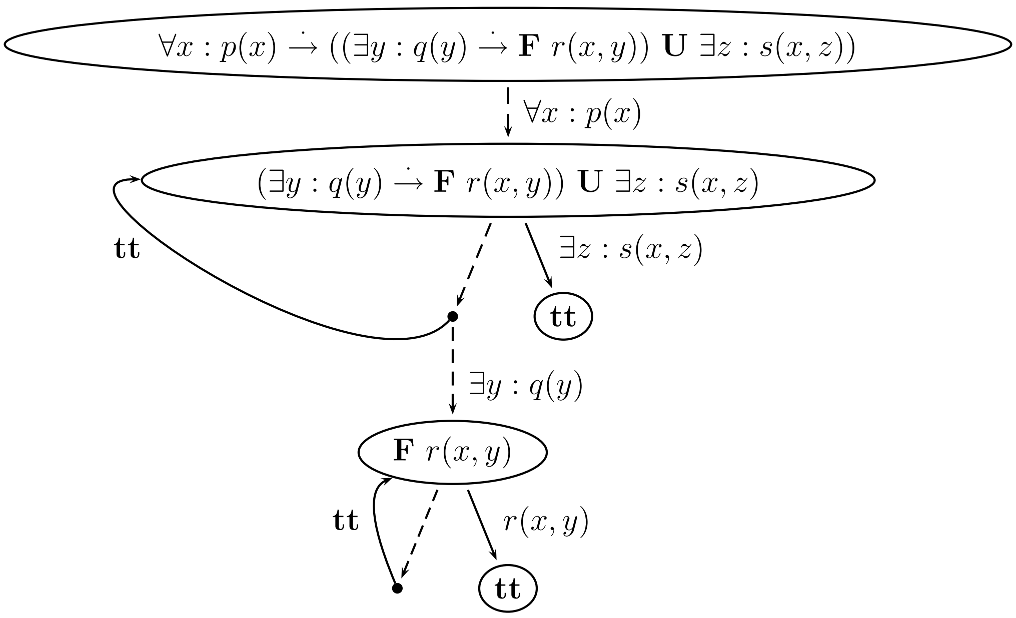
$$\forall x : p(x) \mathrel{\dot{\rightarrow}} ((\exists y : q(y) \mathrel{\dot{\rightarrow}} \mathbf{F}\ r(x,y))\ \mathbf{U}\ \exists z : s(x,z))$$

We shall construct the static part of the parametrised automaton as it will be used by the *split* construction graphically: edges will be labelled with (quantified) propositional formula from the first component of *split* to indicate under which (abstract) input the edge is taken. Without formal introduction, we will use dashed ϵ edges which correspond to those steps in the construction in the *split* function where "no time passes", that is, where recursive invocations of *split* on existence predicates are involved. To bring across the point that a specific sub-automaton should only be executed under a specific binding obtained through quantification, we shall add such information also to ϵ (dashed) edges. It serves only as a reminder that any concrete edges should be conjoined under that quantification. They correspond to an element in a chain of quantifications in the first component of *split* (see initial edge from the root to the first node for an example). These edges will be eliminated subsequently so that successor states can be looked up with a single comparison.

If an edge is labelled with **tt**, it shall be taken under any input. To keep the graph readable, we use several final **tt** nodes, although in practice there is only one.

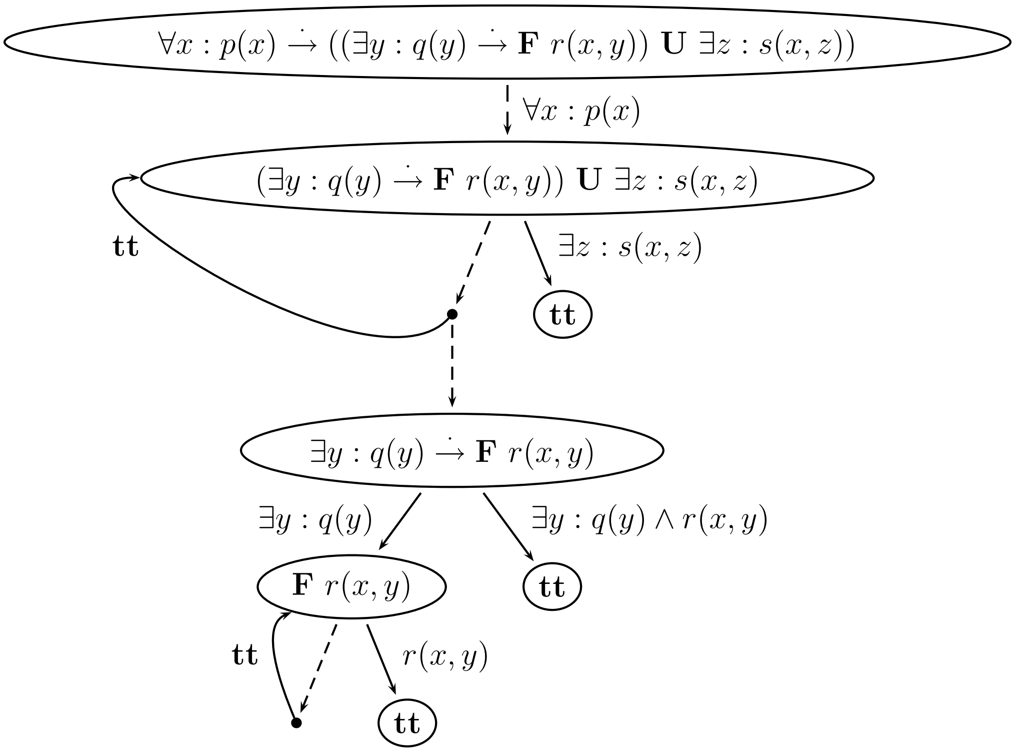


Now we can collapse intermediate nodes which are only pointed to by an unlabelled ϵ edge with their parent:

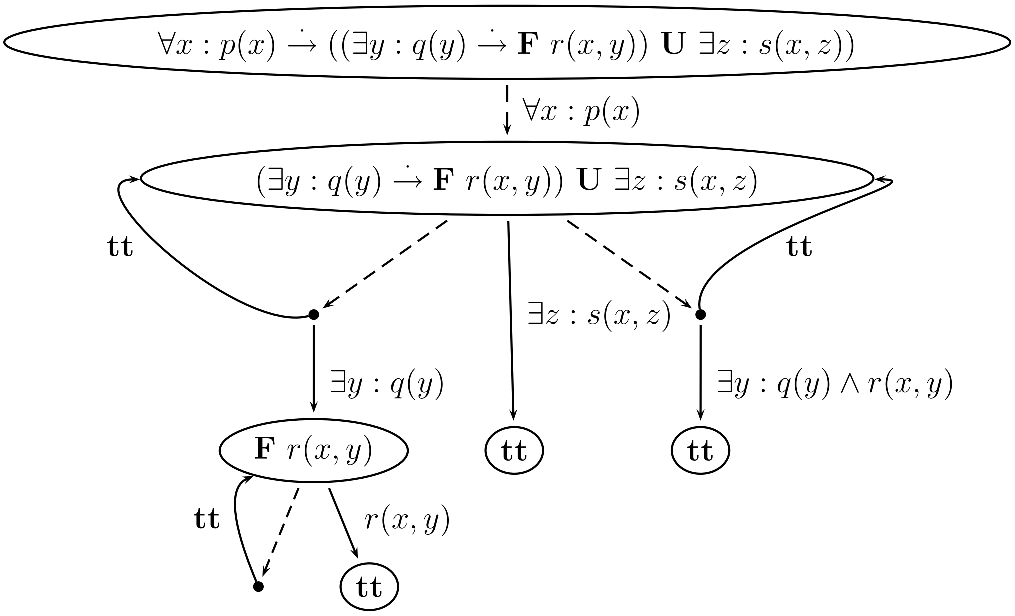


The resulting automaton clearly shows how the ordering of quantifiers in the formula is preserved: on descent into a subformula via a labelled ϵ edge, the value is bound and only available to the subformula. Any self-loops, which are by construction the only edges not pointing to a sub-formula of the current node label, stem from application of the *combine* function.

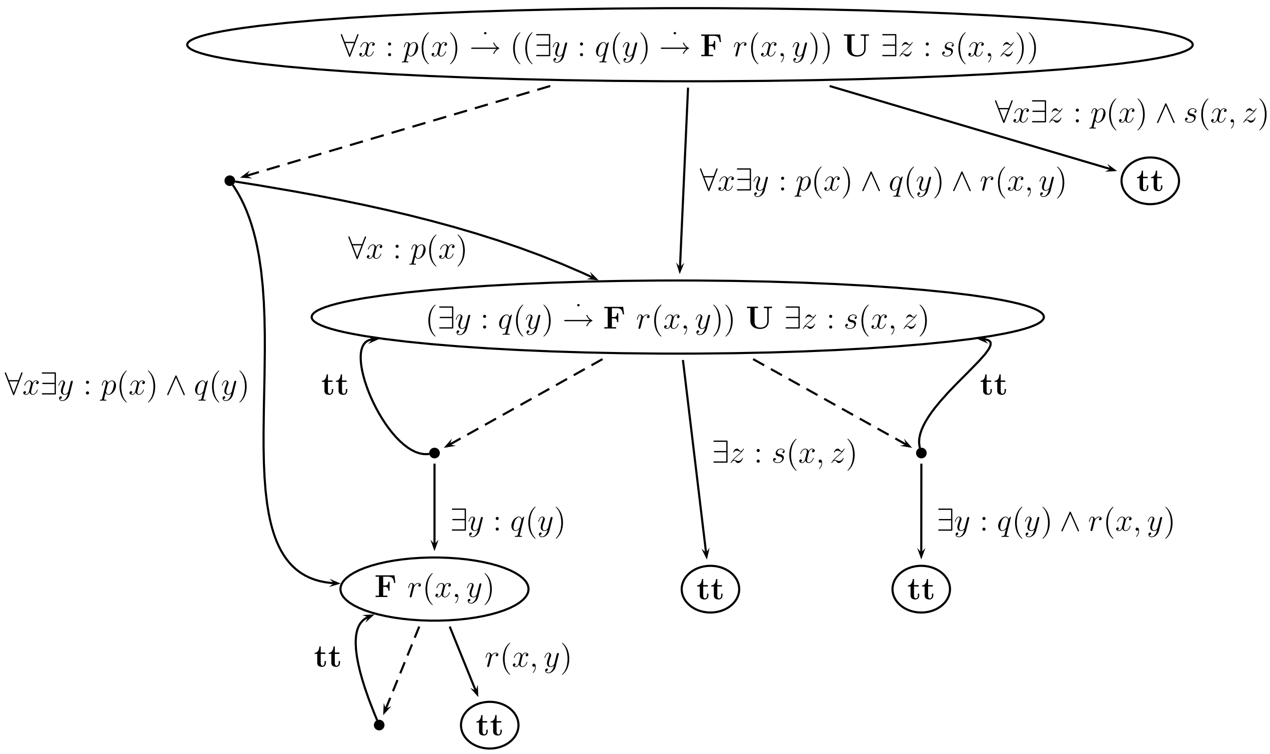
Next, we eliminate the labelled ϵ edges successively, starting from the bottom. This step basically corresponds to the combination of conjuncts in the definition of the transition function. We start with the edge leading into the Finally node. We unroll the subtree once, propagating the quantified formula into the leaves:



Next, we eliminate the unlabelled ϵ edge leaving the conjunction below the *Until* node. Since the node below has two alternative (disjoint!) branches, they propagate into the preceding Until node:



The only (labelled) ϵ edge to eliminate is now the one from the root node, which we eliminate in a similar fashion.



Observe that *all* ϵ edges have now been removed (except those we chose for depicting conjoined nodes). This allows us to determine the successors of each node with a simple lookup. Note that the propositional formulae on disjoint branches originating from the same node may still overlap.

We now wish to evaluate the formula on the trace

$$w := \{p(1), q(1)\}; \{q(2)\}; \{s(1,2)\}; \{r(1,1), r(1,2)\}.$$

In the following, we will refer to the three non-trivial states of the automaton as q_0, q_1, and q_2, numbering them top-down. The automaton starts in its initial configuration s_0, that is, the start state q_0 with the empty binding. As we will generally work with the power-set structure of alternating automata and their run trees, we already denote this configuration as a singleton conjunct:

$$s_0 : \{\{(q_0, \beta_0)\}\}, \text{ where } \beta_0 = \beta_\varnothing$$

For the first state $w[0] = \{p(1), q(1)\}$, only the first (left-most) edge can be taken, leading into the configuration

$$s_1 : \{\{(q_1, \beta_1), (q_2, \beta_2)\}\}, \text{ where } \beta_1 = \{x/1\} \odot \beta_0, \text{ and } \beta_2 = \{x/1, y/1\} \odot \beta_0$$

which, when interpreted again as formula reads

$$[(\exists y : q(y) \dot{\rightarrow} \mathbf{F}\ r(1,y))\ \mathbf{U}\ \exists z : s(1,z)] \wedge \mathbf{F}\ r(1,1)$$

Note that y does not occur free in q_1, we will see in the next step with $w[1] = \{q(2)\}$ that the desired behaviour is captured by the new binding $y/2$.

$$s_2 : \{\{\overbrace{(q_1, \beta_1), (q_2, \beta_3)}^{\text{successors of } q_1}, \underbrace{(q_2, \beta_2)}_{\ldots\text{of } q_2}\}\},\ \beta_3 = \{y/2\} \odot \beta_1 = \{x/1, y/2\}$$

With $w[2] = \{s(1,2)\}$, we will satisfy the $\exists z : s(1,z)$ for $\{z/2\} \odot \beta_1$. Since the **tt** true state in an otherwise nonempty subformula is redundant according to the semantics, we prune this configuration. Remember that alternating automata for the finite path semantics do not need any exiting edges from a **tt** state to accept a prefix of the trace. The remaining states are stationary due to **tt** edges:

$$s_3 : \{\{(q_2, \beta_3), (q_2, \beta_2)\}\}$$

The last state in the trace will then validate the formula: both pending Finally states are satisfied and pruned:

$$s_4 : \{\varnothing\}$$

If the trace would have ended after $w[3]$, the configuration would inform us that two eventualities in s_3 with their respective bindings have not been fulfilled.

3.6 Alternative Approaches

Büchi Automata and Finite Paths

Model Checking is usually concerned with checking ω-regular properties [63] on paths in a Kripke structure. Such a structure is a state-labelled transition system with directed edges. LTL formulae represent the star-free, non-counting subset of ω-regular languages [119, 120].

A substantial complexity of model checking comes from the exponential overhead of converting the alternating finite automaton for the negated formula into a (generalised) Büchi automaton which accepts as input infinite words, and then computing the product of the languages recognised by the two automata to check it for emptiness.

This almost naturally raises the question whether Büchi automata would provide a viable solution for the problem of checking *finite* paths. [60] reports on an optimised construction of nondeterministic automata from LTL formulae that is based on a well-known tableau-like translation [59]. We also considered whether automata generated by the popular tool `ltl2ba` by Gastin and Oddoux [58] could be used. It turned out that for infinite words, the exact position of an accepting state of the automaton in the cycle of a “lasso” does not matter, since *any* automaton state in the cycle will be visited infinitely often.

However, for a finite run, we need to know exactly whether we are on an accepting state or not. In the case of `ltl2ba`, we found that especially for formulae containing the explicit Next operator this was a problem. Also, we note that different tools place the accepting states differently for the same LTL formula as has been observed by others before. Other peculiarities of trying to use Büchi automata on finite paths are detailed in [77].

We conclude that on one hand we would either have to find an existing algorithm to convert LTL formulae to Büchi automata which exhibits the desired behaviour with regard to accepting states, or develop our own. On the other hand, we have seen that directly using alternating finite automata does not have any particular problems in the general case.

Algebraic Alternating Automata

Finkbeiner, Sankaranarayanan, and Sipma proposed *algebraic alternating automata* [52] (inspired by *extended alternating automata* used in *Query Checking* [22, 24]), which collect statistics over runtime executions. Their framework allows to evaluate queries over finite traces like "what is the average number of retransmissions" or "what is the maximum packet delay". Basic observations on the trace constitute experiments, which are then aggregated with the help of algebraic alternating automata.

Each symbol in a formula gets an additional function corresponding to its arity, that is, propositions and the **X** operator carry a unary function, Boolean functions $\vee$, $\wedge$, and **U** a binary function. These functions are used to combine results from both branches. Similarly, for aggregate statistics *interval* and *unconditional collection* can be used to compute, for example, the maximum number of times that a predicate holds in a specific interval.

The collection mechanism of this framework could surely be used to "accumulate" the variable bindings used in our approach instead of some statistics. Unfortunately, the overall approach makes this rather impractical, since the overall result does not influence the acceptance condition of the automaton, as our mechanism does. Acceptance is still defined through the standard semantics of alternating finite automata, and the statistics are rather a byproduct. This means that there is no way to shortcut evaluation—additionally to the trace of variable bindings, a separate pass for checking the input with respect to such a binding would be required. This could be achieved *inline* by crafting special functions for each position in the alternating automaton which include some knowledge about their position in the formula to produce a Boolean decision along the same lines as *succ* from Section 3.5.2. We have not investigated this approach in detail because of this unorthogonalness and the necessary influence of the aggregate on the acceptance of the automaton. Nevertheless, we feel that nothing should impede a joint implementation of our parametrised framework with the collection of statistics.

Alternatives to LTL

In [74], an algorithm that generates efficient dynamic programming algorithms from *past time* LTL formulae is presented. They test whether the formula is satisfied by a finite trace of events and run in linear time. The constant factor depends on the size of the LTL formula. Memory consumption is constant and also depends only on the formula size.

It also offers a few non-standard operators which the authors found useful like a property *ending* or *starting* in the current state, which largely contribute to the conciseness of formulae. The algorithm was implemented in the Java PathExplorer architecture [71].

However, there seems to be a common agreement that generally future time LTL is much more suitable for human specification and usually the option of choice. Other formalisms like past time LTL are then only provided as an add-on.

It has also been noted in [56] that past time operators do not contribute additional expressiveness to a future time-only logic, although formulae are more succinct. Algorithms for eliminating one kind of temporal operators are known since [55], and recently complexity results and the exponential cost for certain properties when only future or past are available have been well studied [94, 95].

In general, we do not expect any difficulties in adapting our mechanism to past time LTL while profiting from the benefits mentioned above.

Of course it is arguable whether LTL in general is an appropriate specification language. Especially for novices, a more familiar syntax might be provided by regular expressions or even a graphical specification language. The former has been pursued by the Tracematch implementation [2] that we extensively compared our Tracechecks formalism [19] with. As the underlying framework will usually resemble a finite automaton, it should be possible to devise frontends that map quantification and instantiation into our framework. Also, [45] provides a survey of specification patterns and provides a comprehensive guide on how to express different properties in varying formalism.

It should be noted that some properties, like for example, that an event has to happen every n steps cannot be expressed in temporal logic but with regular expressions [127]. Parametrised regular queries have recently been discussed in [91].

In the static verification community, several other specification languages like SUGAR [15] and FORSPEC [5] exist, which also contain additional syntactic sugar hiding the temporal logics in the semantic layer to make the input languages more user-friendly.

4 Evaluating Parametrised Formulae at Runtime

In this chapter, we tie the knot between the operational model of a program that generates a trace and the verification of pLTL formulae on said trace. First, we will discuss how to map the event trace into a path for the runtime checker. While there exists an apparent one-to-one correspondence, we will look into filtering the trace with respect to a *given signature*, which could, for example, be derived from the propositions in the formula. The filtering can influence the evaluation of formulae, especially with respect to triggering redundant evaluation in the automaton.

4.1 Trace Extraction

Let's look back at the operational trace semantics of a $\mathbb{PL}_{Int}$ program π from the Definition 19:

$$\texttt{Exec}_{\pi}^{\rho,\mathcal{I}} : Dom^n \to \textsc{Evt}^*$$

for some value domain Dom, and $\mathcal{I} = \langle Dom, \xi \rangle$ some interpretation. Although the trace is the final result of the execution of a program, it is easy to see from our definition of the iteration semantics that the trace is a monotonically growing list. That is, it grows with each instruction, and already generated items on the trace are never modified.

On the one hand, this allows us to consume the trace in a *lazy* fashion, where we read the trace incrementally as it is generated. On the other hand, this does not inhibit us from considering the *entire trace*, for example, in an offline checking setup. Consequently, we will not model this formally, but rather assume that the trace is accessible to us in a subsequent fashion.

Filtering Paths

Recall from Section 2.7 that an item on the trace will either be a τ action or an event. Thus, we can literally take an event as a fully instantiated parametrised proposition. Because object identifiers occur in events, the domain of the logic must encompass both the values of the program domain and the object identifiers:

$$\begin{aligned}
\mathcal{PN} &:= \{\texttt{method enter}^{(5)}, \texttt{method exit}^{(6)}, \texttt{set}^{(5)}, \texttt{get}^{(5)}, \texttt{new}^{(4)},\\
&\qquad \texttt{assert}^{(4)}, \texttt{lock}^{(3)}, \texttt{unlock}^{(3)}\}\\
\mathcal{D} &:= ODom = Dom \cup \mathcal{O}
\end{aligned}$$

Observe that some mapping of thread identifiers, classes, methods, and attributes into the underlying domain must be provided. They will usually be mapped to strings or integers.

Since only one event may happen at a time, this naturally translates to a state with only a singleton set labelling. The τ action is mapped to the empty set of propositions. It is clear that this model has the following basic properties:

- a state is either labelled with the empty set, or exactly one proposition,
- each event corresponds to a program state transformation which is visible either in the control flow as a method call/return, or in the heap,
- the model may contain proposition names which do not occur in the formula.

This mapping from events to states has an effect on the evaluation of formulae with respect to quantifiers: as for each state, there exists *at most one valuation* for any variables in a parametrised proposition in the formula, universal and existential quantification coincide for singleton states. Due to the way how the quantifiers differ, however, handling of empty sets induced by a τ action is distinct: a subformula with a primary universal quantifier will use the short-cut rule from Definition 66 and default to accept, while it would fail in the presence of an existential quantifier.

If by looking at the formula, we can determine that some events (for example, τ actions) do not have any effect on the evaluation of a formula, we can speed up the computation by telling the framework that we are not interested in some events and save ourselves some work by filtering them out from the trace. Also, by eliminating as many "superfluous" states as possible from the model, we can cut down the memory consumption of a trace if we can trim non-relevant states.

When checking a specific property, we observed that the formula is often invariant to propositions with constructors that do not occur in the formula at all (but could nonetheless occur in the trace).

For example, formulae may prohibit future behaviour through $\mathbf{G}\ \neg p(x_1, \ldots, x_n)$ ("Never"). Or, they require some eventuality which is specified through a singleton positive proposition or a set of positive propositions: $\mathbf{F}\ p(x_1, \ldots, x_n)$. We invite the reader to consider what the effect of the dual behaviour is on program traces, that is, requiring some event in every state through $\mathbf{G}\ q(x_1, \ldots, x_n)$, or accepting absence of some event $\mathbf{F}\ \neg q(x_1, \ldots, x_n)$. Although these are surely valid formulae, we think that they are not very common on program traces where only a single proposition holds at most in each state, which is certainly confirmed by our examples.

We observe a similar effect for more general formulae with Release or Until operators, which is to be expected as Globally and Finally are special cases of Release and Until after all: "something (bad) does not happen until a certain event" is expressed through $\neg p(x_1, \ldots, x_n)\ \mathbf{U}\ q(y_1, \ldots, y_m)$. The dual Release usually has the form $p(x_1, \ldots, x_n)\ \mathbf{R}\ \neg q(y_1, \ldots, y_m)$. As above, any propositions except those occurring in the formula do not influence the result as they are subsumed by the negated propositions: the negated propositions guard the recurrent formula which defers evaluation until the next step, while the positive propositions indicate the successful evaluation.

We now generalise this concept through signatures and give the user the opportunity to influence the set of propositions which will occur during evaluation of a formula.

Definition 72 (Signature of a formula)
The *signature of a* pLTL *formula* ϕ is defined through the set of used proposition names:

$$\begin{aligned}&S : p\mathrm{LTL} \to 2^{\mathcal{PN}}\\&S(\phi) := \{p \in \mathcal{PN} \mid \exists p(u_1, \ldots, u_n) \in cl'(\phi)\}\end{aligned}$$

Definition 73 (Consistent filtering function)
Given a *filtering function* $\lambda_e : \textsc{Evt}^* \to (2^{\mathcal{P}_\perp})^*$ that translates a stream of events into a stream of ground propositions over $ODom$, we call the filtering function λ_e and a formula ϕ *consistent*, iff

$$\forall p \in S(\phi) : \exists x \in \textsc{Evt} : \lambda_e(x) = p(d_1, \ldots, d_n), d_1, \ldots, d_n \in ODom.$$

Otherwise, some propositions would never be able to occur at all in the trace, which usually indicates some problem in the specification.

Note that the input and output streams do not necessarily have to have the same length. For example, this function may choose to eliminate τ steps completely from the trace.

Definition 74 (Runtime verification of temporal assertions)
Given a program $\pi = \langle \mathcal{C}_\pi, args\rangle$, an interpretation $\mathcal{I} = \langle ODom, \xi\rangle$ for functions and predicates (although a formula does not actually have to use them), a consistent filtering function λ_e, and a positive pLTL formula ϕ over $\mathcal{P}_\perp$, we state the following:

A program π satisfies a temporal specification ϕ, if

$$(\lambda_e \circ \mathtt{Exec}_\pi^{\rho,\mathcal{I}})(args) \models \phi.$$

In the following, we give a sample mapping, although other mappings are possible:

Definition 75 (Mapping)
We define the mapping of a trace through the following domain

$$\begin{aligned}\mathcal{PN} &:= \{\mathtt{method\ enter}^{(5)}, \mathtt{method\ exit}^{(6)}, \mathtt{set}^{(5)}, \mathtt{get}^{(5)}, \mathtt{new}^{(4)},\\&\quad\ \mathtt{assert}^{(4)}, \mathtt{lock}^{(3)}, \mathtt{unlock}^{(3)}\}\\\mathcal{D} &:= ODom\end{aligned}$$

and the filtering function

$$\begin{aligned}\lambda &: \textsc{Evt} \to 2^{\mathcal{P}_\perp}\\\lambda(\tau) &:= \varnothing\\\lambda(ev) &:= \{ev\},\ \text{otherwise}\\[1ex]\lambda_e(w[0]; \ldots; w[k]) &:= \lambda(w[0]) \ldots \lambda(w[k])\end{aligned}$$

Remark 76
Let's recall the stack example from the introduction to Chapter 2: there must never be an attempt to take an item from the initially empty stack via `Stack.pop` until at least one item has been pushed onto it through `Stack.push`. (Note that LTL is not powerful enough of checking the more general, but context-free, property of never taking an item from *any* empty stack.) We can express this property through the following filtering function and formula:

$$\lambda(ev) := \begin{cases} \{\texttt{push(s,x)}\}, \text{ if } ev = \texttt{method exit}(tId, \texttt{this}, Stack.push, s, x, r) \\ \{\texttt{pop(s)}\}, \text{ if } ev = \texttt{method enter}(tId, \texttt{this}, Stack.pop, s) \\ \varnothing, \text{ otherwise} \end{cases}$$

$$\lambda_e(x; w) := \lambda(x); \lambda_e(w)$$

$$\phi := \forall s \exists x : \texttt{push(s,x)} \ \mathbf{R} \ \neg\texttt{pop(s)}.$$

An alternative which illustrates the interaction between filter and formula would be the following mapping where we only employ a unary `push` proposition

$$\lambda(ev) := \begin{cases} \{\texttt{push(s)}\}, \text{ if } ev = \texttt{method exit}(tId, \texttt{this}, Stack.push, s, x, r) \\ \{\texttt{pop(s)}\}, \text{ if } ev = \texttt{method enter}(tId, \texttt{this}, Stack.pop, s) \\ \varnothing, \text{ otherwise} \end{cases}$$

with the formula

$$\phi := \forall s : \texttt{push(s)} \ \mathbf{R} \ \neg\texttt{pop(s)}.$$

Since it does not matter which element exactly is pushed onto the stack (elements are existentially quantified), we might as well abstract it already in the filter instead of binding it explicitly in the formula, if it is not going to be used deeper in the formula, anyway. This does not reduce the number of evaluation steps, but eliminates a variable binding.

Next, we present an example for a concurrency property of a program. It joins all components we have visited so far, that is, the program, the specification in form of a pLTL formula, and a filtering function.

4.2 Example: Lock-order Reversal

As an example for a refutation, we shall consider an actual concurrent programming problem we reported in [117]: to avoid the problem of Lock-order Reversal (LOR) (see also [68, 16]), we would like to assert through an LTL formula that if two locks are taken in a given order (with no unlocking in between), the system should warn the user if he also uses these locks in swapped order because in concurrent programs this would mean that two threads could deadlock when their execution is scheduled in an unfortunate order.

```
class Main

method main
 var lockA lockB

 lockA := new ()
 lockB := new ()
 thread this.take(lockA,lockB)
 this.take(lockB,lockA)
```

```
method take(l1,l2)

L1: lock l1
    lock l2
    /* critical section */
    unlock l2
    unlock l1
    jmp L1
```

Figure 4.1: Sample code exhibiting potential Lock-order Reversal

It should be noted that this warning might not be adequate as it is always possible to construct a program which takes the locks in reverse order and will never deadlock. This can be achieved by adding another lock, a so-called gate lock, to the program which assures mutual exclusion and prevents the system from deadlocking. An algorithm which avoids this kind of false warnings in the presence of gate locks is detailed in [68].

This particular technique of noticing inadvertent lock-reversal at runtime is employed in the development phase of the recent FreeBSD operating system kernel under the name of *witness* [10]. In [68] the same technique is used in the Java PathFinder [125], a model checker for JAVA applications.

We consider two concurrent threads competing repeatedly for two locks A and B. It is not possible to acquire several locks in one go, so both locks have to be obtained sequentially. If we assume that the first thread tries to obtain them in the order A, B and the second one in B, A, we can see that sooner or later the system ends up in a state where thread 1 holds lock A and waits for B to become available while thread 2 holds B waiting on A! This situation is exactly the circular holding pattern which indicates a deadlock (see Chapter 7 of [110]). In small programs the bug might be easy to spot. For larger applications we can conclude that it is very hard to prevent this error from simply perusing the source code. Even when testing the program the error might not occur for various runs. For example it may be possible that threads 1 and 2 are never executed interleaved but rather in a sequential manner, thus giving no rise to the erroneous behaviour at all.

The trace data we will be interested in are just the `lock` and `unlock` operations (refer to Section 2.7 for the propositions available to our framework). We need both the information as to *which lock* is affected and *which thread* is taking the action.

The program in Figure 4.1 will generate a trace containing the above propositions, possibly in the order indicating the erroneous behaviour. We can detect this and warn the developer that his application has the potential to enter a deadlock under certain conditions.

Notice that we do not want to abort the execution in this example: we are here also interested in mere warnings, as a violation of the formula might not coincide with a deadlock. To observe the behaviour of the whole execution path (of which the erroneous behaviour might only be a sub-path), we wrap the formula into the temporal Globally.

Thus, we obtain for two threads t_i, t_j and two locks l_x, l_y the following formula (the `this` component of the events is not interesting to us and therefore replaced by the wildcard _ to match anything, which can be considers as a fresh unique instantiation of an existentially quantified variable):

$$\begin{aligned}
&\mathbf{G}\ [\forall t_i \forall l_x : \mathtt{lock}(t_i, _, l_x) \dot{\rightarrow} ([\neg \mathtt{unlock}(t_i, _, l_x)\ \mathbf{U}\ \exists l_{z'} : \mathtt{lock}(t_i, _, l_{z'})] \\
&\quad \rightarrow [\neg \mathtt{unlock}(t_i, _, l_x)\ \mathbf{U}\ \exists l_z : \mathtt{lock}(t_i, _, l_z) \dot{\rightarrow} \\
&\qquad \forall l_y : \mathtt{lock}(t_i, _, l_y) \dot{\rightarrow} \mathbf{G}\ \neg(\exists t_j : \mathtt{lock}(t_j, _, l_y) \dot{\rightarrow} \\
&\qquad\quad (\neg \mathtt{unlock}(t_j, _, l_y)\ \mathbf{U}\ \mathtt{lock}(t_j, _, l_x)))])],\ \ t_i \neq t_j, l_x \neq l_y, l_z \neq l_x, l_z' \neq l_x
\end{aligned}$$

The formula has several parameters: two locks and two threads. The lock l_z is necessary because there cannot be any information transfer from the right-hand side of an Until to any "subsequent" formula in pLTL. We use an implication to rebind the same event, but now to the proposition with the variable l_y. For this formula, $\exists l_z : \mathtt{lock}(t_i, \_, l_z)$ and $\forall l_y : \mathtt{lock}(t_i, \_, l_y)$ will always coincide[1] .

Using the conventional approach, we would have to pre-generate all possible formula instantiations. This pre-generation could either be based on a fixed maximum number of locks and threads, or on lock/thread creation at runtime. Both approaches have drawbacks. The first one will obviously miss anything with values larger than the specified upper bounds. Also, additional abstractions might be required since for many programming languages, the object identifiers will be arbitrary non-monotonic 32 or 64 bit values. The latter requires additional instrumentation of object creation and the current trace, which for memory reasons will usually be limited to only a suffix of the run.

Take a close look at the formula: once t_i, l_x, and l_y have been bound and evaluation proceeds to the "Never" ($\mathbf{G}\ \neg \ldots$), we have partially instantiated the right-hand side for the remainder of the formula. Only the thread id of the concurring thread remains free.

The constraints on the identifiers are *predicates* (see Section 3.3), which, if this formula is going to be used in practice, need to be moved to more appropriate places. Their position in the formula should be immediately after the respective event which will cause the variables mentioned in the constraint to become fully instantiated. That is, the $l_z \neq l_x$ should be integrated with the event binding the l_z (respectively, the constraint for l_y and t_j). With the appropriate filter function (see Figure 4.2) which also eliminates any states which do not relate to locking we obtain:

[1] In the J-LO implementation, this was not necessary, see the formula we reported in [115].

$$\lambda(ev) \;:=\; \begin{cases} \{\texttt{lock}(t, l_o)\}, \text{ if } ev = \texttt{lock}(t, \texttt{this}, l_o), \\ \{\texttt{unlock}(t, l_o)\}, \text{ if } ev = \texttt{unlock}(t, \texttt{this}, l_o), \\ \varnothing, \text{ otherwise} \end{cases}$$

$$\lambda_e(ev; w) \;:=\; \begin{cases} \lambda(ev); \lambda_e(w), \text{ if } \lambda(ev) \neq \varnothing \\ \lambda_e(w), \text{ otherwise} \end{cases}$$

Figure 4.2: Filter function for Lock-order Reversal

$$\begin{aligned}
&\mathbf{G}\ [\forall t_i \forall l_x : \texttt{lock}(t_i, l_x) \dot{\rightarrow} ([\neg\texttt{unlock}(t_i, l_x)\ \mathbf{U}\ \exists l_{z'} : \texttt{lock}(t_i, l_{z'}) \dot{\rightarrow} l_{z'} \neq l_x] \\
&\quad \rightarrow \neg\texttt{unlock}(t_i, l_x)\ \mathbf{U}\ \exists l_z : \texttt{lock}(t_i, l_z) \dot{\rightarrow} [l_z \neq l_x \\
&\qquad \wedge \forall l_y : \texttt{lock}(t_i, l_y) \dot{\rightarrow} (l_y \neq l_x \wedge G\ \neg(\exists t_j : \texttt{lock}(t_j, l_y) \dot{\rightarrow} [t_i \neq t_j \\
&\qquad\quad \wedge (\neg\texttt{unlock}(t_j, l_y)\ \mathbf{U}\ \texttt{lock}(t_j, l_x))]))])]
\end{aligned}$$

A sample run shall illustrate how the formula behaves. Before proceeding, we will assign names to the different subformulae so that we can refer easier to a configuration in the evaluation. We will also push down the negation to obtain positive normal form (see Definition 25):

$$\begin{aligned}
\Psi &:= \mathbf{G}\ [\forall t_i \forall l_x : \texttt{lock}(t_i, l_x) \dot{\rightarrow} (\varphi^{\mathbf{R}}(t_i, l_x) \vee \varphi^{\mathbf{U}}(t_i, l_x))] \\
\varphi^{\mathbf{R}}(t_i, l_x) &:= \texttt{unlock}(t_i, l_x)\ \mathbf{R}\ \forall l_{z'} : \texttt{lock}(t_i, l_{z'}) \dot{\rightarrow} \neg(l_{z'} \neq l_x) \\
\varphi^{\mathbf{U}}(t_i, l_x) &:= \neg\texttt{unlock}(t_i, l_x)\ \mathbf{U}\ \exists l_z : \texttt{lock}(t_i, l_z) \dot{\rightarrow} [l_z \neq l_x \\
&\qquad\qquad \wedge \forall l_y : \texttt{lock}(t_i, l_y) \dot{\rightarrow} (l_y \neq l_x \wedge \varphi'(t_i, l_x, l_y))] \\
\varphi'(t_i, l_x, l_y) &:= \mathbf{G}\ \forall t_j : \texttt{lock}(t_j, l_y) \dot{\rightarrow} [\neg(t_i \neq t_j) \vee \varphi''(t_i, t_j, l_x, l_y)] \\
\varphi''(t_i, t_j, l_x, l_y) &:= \texttt{unlock}(t_j, l_y)\ \mathbf{R}\ \neg\texttt{lock}(t_j, l_x)
\end{aligned}$$

We remind the reader that thus Ψ, $\varphi^{\mathbf{R}}$, φ', and φ'' are accepting states of the automaton at the end of the trace, while $\varphi^{\mathbf{U}}$ is not. This means that any clause containing $\varphi^{\mathbf{U}}$ will not accept at the end of the trace either. Figure 4.3 gives the schematic structure of the corresponding automaton.

The sequence of events that we want to use is:

Thread 1:	$\texttt{lock}(t_1, l_A)$	$\texttt{lock}(t_1, l_B)$	$\texttt{unlock}(t_1, l_B)$		
Thread 2:				$\texttt{lock}(t_2, l_B)$	$\texttt{lock}(t_2, l_A)$

Evaluation starts from the initial configuration:

$$s_0 = \{\{(\Psi, \beta_\varnothing)\}\}$$

On taking the first lock, t_i and l_x are bound as evaluation descends both branches from the root. We obtain partially instantiated formulae, waiting for the second

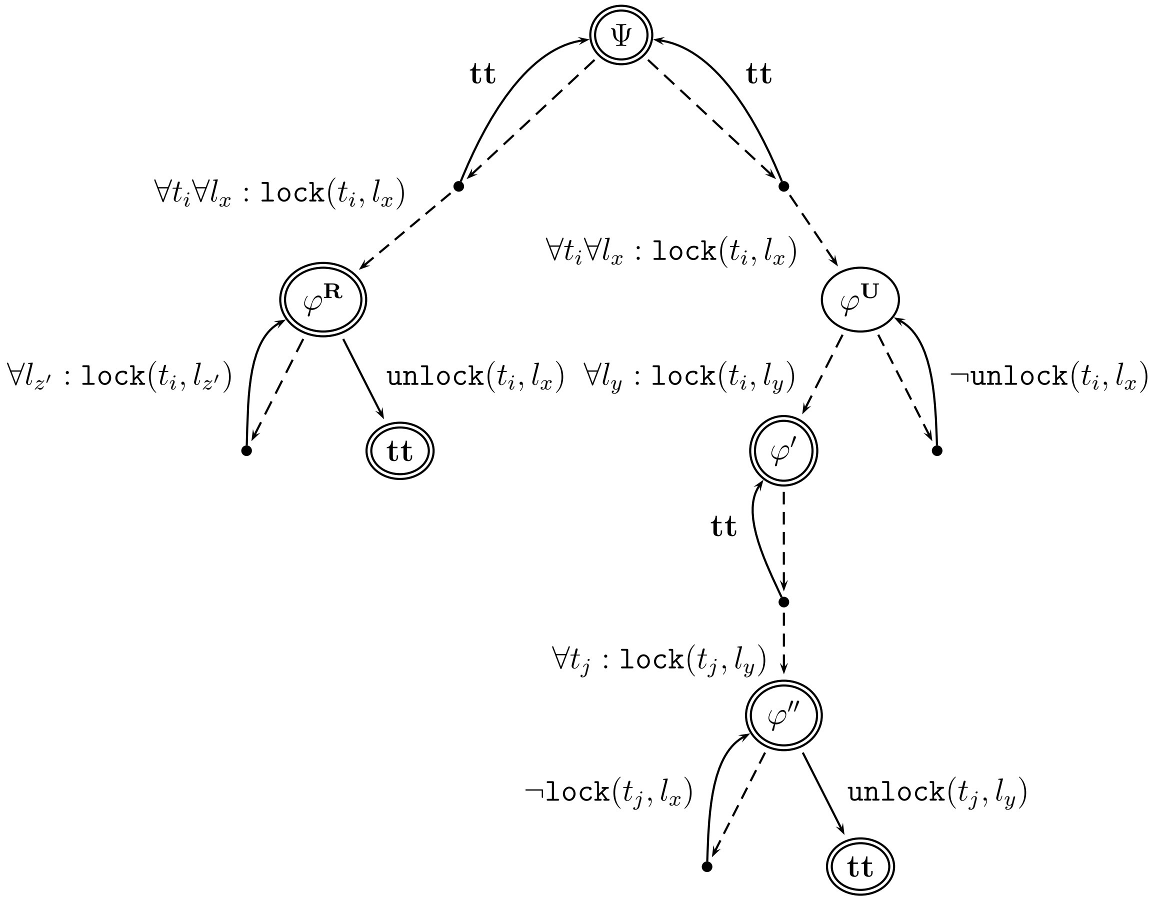


Figure 4.3: Automaton for Lock-order Reversal

lock event by the same thread while the first lock is held. The right-hand side of the Until defers since the lock event does not fulfil the constraint. Remember that we also remain in the initial configuration due to the outer Globally.

The configuration of the parametrised automaton is thus:

$$\begin{aligned} s_1 \;:=\; \{\; & \{(\Psi, \beta_\varnothing), (\varphi^{\mathbf{R}}, \{t_i/1, l_x/A\})\}, \\ & \{(\Psi, \beta_\varnothing), (\varphi^{\mathbf{U}}, \{t_i/1, l_x/A\})\} \;\} \end{aligned}$$

The second event `lock`(t_1,l_B) disproves the $\varphi^{\mathbf{R}}$ in the first clause since it is no unlock event. In the second clause, the left-hand side of the Until fails, thus requiring us to check the right-hand side. There is in fact a locking statement, satisfying the constraints so evaluation descends into φ'. The Ψ in the second clause is expanded similarly to the previous step. The following table shows the set of successors of each component:

$$\begin{aligned} (\Psi, \beta_\varnothing) &\rightsquigarrow \{\; \{(\Psi, \beta_\varnothing), (\varphi^{\mathbf{R}}, \{t_i/1, l_x/B\})\}, \\ &\qquad\; \{(\Psi, \beta_\varnothing), (\varphi^{\mathbf{U}}, \{t_i/1, l_x/B\})\} \;\} \\ (\varphi^{\mathbf{R}}, \{t_i/1, l_x/A\}) &\rightsquigarrow \mathbf{ff} = \varnothing \\ (\varphi^{\mathbf{U}}, \{t_i/1, l_x/A\}) &\rightsquigarrow \{\{(\varphi', \{t_i/1, l_x/A, l_y/B\})\}\} \end{aligned}$$

Combining the two resulting clause sets using the $\otimes$ operator, we obtain two new clauses due to the two disjuncts produced from the initial state:

$$\begin{aligned} s_2 \;:=\; \{\; & \{(\Psi, \beta_\varnothing), (\varphi^{\mathbf{R}}, \{t_i/1, l_x/B\}), (\varphi', \{t_i/1, l_x/A, l_y/B\})\}, \\ & \{(\Psi, \beta_\varnothing), (\varphi^{\mathbf{U}}, \{t_i/1, l_x/B\}), (\varphi', \{t_i/1, l_x/A, l_y/B\})\} \;\} \end{aligned}$$

The third event `unlock`(t_1, l_B) will cause the following successors to be calculated:

$$\begin{aligned} (\Psi, \beta_\varnothing) &\rightsquigarrow \{\{(\Psi, \beta_\varnothing)\}\} \\ (\varphi^{\mathbf{R}}, \{t_i/1, l_x/B\}) &\rightsquigarrow \mathbf{tt} = \{\varnothing\} \\ (\varphi^{\mathbf{U}}, \{t_i/1, l_x/B\}) &\rightsquigarrow \mathbf{ff} \\ (\varphi', \{t_i/1, l_x/A, l_y/B\}) &\rightsquigarrow \{\{(\varphi', \{t_i/1, l_x/A, l_y/B\})\}\} \end{aligned}$$

Composing the results, we get:

$$s_3 \;:=\; \{\; \{(\Psi, \beta_\varnothing), (\varphi', \{t_i/1, l_x/A, l_y/B\})\} \;\}$$

The first `lock` event in the second thread will cause a new instantiation from Ψ with $t_i/2$ and l_x/B. For $\varphi^{\mathbf{R}}$ and φ' we obtain:

$$\begin{aligned} (\Psi, \beta_\varnothing) &\rightsquigarrow \{\; \{(\Psi, \beta_\varnothing), (\varphi^{\mathbf{R}}, \{t_i/2, l_x/B\})\}, \\ &\qquad\; \{(\Psi, \beta_\varnothing), (\varphi^{\mathbf{U}}, \{t_i/2, l_x/B\})\} \;\} \\ (\varphi', \{t_i/1, l_x/A, l_y/B\}) &\rightsquigarrow \{\; \{(\varphi', \{t_i/1, l_x/A, l_y/B\}), \\ &\qquad\; (\varphi'', \{t_i/1, t_j/2, l_x/A, l_y/B\})\} \;\} \end{aligned}$$

$$
\begin{aligned}
s_4 \;:=\; \{\; &\{(\Psi, \beta_\varnothing), (\varphi^{\mathbf{R}}, \{t_i/2, l_x/B\}), (\varphi', \{t_i/1, l_x/A, l_y/B\}), (\varphi'', \{t_i/1, t_j/2, l_x/A, l_y/B\})\}\; \} \\
&\{(\Psi, \beta_\varnothing), (\varphi^{\mathbf{U}}, \{t_i/2, l_x/B\}), (\varphi', \{t_i/1, l_x/A, l_y/B\}), (\varphi'', \{t_i/1, t_j/2, l_x/A, l_y/B\})\}\; \}
\end{aligned}
$$

Triggering $\texttt{lock}(t_2, l_A)$:

$$
\begin{array}{lcl}
(\Psi, \beta_\varnothing) & \rightsquigarrow & \{\; \{(\Psi, \beta_\varnothing), (\varphi^{\mathbf{R}}, \{t_i/2, l_x/A\})\}, \\
& & \;\;\{(\Psi, \beta_\varnothing), (\varphi^{\mathbf{U}}, \{t_i/2, l_x/A\})\}\; \} \\
(\varphi^{\mathbf{R}}, \{t_i/2, l_x/B\}) & \rightsquigarrow & \mathbf{ff} \\
(\varphi^{\mathbf{U}}, \{t_i/2, l_x/B\}) & \rightsquigarrow & (\varphi^{\mathbf{U}}, \{t_i/2, l_x/B\}) \\
(\varphi', \{t_i/1, l_x/A, l_y/B\}) & \rightsquigarrow & \{\; \{(\varphi', \{t_i/1, l_x/A, l_y/B\})\}\; \} \\
(\varphi'', \{t_i/1, t_j/2, l_x/A, l_y/B\}) & \rightsquigarrow & \mathbf{ff}
\end{array}
$$

The resulting configuration is

$$s_5 := \varnothing$$

which corresponds to an overall refutation since the empty set of disjunctions resembles $\mathbf{ff}$. The mechanism has thus correctly detected the Lock-order Reversal.

5 Applications and Implementations

In this section we discuss how event data can be obtained from real-world programs which are available either as source code or as a compiled binary.

For the primitive object-based language proposed in the motivation of this thesis, events had already been integrated into the execution model, so no additional instrumentations need to take place.

In the following, we discuss how instrumentation of target applications written in modern programming languages can be achieved. The three programming languages HASKELL, C, and JAVA will be treated in detail. Additionally, we point out general techniques and requirements to efficiently support obtaining traces. Some implications of online versus offline trace checking are discussed.

5.1 General Remarks

For best adoption of our tool, as few hurdles as possible should be in the way of a prospective user. In Runtime Verification, usage boils down to two stages: firstly, to obtain the necessary events, instrumentation of the *application under test* (AUT), possibly already with respect to a specific property to verify. This step is almost always required. Only under very specific circumstances no source modification will be required (for example, shared libraries in the C setting may be sufficient to cover all locations to instrument, see Section 5.3 below). Manipulation of the original application can then be done through the *dynamic linker* of the operating system. Secondly, the application is run with some input to obtain a trace and the results is interpreted.

Usually, this will be done by a developer who has access to the full source code of the application and tools required to build it from scratch (the so-called *tool chain*, consisting often of at least a pre-processor, compiler, and linker). For interpreted languages, instrumentation of the interpreter might be an option.

Under some circumstances, not all components of the application may be equally accessible: for example, some required libraries may be available only in binary executable format, or even the whole AUT. In that case, only a limited subset may effectively be instrumented.

For the actual instrumentation, the points of interest for generation of events in the source must be identified, that is, in the coarsest approximation we suggest the same events (method calls, field access) as outlined in our motivation.

With the appropriate tool support, instrumentation can be easily automated: in the field of software engineering many tools for refactoring have been developed, which basically is what a developer is doing if he analyses the structure of his program and manipulates its behaviour in a structured way. We will outline how to use the powerful technique of Aspect-oriented Programming in the JAVA setting (see Section 5.4) to define the set of interesting events and then automatically insert both the actual event collection and the engine for verifying our parametrised formula into an existing JAVA application.

Online versus offline checking

Instrumenting method calls will thus require additional code at the beginning and the end of each method (or at least those methods of interest): data for each event must be collected and passed to the engine doing the verification. As in our simple language, all necessary values will usually be present in locally visible object attributes or variables. The engine may either verify the specification synchronously (*online*), that is, the application generating the event will be suspended until the verifier has decided that this event will not trigger a specification violation, or *offline*, where the checker runs asynchronously to the application.

In *online checking*, if a violation occurs, the user should be informed, including as many details of the trace that led up to the violation as possible, although practical considerations like memory consumption will put limits on, for example, the size of the available trace. Depending on the concrete setting, this information may also pinpoint the exact location of the instruction in the source code and other data usually available for debugging (for example, the current call stack).

Often, the verifier introduces unacceptable delays into the application through the computational overhead of checking the properties: during online verification, it may either become unresponsive or behave differently due to timing issues. This holds especially true for real-time and interactive applications. In this case, *offline checking* can be used, where instead of running the verification engine, event or trace data should only be collected and stored. Then, verification can occur in a separate step. Of course, the invalidation of a specification formula can then no longer be mapped to a specific program state, as only the abstract event data is available. Another advantage of offline checking is that trace data from multiple sources may be integrated to achieve broader coverage of possible behaviour [29].

An offline checker can be easily used as an online checker without having to think about integrating the software using standard interprocess communication facilities of modern operating system like sockets and TCP/IP network communication. For example, a checker implemented in HASKELL can analyse traces from a C program. If a simple protocol is implemented between the data aggregation side in the application and the checker, the checker can suspend the application while evaluating the next step. Since the checker should not care much where its input is coming from (either a file or a network connection), almost no additional logic except being able to handle the handshake protocol is required. The instrumentation in the application then handles only event aggregation at its side of the handshake protocol. In fact this was implemented in the HASKELL prototype (see below). Note that the evaluation of predicates over bound values is usually only effectively possible in the context of the instrumented application, though.

Path Coverage

It is commonly agreed on that testing can never provide as reliable results as formal verification. Dijkstra's quote about testing versus verification is even 30 years later still applicable. However, for the various reasons detailed in Chapter 2, practical static verification of arbitrary programs is still only a future goal. Nonetheless, we have observed that assertions are an integral part of software products (at least of their development versions). Often, they are controlled by compile-time flags to disable them once the final version is prepared for release.

Clearly there is a market for dynamic analysis of program properties, as the *Design by Contract* community shows [96]. We hope to have convinced the reader that a parametrised, temporal framework enables developers to specify high-level properties, which cannot be easily asserted without hard-coding some property-specific finite state machine-based mechanism into the product.

Unfortunately, the advantage of verification of these advanced properties has to be bought with a sometimes substantial runtime and memory overhead (see [19] for some of our measurements, and [8] for a comparison of different implementations related to matching traces in a program). Therefore, based on the observations above, we recommend for the application of the temporal assertion framework to use temporal assertions at development time and use adequate (automated) test suites to cover as many possible execution paths as possible. An automated way of generating test cases and verifying corresponding properties has been studied for example in [6].

We proceed to give some details on our experience with applying our technique.

5.2 Runtime Verification of Concurrent Haskell Programs

Our work on Runtime Verification [116, 117, 118] eventually started out with an implementation for testing applications in the same language like the checker: the functional programming language Haskell [102] was and still is our language of choice for implementing algorithms since it is well suited to manipulate lists or sets. Functional programming lends itself naturally to any kind of stream processing.

Haskell is a lazy language, that is, an application is generally a computation of a term, where evaluation is data-driven: it might be deferred until the value is required for subsequent calculations. This leads to an evaluation order which cannot be derived easily from program source code (for example, it is not feasible to single-step through program execution as you would in C or Java). So from that perspective, one would not expect it to be subject to Runtime Verification with a focus on control flow and method (function) invocation. In fact, we have limited our analysis to the imperative part of the language, where interleaved I/O operations and processes are involved [101].

Concurrent Haskell [103] integrates lightweight threads into Haskell's IO framework. As threads can communicate via shared data structures, different schedules can lead to different results. Communication takes place by means of `MVars` (*mutable variables*). These `MVars` also take the role of semaphores protecting critical sections which makes them especially interesting to analyse.

In the IO environment, threads can create `MVars` (`newEmptyMVar`), read values from `MVars` (`takeMVar`), and write values to `MVars` (`putMVar`). If a thread tries to read from an empty `MVar` or write to a full `MVar`, then it suspends until the `MVar` is filled respectively emptied by another thread. `MVars` can be used as simple semaphores, for example to assure mutual exclusion, or for simple inter-thread communication. Based on `MVars`, higher-level communication objects are built but are out of the scope of this thesis.

For triggering state transitions, two possible ways of instrumentation are provided: the checker API is able to set and release propositions over arbitrary (thus user-definable) Haskell data types. A separate instruction triggers evaluation, allowing to accumulate multiple modifications to a state before calculating the successors. Evaluation proceeds in a separate thread and can be configured to be either synchronous or asynchronous (see above). We also experimented with fluent-based propositions [61], where a proposition holds until it is explicitly released.

For convenience, the Concurrent Haskell primitives for concurrent programming have been overloaded to emit generic events through a wrapper library. This allows simple recompilation of the application under test without having to place explicit annotations into the source if one is only interested in the concurrent behaviour of the application.

The library also provides a formula for checking the Lock-order Reversal pattern, which can thus be activated through a single instructions without having to understand neither LTL nor the verifier.

The implementation of the "staged" binding mechanism for partially instantiated properties that we reported in [117] as a precursor to our current framework is illustrated through the source code fragment in Listing 5.1.

The template function takes a list of propositions as argument. The template engine will generate the input from the current state and passes it to the function. The example uses the `ThreadProp` proposition type which is triggered on `MVar` access and contains information about the current thread identifier and the affected `MVar`.

We use pattern matching to extract the arguments from the list of parameters and discard the rest through the "`_`"-placeholder. If the arguments do not fulfil the dynamic constraints, the second definition of `lock` tells the template engine that the propositions were unsuitable for instantiation by returning `Nothing`[1]: the guard tests if we are really instantiating the pattern with one thread and two different

[1]The predefined Haskell data type `Maybe` is defined as follows:

```
data Maybe a = Nothing | Just a
```

```haskell
newtype TemplateF a = TF  -- new data type for templates
  ([Prop a] -> Maybe (Either (TemplateF a) (LTL a)))

lock :: Ord a => [Prop a] ->
          Maybe (Either (TemplateF a) (LTL a))
lock (p1@(ThreadProp i1 x1):p2@(ThreadProp i2 y1):_)
  | i1 == i2 && x1 /= y1 = (Just . Left . TF)
      (\ xs -> lock2 (p1:p2:xs)) -- return anonymous function
  | otherwise = Nothing            --   with partial application
lock _ = Nothing

lock2 :: Ord a => [Prop a] ->
           Maybe (Either (TemplateF a) (LTL a))
lock2 (p1@(ThreadProp i1 x1):p2@(ThreadProp i2 y1):
       p3@(ThreadProp j1 y2):p4@(ThreadProp j2 x2):_)
  | x1 == x2 && j1 == j2 && y1 == y2 && i1 /= j1 =
      (Just . Right) $ (phi p1 p2) --> g (Not (phi p3 p4))
  | otherwise = Nothing
lock2 _ = Nothing

-- helper function:
phi u v = (u :/\: (Not v)) :/\: (u `U` v)
```

Listing 5.1: Staged template mechanism for partial instantiation of the Lock-order Reversal pattern in Haskell locks. It is not possible to use the same variable multiple times in pattern matching expression like it is in the framework presented in this thesis, so we have to use fresh variables for each thread and each lock and explicitly test in the guard those variables which have to coincide.

After the tests succeed, we use the saved expressions in `p1,p2` to enter the second part of the formula, waiting for the two remaining propositions. Notice that the complete template can be instantiated for the first time only after at least four (different) propositions fulfilling the necessary requirements have been set by the program. At this point, we can add the instantiated formula to the formulas already present and begin checking it, starting from the current state, or some suffix of the trace.

The verifier is able to read data (in a specific format) from an external resource, and, if connected via a socket, implements the proposed protocol to synchronously evaluate formulae. This allows us to reuse the verifier engine written in Haskell to verify traces from C programs online.

Like its Java counterpart J-LO, it supports only implicit quantification and does not implement overlapping propositions.

5.3 C programs and compiled programs

Instrumenting a C application proceeds along the same lines as the manual instrumentation in Haskell: first, the set of "interesting" instructions in the source code must be identified. Then, instructions which resemble events and pass the required data on to the verifier's runtime system (which will usually be linked into the resulting application) must be inserted. Tool support to provide a more declarative means of transforming existing source code is scarce and is usually limited to preprocessing. Thus, if a reasonable layer of abstraction is not already provided through a modular design of the application at hand, collecting events will mean adding many interceptions and call-backs into the verifier in many different locations, spread over different files.

For C++ applications, AspectC++ [113] could probably be used for instrumentation along the same lines which we will present in the section on Aspect-oriented Programming used in Java, although we have not used this tool in practice. AspectC++ has already been considered in the context of monitoring in [92].

For binary instrumentation, it is possible to take advantage of the runtime linking of compiled applications: since many programs will usually use the same set of libraries, it does not make sense to statically replicate this same set of functionality in every binary. Instead, shared libraries, also called *dynamic link libraries* (DLLs) on Microsoft Windows, exist in a shared directory on the machine and programs only contain references to those libraries together with, for example, versioning information in case different generations of APIs are available.

On loading such a dynamically linked application, the runtime linker will use the reference to open the appropriate shared library at most once: if another running program is already using the same library, the static part of the library is already present in memory and can be shared by both programs. Only fresh local storage for, for example, static variables used by library functions must be allocated on a per-application basis.

Function calls in the program use a symbol table to resolve the addresses of the functions in the shared library. They are resolved at load time. Since the early days of shared libraries, ways of influencing the resolution mechanism have been available. For example, before using the default references to shared libraries in the program, the user can pre-load specific libraries by listing them in the environment variable `LD_PRELOAD`, so that function names will be resolved from those first by the runtime linker before falling back to the system-wide libraries. As the namespace of functions is generally flat, this provides a convenient way of intercepting function calls on an individual basis without having to recompile it. Once inside an intercepted function call, an event can be generated, and special functions of the runtime linker like `dlopen()` can be used to pass control to the library the application was originally linked with. This technique has been used in the past, for example, for dynamically putting a security layer in top of network communication without having to change the original application.

The limitations of this approach are clear: the library used for replacing the original functions must provide exactly the same API. Also, *only* calls to shared libraries can be intercepted. Plain function calls in the original application will already have been translated to jumps to raw addresses within the resulting binary and can thus not be intercepted or instrumented without modifying the binary. Another limitation is that the interception cannot be scoped easily: *all* references to the affected functions are overloaded, making targeted instrumentation of only parts of the application impossible. Our approach in the thesis at least allowed information about the calling context in the form of the *this* argument to the `method enter/exit` event. This especially holds true when instrumenting standard functions from the system libraries which are also used in dependant libraries, although one is only interested in targeting the base application. Too much trace data will be generated, even to an amount of making it impossible to use the application at all.

Although also other programming languages compile into code which uses shared libraries, usually the functions use complex data structures as arguments, so this approach is not easily applicable to other settings.

The program supervision tool VALGRIND (see Section 6.1) allows interception of almost *each machine code instruction* of a binary program. The upcoming instruction is decoded into an internal representation and the approriate hooks are triggered before executing the original instruction. Thus it allows for a much finer granularity. Triggered actions are generally more lightweight and concerned with, for example, memory access or cache performance.

As a case-study for the Lock-order Reversal Problem, we developed a shared library which uses the `LD_PRELOAD`-mechanism described above to overload the threading primitives from the POSIX specification which are pulled in from a shared system library. After a short description of the API, we summarise our experience with this

Function name	Primitive
pthread_create	Thread creation
pthread_mutex_lock/unlock	Semaphores for mutual exclusion
pthread_cond_wait/signal	Condition variables
Augmented API	
pthread_name	Associate thread to name mapping

Figure 5.1: POSIX Threading API

approach and discuss manual instrumentation of the Apache web server.

Shared library instrumentation for threaded application

The POSIX Threading Specification [83] defines the `pthread_*` API, which amongst other things, provides thread creation and mutex-based synchronisation (see Figure 5.1). In addition to mutexes, condition variables are used avoid polling when checking for a condition.

As outlined above, we intercept calls to the API and pass the events through the means of interprocess communication to the Haskell program implementing the checker. Whether validation should be synchronous or asynchronous is configurable. If source code instrumentation is possible, our extended API also includes a function to assign human-readable names to threads which are used in the output.

One severe problem we observed is the problem of applying instrumentation only to parts of the base application (or rather, *only* the base application, and not dependant libraries): the symbol namespace of C applications is flat, so that on preloading, *all* library calls will be redirected to the new library. For the FreeBSD 4.11 operating systems, this means that also calls to the `pthread`-functions from the standard C runtime `libc`, which it needs for internal reasons, will be instrumented. In our case study, we found that a lot of time was required to separate threads of the application from the internal managment threads. These managment threads were also created in different orders depending on the functions invoked from `libc`.

Not all functions of the C runtime library such as `fprintf()` (to send debugging data to the connected debugger) could be used until the runtime had set itself up properly, since those functions would also require calls to the threading API—which then would again try to send data to the debugger! A deadlock would occur, and much detail had to be put into tracking those (undocumented) dependencies and defer using any such function until it was safe to do so. Furthermore, we expect those dependencies to be different from runtime system to runtime system, that is, even on more recent version of the FreeBSD operating system they may be different. On Linux or Solaris, there might be yet other pecularities to be observed.

For manual instrumentation of C applications, we provide a naming API to assign names to threads and mutexes. They can be used to interpret the traces, as thread identifiers are only 32 bit integer values, which additionally change from run to run of an application. The names have been used for example in the visualisation through the Concurrent Haskell Debugger [20, 30]. When investigating the Apache webserver (see our motivation for parametrised propositions in Section 3.2), sensible names had to be generated from the apparent function of the object. Since the number of threads and locks in the server is configurable, fresh names are instantiated from templates (for example, `child 1`, `child 2`, `lock 1`).

Another peculiarity of some C compared to JAVA is that some data structured do not necessarily need to be explicitly initialised. The mutex data structure of the `pthread`-API for example does not require initialisation (like threads do through `pthread_create`), but may be initialised from the runtime library on first use. In such cases, we had to identify the (sometimes presumed) location where this data structure would be used for the first time and insert the calls to our augmented naming API.

Alas, the web server has a rather conservative use of locks, so we have not been able to detect any problematic behaviour with respect to the Lock-order Reversal pattern.

Once an application is running in synchronous mode with the HASKELL checker, we are able to obtain a stack dump through use of the `gcore` library when a temporal assertion was violated. Together with the recorded trace, this dump of the call stack can then be investigated by the user to find out more about the origin of the violation, for example, by inspecting global variables or variables in the local or preceeding stack frames in the GNU Debugger `gdb`. This is also the primary way of mapping the event triggering the assertion to a specific line of code in the program, as events itself do not carry any source code information. Often, many locations may generate the same event, so it is not obvious from the event itself from which part of the application it is coming from.

Our tool discovered a possible Lock-order Reversal in the graphical `Qt` plugin for the instant messaging client `licq`, but it proved infeasible without major investments to track down the exact location in the library since no debugging symbols were compiled into it. Unfortunately, for a sufficient large project like the `Qt` library, it is very time-consuming to set up an appropriate debugging environment. Almost an arbitrary amount of time can be invested into understanding a third-party application. We expect it to be much easier to use our framework integrated into the development process, combined with understanding of the internals of the application.

Another use case derived from the POSIX specification relates to the use of so-called "async-safe calls": in a multi-threaded application, not all functions from the threading API may be invoked after a call to the heavy-weight `fork()` function from the runtime library which starts a new process. In fact, the new process is limited to single-threaded behaviour and may no longer invoke any functions of the threading API. To enforce this at runtime and providing a helpful error message instead of making the runtime system fail ungracefully because of inconsistent C runtime data structures, the following pattern can be employed:

$$\mathbf{G}\ (\texttt{fork} \rightarrow \mathbf{G}\ \neg\texttt{async\_unsafe})$$

where `async_unsafe` is the set of forbidden events. Observe that this formula does not need any quantification since `fork()` does not take any parameters and the property needs to be checked for the whole application, and not only a specific object.

5.4 Instrumenting Java programs

In [115] we investigated the possibility of automatic instrumentation of an existing Java program through the use of Aspect-oriented Programming (AOP) [46].

In the programming languages community, it was understood very early that programming methodologies that go beyond object-oriented programming can be useful and gave also rise to meta-programming or reflective programming, to name just two. The term "AOP" was coined by Gregor Kiczales in 1996 while at the Xerox Corporation research facility, where also AspectJ, one of the first practical implementations of AOP, was developed in the late 1990s.

AOP allows to *separate crosscutting concerns*, where a *concern* is usually a logical component of the application, which is implemented throughout the application (often truly crosscutting through several source files and modules). Probably the most cited examples are logging, tracing, authentication, caching, or transactions. For literature specific to AspectJ, the variant of AOP pertinent to Java, see for example [90] or [34].

The perceived advantage of AOP is being able to maintain the code *implementing* a concern in a single location and offering a way of *addressing* the locations in the program that make use of the offered functionality. Any changes to the implementation of a concern can then be done independently from the application, without having to thread source code changes through to all call sites.

The *aspect* offers a new unit of modularisation: it bundles the *logic* (implementation of concerns) and the *weaving rules* that specify how to integrate the concern with the application to obtain the final application. For example, if we consider an ATM interface, we might specify that every function manipulating the account total should verify that the session is authenticated, and, if that is not the case, prompt for authorisation. The weaving rules could then consist of a list of all API methods which in fact debit or credit the account.

Then, the *weaver* uses this information and automatically manipulates the program, so that manually inserting code for the logic in the addressed methods is not necessary. *Static crosscutting* is the first step, which does not yet modify the behaviour of the application. It usually just inserts callbacks and hooks into existing classes or methods. These can then be used in *dynamic crosscutting*, where behaviour is in fact changed by augmenting or even replacing the core program execution flow.

The rules that specify where modifications should take place need a specification language. This language must be able to address states in the dynamic execution of a program. In AspectJ, this is achieved through *pointcuts*, which select *join points* that identify single events. This includes method calls, or attribute or variable access. For a detailed discussion of the ASPECTJ join point model, we refer the interested reader to Chapter 2.4 of [90].

The actual behaviour that is to be executed when a pointcut matches is specified in the *advice*, which is usually a program fragment in the same language as the application. It is very much like a method body. Advice also carries a specification of whether it should be executed *before*, *instead*, or *after* the matched join point.

As the join points expose a subset of the current state (for example, for *before* advice they expose for method calls the arguments, and for *after* advice also the result), these bound values can be used in the advice. This also allows modification of return values through *around* advice, which is executed instead of a method.

Dynamic matching can also make use of an `if()` pointcut. This pointcut captures join points based on some conditional check in the form of a Boolean expression on the join point. It can refer to any variable that is exposed by a pointcut or is otherwise visible. Naturally evaluation of this pointcut has a highly dynamic notion. An `if()` pointcut not guarded by an event-based join point like a method call will incur a severe performance penalty since it will be checked at *every* join point, that is, on every single step of the execution again and again.

Furthermore, ASPECTJ offers pointcuts which go beyond events: lexical scoping is available via `within()/withincode()`, and there are even the context-aware pointcuts `cflow()/cflowbelow()` which take other pointcuts as arguments and select all join points within the execution of the inner pointcut. This can be used, for example, to capture all join points in the *dynamic* context of a method call, including proper handling of recursion through counters. We will not elaborate on those advanced constructs since they are beyond the scope of this thesis. However, we note that `cflow` exposes a somewhat temporal behaviour and we conclude that this is not an entirely perpendicular feature to temporal logic.

The similarity between join points and the events generated by our underlying programming language are of course not only incidental. They both stem from the same need to be able to reason (or at least talk) about points in the execution of an application. Every such point corresponds to a state change in the application, which in turn closely corresponds to the abstracted trace we use for verification.

The motivation for designing our events was to obtain a trace which resembles the execution path of the application as close as possible. The same need drives the join point model, where finer granularity means being more expressive, in the sense that there are more locations where advice can be attached to and executed.

Our motivation for Runtime Verification centred around reasoning about dynamic control flow, especially method calls. So for JAVA applications, we can simply use pointcuts to intercept the execution (at points very similar to those in our modelling language) and trigger evaluation of a set of formulae with respect to the current event/state.

The Java-Logical Observer (J-LO)

In the *Java-Logical Observer* J-LO (presented in detail in [18, 115]), we followed this approach and, instead of using some intermediate events, directly used AspectJ pointcuts as propositions. It is event-based, that is, there are no overlapping propositions on the trace. Quantification is determined through the operator (for example, Finally implies existential quantification, while Globally implies universal quantification.

A major difference from J-LO to the framework proposed in this thesis is that the set of overall events is derived from the propositions occurring in the formula: in the first step, the formula, which is stored as a Java5 annotation [84], is extracted from a Java bytecode file. A formula may also contain a body that shall be executed whenever the formula is violated. If no body is present, a generic message is printed on violation.

Then, through the AspectBench Compiler (abc) [7], callbacks from the instrumented application into the verifier are generated by means of AspectJ for each pointcut in the formula. The weaver has already a limited means of pointing out (statically) unmatched pointcuts which could in principle be used to short-cut evaluation of the affected propositions since they can never be true in any run. `if()` pointcuts are handled separately and have to be evaluated within the verifier advice, as in our extension they may refer to previously bound objects which are no longer in the current scope. If object references are used, pointcuts and the advice will refer to the current object attributes only. Any previous object attribute values have to be bound to variables exposed in the pointcut explicitly if they are required in alter stages of the temporal evaluation. In [115], we also introduced the notion of joining pointcuts in temporal formulae by binding values in logical variables that may occur in multiple pointcut expressions.

At execution time, advice will be triggered every time a pointcut matches. The verifier performs its task, either executing the advice attached to the formula with the current bindings in case of a refutation, or changing state and proceeding with the execution of the original application. Change state means that new values may be bound to free variables and evaluation proceeds with the unwound partially instantiated remaining formula.

At program termination, the remaining formulae are resolved according to their finite paths semantics, that is, eventualities (**U** formulae) fail since they have not been fulfilled, and safety properties are evaluated to `tt` (**R** formulae).

The careful reader will have noticed that by choosing appropriate AspectJ points and their exposed parameters, a mechanism similar to the filtering described in Section 4.1 is available.

By relying on the propositions actually occurring in the formula, it is possible to influence the model that the formula is checked against. This means that propositions occurring in tautological sub-formulae will select additional join points from the execution path that may affect the evaluation result of other subformulae.

With respect to *garbage collection*, the J-LO implementation uses only *weak ref-*

erences to objects. This means that the JAVA garbage collector can reclaim objects from the heap if the verifier is the only entity holding a reference to it. Objects which have disappeared but are still bound in the context of a formula can be detected in the next transition of the verifier. All propositions referring to such an object are refuted: they will never occur on the trace again.

We have successfully tested our implementation with various assertions over data structures as well as on an instance of the Lock-order Reversal pattern from Section 4.2.

The work of Allan *et al.* [2] discusses an instance of the *safe iterator* pattern in the JAVA drawing package *JHotDraw*[2], which is often used as a proving ground for aspect-oriented techniques. We were able to capture the requirement in a formula (see Listing 5.2, slightly edited due to the awkwardness of the annotation syntax) and reproduce their results by executing a sequence of events violating the pattern in the graphical user interface. The error was properly picked up. If no instrumentation had been present, the error would probably have gone unnoticed.

5.5 Using Metadata to Record Semantic Wisdom

In [19], we investigated a more user-friendly approach of storing interface-specific properties in a standardised way together with the interface in a so-called *Tracecheck* as an extension to the meta data-based approach already presented in [115].

By specifying otherwise usually only loosely documented properties and usage-constraints of interfaces in a machine-readable way, they can be consumed both by humans as additional documentation and automated tools such as J-LO to verify them at runtime. Especially for object-oriented systems with inheritance, runtime verification can be inherited top-down, for example, from an abstract interface to a concrete implementation.

As an additional advantage, keeping such semantic properties like the "temporal interface" of a class separate from the code actually implementing the necessary mechanism to verify them, we hope to be able to analyse different such specifications with respect to conflicts or potential for interference. This would not be as easily possible if, like in the JAVA iterator example, some home-grown mechanism is implemented.

Tracechecks use a syntax similar to Tracematches (see "Related Work" in Section 6.1), but allow a more flexible use of free variables in formulae. Listing 5.3 shows how the "safe iterator usage" is specified as a Tracecheck together with a body that is to be executed whenever the temporal assertion is violated.

[2] `http://www.jhotdraw.org/`

```
public class FailSafeEnumJLO {

@LTL("java.util.Vector c, java.util.Enumeration i:
 G((
      exit(call(java.util.Enumeration+.new(..))
       && args(c)) returning i
   ) -> (
     X(
        G(
            (
              entry((call(* java.util.Vector.add*(..))
              || call(* java.util.Vector.remove*(..))
              || call(* java.util.Vector.clear())
              || call(* java.util.Vector.retainAll(..))
              || call(* java.util.Vector.set*(..))
              || call(* java.util.Vector.insertElementAt(..)))
             && target(c))
            ) -> (
              G(!(
                 entry(call(java.lang.Object
                   java.util.Enumeration.nextElement()) && target(i))
                   )
                )
              )
          )
       )
   )
 )"
 ...
}
```

Listing 5.2: Fail-safe iterator annotation applied in the *JHotDraw* use case

```
tracecheck(Collection c, Iterator i) {

 sym iterator(Collection c, Iterator i) after returning (i):
    call(Collection+.iterator()) && target(c)
 sym modify(Collection c) after returning:
    (call(Collection+.add(..)) ||
     call(Collection+.remove(..))) && target(c)
 sym next(Iterator i) before:
    call(Iterator.next()) && target(i)

 G(iterator(c,i) -> G(modify(c) -> G(!next(i))) ) {
   throw new ConcurrentModificationException
                ("Collection "+c+" modified!");
 }
}
```

Listing 5.3: Safe iterator Tracecheck

6 Conclusion

6.1 Related Work

Monitoring Frameworks

HAWK and EAGLE

HAWK [36] is a (not publicly available) JAVA-oriented extension of the rule-based EAGLE logic [13, 14] that has been shown capable of defining and implementing a range of finite trace monitoring logics, including future and past time temporal logic, metric (real-time) temporal logics, interval logics, forms of quantified temporal logics, extended regular expressions, state machines, and others.

A monitor for a HAWK formula checks if a finite trace of program events satisfies the formula. Monitoring is achieved on a state-by-state basis avoiding any need to store the input trace. HAWK extends EAGLE with constructs for capturing parametrised program events such as *method calls* and *returns*. Parameters can be the executing thread, the objects that methods are called upon, arguments to methods, and return values. HAWK allows to refer to these events in formulae. As it is targeted to event-based systems, only implicit quantification is available. The tool synthesises monitors from formulae and automates program instrumentation. In [8], some performance measurements have been provided that indicate that at least the current version suffers from excessive memory consumption.

Definitions of HAWK logic observer specifications are written in a separate language. From this, an equivalent EAGLE specification and an ASPECTJ aspect are generated. The following sample states that if a file is ever written, it must have been opened before:

$$\texttt{Always}([\texttt{file?.write}(*)]\ \texttt{Previously}(\langle\texttt{file.open}()\rangle\ \texttt{true}))$$

The authors proposed an even tighter integration of their tool chain within a JAVA program through ASPECTJ, so that temporal formulae become part of the ASPECTJ pointcut language. We believe that the implementation of our JAVA prototype J-LO is a sufficient proof of concept that this is indeed a practicable solution.

Java PathExplorer

JavaPathExplorer [75] due to Havelund and Roşu reasons about traces. They use similar semantics of LTL over finite paths, although their approach to instrumentation of JAVA programs is not AOP based. Not to be confused with the *Java PathFinder* [125], the explicit state model checker for JAVA programs.

Tracecuts and tracematches

Walker and Viggers [126] proposed a language extension to AspectJ, *tracecuts*. Tracecuts do not match on events in the execution flow as AspectJ pointcuts do, but instead match on traces of such events. Those traces are specified by means of context-free expressions over pointcuts. Since this approach provides a language extension, it cannot be used in combination with ordinary Java compilers. Tracecuts do not provide automatic tracking of state. No implementation has been provided. Inspired by this work, Allan *et al.* [2] extended the *abc* compiler with *tracematches* which allow to bind free variables in pointcut expressions. The implementation in *abc* follows similar thoughts as the approach we proposed in [117]. Allan *et al.* however do not employ alternating automata in their model.

Java-Mac

Java-MaC [88] is a runtime-assurance tool for Java. The Meta Event Definition Language (MEDL) is used to specify safety properties. As the MaC architecture is designed to be language-independent, a Primitive Event Definition Language (PEDL) provides the binding to the target language, here Java. While Java-PEDL has been designed to closely correspond to Java, it is not as comfortable to use as AspectJ where expressions are not *modelled after* Java, but in fact are Java expression. Also, state in MEDL seems to be limited to primitive types.

Java-MOP

The MOP (Monitor Oriented Programming) framework [27] is designed to build monitors for object-oriented programs. A monitor is divided into a logic engine and a logic plugin. The engine generates the code that checks the trace. The plugin is concerned with extracting trace data from the program and submitting it to the engine. Java-MOP [28] is the corresponding framework for Java. It offers future- and past-time linear temporal logic, as well es extended regular expressions. Properties are specified as comments in the source code. It does not support quantification.

Temporal Rover

Temporal Rover [44] is a commercial product by Time Rover, Inc. It handles future and past time metric temporal logic requirements embedded in comments by manual source-to-source transformation.

Gamma

Klose and Ostermann [89] discuss how temporal relations can be expressed in Gamma, an aspect-oriented language on top of a minimal object-oriented core language. Pointcuts are specified in a Prolog-like language and include timestamps that can be compared using the predicates `isbefore/after`. Their prototype requires an already existing trace to apply aspects to and is not applicable to an existing language. Although overlapping propositions could probably be implemented through overlapping timestamps, GAMMA targets event-based systems.

Alloy/Embee

The EMBEE framework [35] checks whether the runtime state of a program at certain user-specified locations conforms to a given object model with the help of the Alloy Analyzer [81]. It uses the Java Platform Debugger Architecture (JPDA) from Sun Microsystems to insert breakpoints into the application under test. Object-model conformance is then checked using the Alloy analyser when such a breakpoint is reached. It is concerned with structural rather than temporal properties of objects.

Instrumentation techniques

Valgrind

VALGRIND [98] is a system for profiling x86 programs by instrumenting them at runtime. Tools for detecting memory management and cache performance are provided. It has been applied to such notable projects as Open/StarOffice and even Nasa Mars lander software. Extending VALGRIND should be the natural choice if applications compiled to native code (for example, code compiled from C or C++) should be instrumented. In fact, an earlier version contained a tool implementing the Eraser-algorithm which detects data-races in multi-threaded programs [108]. But due to the limitations of using shared libraries for interception as we have pointed out in Section 5.3, this feature has been removed from the current version of VALGRIND.

Bytecode engineering libraries like BCEL[1] or Soot [121] can also be used to instrument JAVA bytecode. They allow to insert arbitrary code into an existing class or otherwise modify it. The current implementation of the weaver in the AspectJ compiler *ajc* is based on BCEL. The extensible AspectJ research compiler *abc* we use for our research employs Soot for analysis and weaving.

AOP-based approaches

Temporal logics have already been used together with AOP: In [1], rules based on temporal logics are used to describe sequences of instructions where events should be inserted. The instrumentation happens on a static level and does not consider free variables.

Douence, Fradet and Südholt [42] developed an aspect calculus where advice can be triggered not only via a single joinpoint but via *sequences*. Their work is targeted towards a formal model of joinpoint matching and advice execution, and less on an actual implementation. Their formalism describes regular sequences of joinpoints, so it can rather be compared to the sequential model of tracematches than to ours.

[1] BCEL - The Byte Code Engineering Library, `http://jakarta.apache.org/bcel/`

Consequently, they cannot express overlapping events. It is implemented in the Arachne system [43], a dynamic weaver for C applications.

Vanderperren *et al.* [123] propose the stateful pointcut language JAsCo. Pointcuts trigger transitions in a deterministic finite automaton and advice can be attached to each pointcut. JAsCo does not provide a means of quantification or dynamic bindings, but still it could be a suitable platform to implement Temporal Assertions on top of.

`jassda` framework

The `jassda` framework [21] uses the Java Debug Interface (JDI). Thus, it presents a solution where no instrumentation is required due to the use of an interpreter (the Java Virtual Machine) that already offers event-generation (see Section 5.1). The events roughly correspond to the ones we discussed earlier, although return values from method calls are not easily available.

Before use, the debuggees are configured with respect to the events that are interesting to the debugger. `jassda` does this based on a symbol alphabet of events retrieved from the debug module. Then, CSP-like specifications are checked on the fly on the incoming trace.

Solaris DTrace

DTrace [23] is an event-based tracing mechanism for Sun's Solaris operating system. Trace data is generated by so-called *providers*, where many are already shipped with the default operating system installation. They include events about locking, file access, and system calls. Events and counters expose internal state.

Consumers can dynamically enable or disable kernel or user-land probes. DTrace allows for tens of thousands of instrumentation points, with already 30.000 provided just by the kernel. The high-level control language D (loosely based on C) allows to specify predicates and actions. Sun took care to implement this feature as efficiently as possible, so that no instrumentation indeed translates directly to no overhead. Also, safeguards to accidental system failure through misuse have been devised. In fact, DTrace has rather been designed as *not* being able to influence the system at all, (one potential of the Runtime Verification approach in the sense that it also might contribute behaviour to the system).

Data structures for aggregation can be specified in-kernel for maximal efficiency. Consumers can poll these periodically. Common applications usually include performance analysis, for example, lock contention or (frequency of) memory allocation.

DTrace itself does not directly offer any way of usefully interpreting obtained data, it is rather just the instrumentation framework. We expect that DTrace can be used to monitor many safety properties on the running operating system, although the wealth and frequency of events will surely pose a huge computational burden on consumers and will prohibit complex operations on the data.

jMonitor

jMonitor [86] is a pure JAVA library and runtime utility for specifying event patterns and associating them with user provided monitors that get called when the specified runtime events occur during the execution of JAVA applications. The instrumentation works at the JAVA bytecode level and does not require the presence of source code for the JAVA application that is being monitored.

It is more lightweight than applying AOP since it is applied at class-loader level, but monitors have to be programmed in JAVA, and no declarative specification language is provided. State machines for regular or temporal behaviour (that is, sequences of events expressed as regular expressions or LTL formulae over events), could be implemented on top of jMonitor. The same holds for quantification which must be programmatically handled in the monitor.

Verification

The following tools are usually not directly applicable to a concrete program or programming language and require the necessary (usually non-trivial) abstraction of the program into the input language of the tool.

Parametric Regular Path Queries

Recently in [91] parametric regular path queries have been investigated. They declaratively express queries queries on graphs as regular-expression-like patterns that are matched against paths in the graph. Their use is motivated through use in program analysis and model checking. The authors provide comprehensive benchmarks and complexity results.

SPIN

SPIN [78] is a popular model checker that has been applied to an enormous variety of closed systems, that is, self-contained models without any input. The high-level specification language PROMELA (PROcess MEta LAnguage) allows for a bounded number of processes and channels for communication. Control-structures are non-deterministic, and loosely based on Dijkstra's guarded command language notation. I/O operations resemble Hoare's CSP language. Models can be checked for omega-regular properties, which includes LTL.

Model Checking Java

In [80] Iosif and Sisto define a formal specification technique for expressing properties related to object-oriented source code, and particularly concurrent and distributed code, taking as a reference the JAVA language. They propose a division into *interface* and *implementation properties*. Propositions used in interface properties address control flow and allow reasoning about method calls (method name and arguments) and returns (with return value). The specification language is modelled around LTL and also allows quantification.

Actual translations of JAVA into PROMELA, the input language of SPIN, to check for deadlocks in concurrent programs have been presented in [37, 125].

dSPIN

The dSPIN model checker [38] is an extension to SPIN, that offers efficient means for the verification of concurrent high-level programs. It features dynamic memory allocation, memory references, recursion, and garbage collection.

Verification/Checking of Locking

In [85], Kahlon *et al.* show that for threads with nested access to locks the model checking problem is decidable through pushdown systems with respect to a fixed set of locks and no communication. (It has been shown that, for example, reachability is undecidable even for two threads if they communicate using CCS-style pairwise rendezvous [106].)

6.2 Summary

In this thesis we have presented a framework for the verification of Temporal Assertions for sequential and concurrent programs at runtime. These assertions use the Linear Time Logic LTL to express valid sequences of propositions which correspond to events in the control flow of an application in a simple object-based programming language. Events are, for example, method enter and exit.

In the dynamic control flow of an application, these events occur for specific arguments and return values. To capture the dynamic valuations, we have introduced parametrised propositions with quantified variables that bind values based on the current state only. This state-based quantification avoids having to quantify over potentially huge domains like object references (pointers), or integers.

For event-based systems, like traces obtained from programs where there can be no overlapping of propositions, the effects of quantifiers only differ in their results if the quantified proposition is not present in the current state. This leads in our experience to formulae where a Globally operator implies universal, and Finally existential quantification.

Since LTL properties under some variable assignment are statically generally not efficiently decidable for real-world programs with input (nor even reachability of specific source code locations), we have proposed an algorithm to check the properties on the fly for an actual execution trace.

Our algorithm is based on a breadth-first traversal of the run trees of the alternating finite automaton corresponding to an LTL formula, which incurs a double-exponential overhead in the size of the formula due to the conciseness of alternating finite automata compared to nondeterministic finite automata. We resolve the nondeterminism at runtime: a configuration of the breadth-first evaluation can be represented as a set of sets of automaton states together with a valuation for variables, where the number of states of the automaton is linear in the size of the closure of the LTL formula. The abstract outgoing edges and successor states can be statically computed, but have to be checked against the input under the concrete binding in the configuration in each step. An accepting configuration is one such inner set where all states are in the acceptance set of the alternating automaton.

Although, like assertions, our approach cannot prove the absence of errors, it gives the programmer a more powerful means of automatically checking assumptions about his program at runtime.

Practical examples from object-based and concurrent programs underlined the general usefulness of the approach. A proof-of-concept prototype developed in JAVA confirmed the practicality of our approach.

Bibliography

[1] R. A. Åberg, J. L. Lawall, M. Südholt, G. Muller, and A.-F. L. Meur. On the automatic evolution of an OS kernel using temporal logic and AOP. In *Proc. of Automated Software Engineering (ASE'03)*. IEEE, 2003.

[2] C. Allan, P. Avgustinov, A. Simon, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittamplan, and J. Tibble. Adding Trace Matching with Free Variables to AspectJ. In *OOPSLA '05, San Diego, California, USA*, October 2005.

[3] R. Alur. Timed Automata. In N. Halbwachs and D. Peled, editors, *11th International Conference of Computer Aided Verification (CAV'99)*, volume 1633 of *Lecture Notes in Computer Science*, pages 8–22. Springer, 1999.

[4] R. Alur, K. Etessami, and P. Madhusudan. A temporal logic of nested calls and returns. In K. Jensen and A. Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*. Springer, 2004.

[5] R. Armoni, L. Fix, A. Flaisher, R. Gerth, B. Ginsburg, T. Kanza, A. Landver, S. Mador-Haim, E. Singerman, A. Tiemeyer, M. Y. Vardi, and Y. Zbar. The ForSpec Temporal Logic: A new Temporal Property-Specification Language. In Katoen and Stevens [87].

[6] C. Artho, D. Drusinsky, A. Goldberg, K. Havelund, M. R. Lowry, C. S. Pasareanu, G. Rosu, and W. Visser. Experiments with test case generation and runtime analysis. In E. Börger, A. Gargantini, and E. Riccobene, editors, *Abstract State Machines*, volume 2589 of *Lecture Notes in Computer Science*, pages 87–107. Springer, 2003.

[7] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. abc: an extensible AspectJ compiler. In *AOSD'05: Proceedings of the Fourth international conference on Aspect-oriented software development*. ACM Press, 2005.

[8] P. Avgustinov, J. Tibble, E. Bodden, O. Lhoták, L. Hendren, O. de Moor, N. Ongkingco, and G. Sittampalam. Efficient trace monitoring. Technical Report abc-2006-1, The abc Group, 2006.

[9] C. Baier, B. R. Haverkort, H. Hermanns, J.-P. Katoen, and M. Siegle, editors. *Validation of Stochastic Systems*, volume 2925 of *Lecture Notes in Computer Science*. Springer, 2004.

[10] J. Baldwin. Locking in the Multithreaded FreeBSD Kernel. In S. J. Leffler, editor, *Proceedings of BSDCon 2002, February 11-14, 2002, San Francisco, California, USA*. USENIX, 2002.

[11] T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani. Automatic Predicate Abstraction of C Programs. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation (PLDI'01)*, pages 203–213. ACM Press, 2001.

[12] T. Ball and S. K. Rajamani. The SLAM Toolkit. In Berry et al. [17], pages 260–264.

[13] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Program Monitoring with LTL in EAGLE. In *18th International Parallel and Distributed Processing Symposium (IPDPS 2004)*. IEEE Computer Society, 2004.

[14] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In B. Steffen and G. Levi, editors, *5th International Conference on Verification, Model Checking, and Abstract Interpretation, (VMCAI 2004)*, volume 2937 of *Lecture Notes in Computer Science*, pages 44–57. Springer, 2004.

[15] I. Beer, S. Ben-David, C. Eisner, D. Fisman, A. Gringauze, and Y. Rodeh. The Temporal Logic Sugar. In Berry et al. [17].

[16] S. Bensalem and K. Havelund. Dynamic deadlock analysis of multi-threaded programs. In S. Ur, E. Bin, and Y. Wolfsthal, editors, *Haifa Verification Conference*, volume 3875 of *Lecture Notes in Computer Science*, pages 208–223. Springer, 2005.

[17] G. Berry, H. Comon, and A. Finkel, editors. *13th International Conference on Computer Aided Verification (CAV'01)*, volume 2102 of *Lecture Notes in Computer Science*. Springer, 2001.

[18] E. Bodden. J-LO, a Tool for Runtime Checking Temporal Assertions. Master's thesis, RWTH Aachen University, Germany, 2005. Available from `http://www-i2.informatik.rwth-aachen.de/JLO/`.

[19] E. Bodden and V. Stolz. Tracechecks: Defining semantic interfaces with temporal logic. In W. Löwe and M. Südholt, editors, *5th International Symposium on Software Composition (SC'06)*, volume 4089 of *Lecture Notes in Computer Science*. Springer, 2006.

[20] T. Böttcher and F. Huch. A Debugger for Concurrent Haskell. In *Draft. Proc. of the 14th International Workshop on the Implementation of Functional Languages (IFL'02)*, Madrid, Spain, Sept. 2002.

[21] M. Brörkens and M. Möller. Dynamic Event Generation for Runtime Checking using the JDI. In Havelund and Roşu [73].

[22] G. Bruns and P. Godefroid. Temporal logic query checking. In *Proc. of the 16th IEEE Symp. on Logic in Computer Science (LICS 2001)*, pages 409–417. IEEE Computer Society, 2001.

[23] B. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic instrumentation of production systems. In *USENIX Annual Technical Conference, General Track*, pages 15–28. USENIX, 2004.

[24] W. Chan. Temporal-logic queries. In *Proceedings of the 11th Conference on Computer Aided Verification (CAV 1999)*, volume 1855 of *Lecture Notes in Computer Science*. Springer, 1999.

[25] A. Chandra, D. C. Kozen, and L. J. Stockmeyer. Alternation. *Journal of the ACM*, 28:114–133, 1981.

[26] A. K. Chandra and L. J. Stockmeyer. Alternation. In *17th Annual Symposium on Foundations of Computer Science*, pages 98–108. IEEE, 1976.

[27] F. Chen, M. d'Amorim, and G. Roşu. A formal monitoring-based framework for software development and analysis. In J. Davies, W. Schulte, and M. Barnett, editors, *ICFEM*, volume 3308 of *Lecture Notes in Computer Science*, pages 357–372. Springer, 2004.

[28] F. Chen and G. Roşu. Java-MOP: A Monitoring Oriented Programming Environment for Java. In Halbwachs and Zuck [65], pages 546–550.

[29] T. M. Chilimbi. Asymptotic Runtime Verification through Lightweight Continous Program Analysis (invited talk). In *Fifth Workshop on Runtime Verification (RV'05)*. To be published in ENTCS, Elsevier, 2005.

[30] J. Christiansen and F. Huch. Searching for deadlocks while debugging Concurrent Haskell programs. In C. Okasaki and K. Fisher, editors, *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming (ICFP 2004)*, pages 28–39. ACM Press, 2004.

[31] G. Chugunov, L.-Å. Fredlund, and D. Gurov. Model Checking of Multi-Applet JavaCard Applications. In *Proceedings of the Fifth Smart Card Research and Advanced Application Conference, (CARDIS'02)*, pages 87–96. USENIX, 2002.

[32] E. Clarke and E. Emerson. Design and synthesis of synchronisation skeletons using branching time temporal logic. In *Logic of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer, 1981.

[33] E. Clarke Jr, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, Cambridge, Massachusetts, 1999.

[34] A. Colyer, A. Clement, G. Harley, and M. Webster. *Eclipse AspectJ: Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools*. Pearson Education, 2005.

[35] M. L. Crane and J. Dingel. Runtime Conformance Checking of Objects Using Alloy. In Sokolsky and Viswanathan [112].

[36] M. d'Amorim and K. Havelund. Event-Based Runtime Verification of Java Programs. In *WODA '05: Proceedings of the third international workshop on Dynamic Analysis*, pages 1–7, New York, NY, USA, 2005. ACM Press.

[37] C. Demartini, R. Iosif, and R. Sisto. A deadlock detection tool for concurrent java programs. *Softw., Pract. Exper.*, 29(7):577–603, 1999.

[38] C. Demartini, R. Iosif, and R. Sisto. dSPIN: A Dynamic Extension of SPIN. In D. Dams, R. Gerth, S. Leue, and M. Massink, editors, *SPIN*, volume 1680 of *Lecture Notes in Computer Science*, pages 261–276. Springer, 1999.

[39] E. W. Dijkstra. The humble programmer. ACM Turing Lecture, 1972.

[40] D. Distefano. *On Model Checking the Dynamics of Object-based Software: a Fundamental Approach*. Twente University Press, The Netherlands, 2003.

[41] D. Distefano, A. Rensink, and J.-P. Katoen. Model checking birth and death. In R. Baeza-Yates, U. Montanari, and N. Santoro, editors, *Foundations of Information Technology in the Era of Networking and Mobile Computing. In 2nd IFIP International Conference on Theoretical Computer Science (TCS)*, pages 435–447. Kluwer, 2002.

[42] R. Douence, P. Fradet, and M. Südholt. Composition, reuse and interaction analysis of stateful aspects. In G. C. Murphy and K. J. Lieberherr, editors, *Proc. of the 3rd Intl. Conf. on Aspect-oriented software development (AOSD'04)*. ACM, 2004.

[43] R. Douence, T. Fritz, N. Loriant, J.-M. Menaud, M. Ségura-Devillechaise, and M. Südholt. An expressive aspect language for system applications with Arachne. In *Proc. of the 4th Intl. Conf. on Aspect-oriented software development (AOSD'05)*. ACM Press, 2005.

[44] D. Drusinsky. The Temporal Rover and the ATG Rover. In Havelund et al. [70], pages 323–330.

[45] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st Intl. Conf. on Software engineering (ICSE'99)*. IEEE Computer Society Press, 1999.

[46] T. Elrad, S. Clarke, and M. Akşit. *Aspect-oriented Software Development*. Addison-Wesley, 2004.

[47] D. R. Engler. Static analysis versus model checking for bug finding. In M. Abadi and L. de Alfaro, editors, *CONCUR*, volume 3653 of *Lecture Notes in Computer Science*, page 1. Springer, 2005.

[48] J. Esparza and S. Schwoon. A BDD-based Model Checker for Recursive Programs. In Berry et al. [17], pages 324–336.

[49] K. Etessami and S. K. Rajamani, editors. *17th Intl. Conf. on Computer Aided Verification, (CAV 2005)*, volume 3576 of *Lecture Notes in Computer Science*. Springer, 2005.

[50] D. Evans, J. Guttag, J. Horning, and Y. M. Tan. Lclint: A Tool for Using Specifications to Check Code. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering*. ACM Press, December 1994.

[51] D. Evans and D. Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1):42–51, 2002.

[52] B. Finkbeiner, S. Sankaranarayanan, and H. Sipma. Collecting statistics over runtime executions. In Havelund and Roşu [73].

[53] B. Finkbeiner and H. Sipma. Checking Finite Traces using Alternating Automata. *Formal Methods in System Design*, 24(2):101–127, 2004.

[54] L.-Å. Fredlund. Guaranteeing Correctness Properties of a Java Card Applet. In Havelund and Roşu [76], pages 217–233.

[55] D. M. Gabbay. The Declarative Past and Imperative Future: Executable Temporal Logic for Interactive Systems. In B. Banieqbal, H. Barringer, and A. Pnueli, editors, *Temporal Logic in Specification*, volume 398 of *Lecture Notes in Computer Science*, pages 409–448. Springer, 1987.

[56] D. M. Gabbay, A. Pnueli, S. Shelah, and J. Stavi. On the Temporal Basis of Fairness. In *ACM Symposium on Principles of Programming Languages (POPL'80)*, pages 163–173. ACM Press, 1980.

[57] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In O. M. Nierstrasz, editor, *ECOOP'93—Object-Oriented Programming*, volume 707 of *Lecture Notes in Computer Science*. Springer, 1993.

[58] P. Gastin and D. Oddoux. Fast LTL to Büchi Automata Translation. In Berry et al. [17], pages 53–65.

[59] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In P. Dembinski and M. Sredniawa, editors, *PSTV*, volume 38 of *IFIP Conference Proceedings*, pages 3–18. Chapman & Hall, 1995.

[60] D. Giannakopoulou and K. Havelund. Automata-based Verification of Temporal Properties on Running programs. In *16th IEEE International Conference on Automated Software Engineering (ASE 2001)*. IEEE Computer Society, 2001.

[61] D. Giannakopoulou and J. Magee. Fluent model checking for event-based systems. In *ESEC / SIGSOFT FSE*, pages 257–266, 2003.

[62] P. Godefroid. Model checking for programming languages using Verisoft. In *ACM Symposium on Principles of Programming Languages (POPL'97)*, pages 174–186. ACM Press, 1997.

[63] E. Grädel, W. Thomas, and T. Wilke, editors. *Automata, Logics, and Infinite Games: A Guide to Current Research [outcome of a Dagstuhl seminar, February 2001]*, volume 2500 of *Lecture Notes in Computer Science*. Springer, 2002.

[64] S. Graf and H. Saïdi. Construction of Abstract State Graphs with PVS. In O. Grumberg, editor, *9th International Conference on Computer Aided Verification (CAV'97)*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer, 1997.

[65] N. Halbwachs and L. D. Zuck, editors. *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05)*, volume 3440 of *Lecture Notes in Computer Science*. Springer, 2005.

[66] J. Harrow. Runtime Checking of Multithreaded Applications with Visual Threads. In Havelund et al. [70].

[67] J. Hatcliff and M. B. Dwyer. Using the Bandera tool set to model-check properties of concurrent Java software. In K. G. Larsen and M. Nielsen, editors, *12th International Conference Concurrency Theory (CONCUR 2001)*, volume 2154 of *Lecture Notes in Computer Science*, pages 39–58. Springer, 2001.

[68] K. Havelund. Using Runtime Analysis to Guide Model Checking of Java Programs. In Havelund et al. [70].

[69] K. Havelund, M. R. Lowry, and J. Penix. Formal Analysis of a Space-Craft Controller Using SPIN. *IEEE Trans. Software Eng.*, 27(8):1000–9999, 2001.

[70] K. Havelund, J. Penix, and W. Visser, editors. *SPIN Model Checking and Software Verification (7th International SPIN Workshop)*, volume 1885 of *Lecture Notes in Computer Science*, Stanford, CA, USA, August/September 2000. Springer.

[71] K. Havelund and G. Roşu. Monitoring Java Programs with Java PathExplorer. In *Proceedings of the First Workshop on Runtime Verification (RV'01)* [72].

[72] K. Havelund and G. Roşu, editors. *Proceedings of the First Workshop on Runtime Verification (RV'01)*, volume 55 of *Electr. Notes in Theor. Comput. Sci.* Elsevier, 2001.

[73] K. Havelund and G. Roşu, editors. *Proceedings of the Second Workshop on Runtime Verification (RV'02)*, volume 70 of *Electr. Notes in Theor. Comput. Sci.* Elsevier, 2002.

[74] K. Havelund and G. Roşu. Synthesizing Monitors for Safety Properties. In Katoen and Stevens [87], pages 342–356.

[75] K. Havelund and G. Roşu. An Overview of the Runtime Verification Tool Java PathExplorer. *Form. Methods Syst. Des.*, 24(2):189–215, 2004.

[76] K. Havelund and G. Roşu, editors. volume 113 of *Electr. Notes in Theor. Comput. Sci.* Elsevier, 2005.

[77] K. Havelund and G. Roşu. Testing linear temporal logic formulae on finite execution traces. Technical Report TR 01-08, RIACS, May 2001. Written 20 December 2000.

[78] G. J. Holzmann. *The SPIN model checker: primer and reference manual.* Addison-Wesley, Boston, Massachusetts, USA, September 2003.

[79] M. Huth and M. Ryan. *Logic in Computer Science.* Cambridge University Press, second edition, 2004.

[80] R. Iosif and R. Sisto. Temporal logic properties of Java objects. *Journal of Systems and Software*, 68(3):243–251, 2003.

[81] D. Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, 2002.

[82] T. Jiang and B. Ravikumar. A note on the space complexity of some decision problems for finite automata. *Information Processing Letters*, 40:25–31, 1991.

[83] A. Josey, editor. *The single Unix specification.* The Open Group, Reading, UK, 2002.

[84] Java specification request for metadata annotations (JSR175). `http://jcp.org/en/jsr/detail?id=175`.

[85] V. Kahlon, F. Ivancic, and A. Gupta. Reasoning about threads communicating via locks. In Etessami and Rajamani [49], pages 505–518.

[86] M. Karaorman and J. Freeman. jMonitor: Java Runtime Event Specification and Monitoring Library. In Havelund and Roşu [76], pages 181–200.

[87] J. Katoen and P. Stevens, editors. *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2002)*, volume 2280 of *Lecture Notes in Computer Science*. Springer, 2002.

[88] M. Kim, M. Viswanathan, S. Kannan, I. Lee, and O. Sokolsky. Java-MaC: A Run-time Assurance Approach for Java Programs. *Formal Methods in System Design*, 24(2):129–155, 2004.

[89] K. Klose and K. Ostermann. Back to the future: Pointcuts as predicates over traces. In *Foundations of Aspect-Oriented Languages workshop (FOAL'05)*, 2005.

[90] R. Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications Co., 2003.

[91] Y. A. Liu, T. Rothamel, F. Yu, S. D. Stoller, and N. Hu. Parametric regular path queries. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation (PLDI'04)*, pages 219–230, New York, NY, USA, 2004. ACM Press.

[92] D. Mahrenholz, O. Spinczyk, and W. Schröder-Preikschat. Program Instrumentation for Debugging and Monitoring with AspectC++. In *Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'02)*, pages 249–256. IEEE, 2002.

[93] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer, 1995.

[94] N. Markey. Past is for free: on the complexity of verifying linear temporal properties with past. In *9th International Workshop on Expressiveness in Concurrency (EXPRESS'02)*, volume 68 of *Electr. Notes in Theor. Comput. Sci.* Elsevier, 2002.

[95] N. Markey. Temporal logic with past is exponentially more succinct. *Bulletin of the EATCS*, 79:122–128, 2003.

[96] B. Meyer. Applying "Design by Contract". *Computer*, 25(10):40–51, 1992.

[97] P. Naldurg, K. Sen, and P. Thati. A temporal logic based framework for intrusion detection. In D. de Frutos-Escrig and M. Núñez, editors, *24th IFIP WG 6.1 Intl. Conf. on Formal Techniques for Networked and Distributed Systems (FORTE 2004)*, volume 3235 of *Lecture Notes in Computer Science*, pages 359–376. Springer, 2004.

[98] N. Nethercote and J. Seward. Valgrind: A Program Supervision Framework. In Sokolsky and Viswanathan [112].

[99] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 1999.

[100] J. Olivain and J. Goubault-Larrecq. The Orchids Intrusion Detection Tool. In Etessami and Rajamani [49], pages 286–290.

[101] S. Peyton Jones. Tackling the Awkward Squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In T. Hoare, M. Broy, and R. Steinbrüggen, editors, *Engineering theories of software construction*. IOS Press, 2001.

[102] S. Peyton Jones, editor. *Haskell 98 Language and Libraries—The Revised Report*. Cambridge University Press, 2003.

[103] S. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *23rd ACM Symposium on Principles of Programming Languages (POPL'96)*, pages 295–308. ACM Press, 1996.

[104] A. Pnueli. The Temporal Logics of Programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, 1977.

[105] J.-P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In M. Dezani-Ciancaglini and U. Montanari, editors, *Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer, 1982.

[106] G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Trans. Program. Lang. Syst.*, 22(2):416–430, 2000.

[107] T. C. Ruys and E. Brinksma. Managing the verification trajectory. *STTT*, 4(2):246–259, 2003.

[108] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Transactions on Computer Systems*, 15(4), 1997.

[109] B. Schlich and S. Kowalewski. Model Checking C Source Code for embedded Systems. In *Proceedings of the IEEE/NASA Workshop on Leveraging Applications of Formal Methods, Verification, and Validation (ISoLA 2005)*, 2005.

[110] A. Silberschatz and P. Galvin. *Operating System Concepts*. Addison-Wesley, 4th edition, 1994.

[111] A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logics. *J. ACM*, 32(3):733–749, 1985.

[112] O. Sokolsky and M. Viswanathan, editors. *Proceedings of the Third Workshop on Runtime Verification (RV'03)*, volume 89 of *Electr. Notes in Theor. Comput. Sci.* Elsevier, 2003.

[113] O. Spinczyk, A. Gal, and W. Schröder-Preikschat. AspectC++: an aspect-oriented extension to the C++ programming language. In *Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific*, pages 53–60, Darlinghurst, Australia, 2002. Australian Computer Society, Inc.

[114] L. Sterling and E. Shapiro. *The Art of Prolog.* MIT Press, 1986.

[115] V. Stolz and E. Bodden. Temporal Assertions using AspectJ. In H. Barringer, B. Finkbeiner, Y. Gurevich, and H. Sipma, editors, *Fifth Workshop on Runtime Verification (RV'05)*, volume 144 of *Electr. Notes in Theor. Comput. Sci.* Elsevier, 2005.

[116] V. Stolz and F. Huch. Runtime Verification of Concurrent Haskell (work in progress). *Proceedings of the 12th International Workshop on Functional and (Constraint) Logic Programming (WFLP'03)*, Technical Report DSIC-II/13/03, Universidad Politécnica de Valencia, Spain, 2003.

[117] V. Stolz and F. Huch. Runtime Verification of Concurrent Haskell Programms. In Havelund and Roşu [76], pages 201–216.

[118] R. Theisen. Überprüfung von LTL Eigenschaften zur Laufzeit. Master's thesis, RWTH Aachen University, Germany, 2005.

[119] W. Thomas. Star-free regular sets of omega-sequences. *Information and Control*, 42(2):148–156, 1979.

[120] W. Thomas. A combinatorial approach to the theory of ω-automata. *Information and Control*, 48(3):261–283, 1981.

[121] R. Vallée-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java Optimization Framework. In S. A. MacKay and J. H. Johnson, editors, *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative Research (CASCON 1999)*, pages 125–135. IBM, 1999.

[122] J. van Benthem. Temporal logic. In *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 4, chapter 7. Oxford University Press, 1995.

[123] W. Vanderperren, D. Suvée, M. A. Cibrán, and B. De Fraine. Stateful Aspects in JAsCo. In T. Gschwind and U. Aßmann, editors, *Workshop on Software Composition 2005 (SC'05)*, volume 3628 of *Lecture Notes in Computer Science*. Springer, 2005.

[124] M. Y. Vardi. An automata-theoretic approach to linear temporal logic. In F. Moller and G. Birtwistle, editors, *Logics for Concurrency: Structure versus Automata*, volume 1043 of *Lecture Notes in Computer Science*. Springer, 1996.

[125] W. Visser, K. Havelund, G. Brat, and S. Park. Model Checking Programs. In *15th IEEE International Conference on Automated Software Engineering (ASE 2000)*. IEEE CS Press, September 2000.

[126] R. J. Walker and K. Viggers. Implementing protocols via declarative event patterns. In R. Taylor and M. Dwyer, editors, *Proc. of the 12th ACM SIGSOFT Intl. Symp. on Foundations of Software Engineering*. ACM Press, 2004.

[127] P. Wolper. Temporal logic can be more expressive. *Information and Control*, 56:72–99, 1983.

```
class Main
  method main
     var stack v o
     stack:=new Stack
     o:=new O
     stack.push(o)
     v:=stack.next // head
L1:  b:=(v != nil)
     jmf b L2
     o:=v.o
     ...
     v:=v.next
     jmp L1
L2:  return nil
  // main
// Main
```

```
class Stack
  var o     // Container
      next  // Link

  method push(v)
    var t p
    t:=new Stack
    t.o:=v
    p:=this.next
    t.next:=p
    this.next:=t
    return nil

  method pop()
    var t p o
    t:=this.next
    p:=t.next
    this.next:=p
    o:=t.o
    return o
```

Figure 6.1: Flat source code for stack example in $\mathbb{PL}_{int}$

List of Symbols

Index

Lebenslauf Volker Stolz

Doverack 74
41836 Hückelhoven

Geburtsdatum: 10.12.1974
Geburtsort: Würselen
Familienstand: ledig

Zeitraum	**Ausbildung/Beschäftigung**
10.12.1974	Geboren in Würselen, Deutschland
1985–1994	städtisches Gymnasium Baesweiler, Abiturnote: 2,2
1994–1995	Zivildienst bei der Regionalstelle des Bistums Aachen
1995–2001	Student der Informatik an der RWTH Aachen
20.3.2001	Abschluss als Diplom-Informatiker an der RWTH Aachen, Gesamtnote: sehr gut, Diplomarbeitsthema: "Robuste Verteilte Programmierung in Haskell"
21.3.2001–30.4.2001	Wissenschaftliche Hilfskraft am Lehrstuhl für Informatik II an der RWTH Aachen
1.5.2001–28.2.2006	Wissenschaftlicher Mitarbeiter am Lehrstuhl für Informatik II an der RWTH Aachen
seit 1.8.2006	Post-doc Fellow an der United Nations University, Institute for Software Technology

Aachener Informatik-Berichte

This list contains all technical reports published during the past five years. A complete list of reports dating back to 1987 is available from `http://aib.informatik.rwth-aachen.de/`. **To obtain copies consult the above URL or send your request to: Informatik-Bibliothek, RWTH Aachen, Ahornstr. 55, 52056 Aachen, Email:** `biblio@informatik.rwth-aachen.de`

2001-01 *	Jahresbericht 2000
2001-02	Benedikt Bollig, Martin Leucker: Deciding LTL over Mazurkiewicz Traces
2001-03	Thierry Cachat: The power of one-letter rational languages
2001-04	Benedikt Bollig, Martin Leucker, Michael Weber: Local Parallel Model Checking for the Alternation Free mu-Calculus
2001-05	Benedikt Bollig, Martin Leucker, Thomas Noll: Regular MSC Languages
2001-06	Achim Blumensath: Prefix-Recognisable Graphs and Monadic Second-Order Logic
2001-07	Martin Grohe, Stefan Wöhrle: An Existential Locality Theorem
2001-08	Mareike Schoop, James Taylor (eds.): Proceedings of the Sixth International Workshop on the Language-Action Perspective on Communication Modelling
2001-09	Thomas Arts, Jürgen Giesl: A collection of examples for termination of term rewriting using dependency pairs
2001-10	Achim Blumensath: Axiomatising Tree-interpretable Structures
2001-11	Klaus Indermark, Thomas Noll (eds.): Kolloquium Programmiersprachen und Grundlagen der Programmierung
2002-01 *	Jahresbericht 2001
2002-02	Jürgen Giesl, Aart Middeldorp: Transformation Techniques for Context-Sensitive Rewrite Systems
2002-03	Benedikt Bollig, Martin Leucker, Thomas Noll: Generalised Regular MSC Languages
2002-04	Jürgen Giesl, Aart Middeldorp: Innermost Termination of Context-Sensitive Rewriting
2002-05	Horst Lichter, Thomas von der Maßen, Thomas Weiler: Modelling Requirements and Architectures for Software Product Lines
2002-06	Henry N. Adorna: 3-Party Message Complexity is Better than 2-Party Ones for Proving Lower Bounds on the Size of Minimal Nondeterministic Finite Automata
2002-07	Jörg Dahmen: Invariant Image Object Recognition using Gaussian Mixture Densities
2002-08	Markus Mohnen: An Open Framework for Data-Flow Analysis in Java
2002-09	Markus Mohnen: Interfaces with Default Implementations in Java
2002-10	Martin Leucker: Logics for Mazurkiewicz traces
2002-11	Jürgen Giesl, Hans Zantema: Liveness in Rewriting

2003-01 *	Jahresbericht 2002
2003-02	Jürgen Giesl, René Thiemann: Size-Change Termination for Term Rewriting
2003-03	Jürgen Giesl, Deepak Kapur: Deciding Inductive Validity of Equations
2003-04	Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, Stephan Falke: Improving Dependency Pairs
2003-05	Christof Löding, Philipp Rohde: Solving the Sabotage Game is PSPACE-hard
2003-06	Franz Josef Och: Statistical Machine Translation: From Single-Word Models to Alignment Templates
2003-07	Horst Lichter, Thomas von der Maßen, Alexander Nyßen, Thomas Weiler: Vergleich von Ansätzen zur Feature Modellierung bei der Softwareproduktlinienentwicklung
2003-08	Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, Stephan Falke: Mechanizing Dependency Pairs
2004-01 *	Fachgruppe Informatik: Jahresbericht 2003
2004-02	Benedikt Bollig, Martin Leucker: Message-Passing Automata are expressively equivalent to EMSO logic
2004-03	Delia Kesner, Femke van Raamsdonk, Joe Wells (eds.): HOR 2004 – 2nd International Workshop on Higher-Order Rewriting
2004-04	Slim Abdennadher, Christophe Ringeissen (eds.): RULE 04 – Fifth International Workshop on Rule-Based Programming
2004-05	Herbert Kuchen (ed.): WFLP 04 – 13th International Workshop on Functional and (Constraint) Logic Programming
2004-06	Sergio Antoy, Yoshihito Toyama (eds.): WRS 04 – 4th International Workshop on Reduction Strategies in Rewriting and Programming
2004-07	Michael Codish, Aart Middeldorp (eds.): WST 04 – 7th International Workshop on Termination
2004-08	Klaus Indermark, Thomas Noll: Algebraic Correctness Proofs for Compiling Recursive Function Definitions with Strictness Information
2004-09	Joachim Kneis, Daniel Mölle, Stefan Richter, Peter Rossmanith: Parameterized Power Domination Complexity
2004-10	Zinaida Benenson, Felix C. Gärtner, Dogan Kesdogan: Secure Multi-Party Computation with Security Modules
2005-01 *	Fachgruppe Informatik: Jahresbericht 2004
2005-02	Maximillian Dornseif, Felix C. Gärtner, Thorsten Holz, Martin Mink: An Offensive Approach to Teaching Information Security: "Aachen Summer School Applied IT Security"
2005-03	Jürgen Giesl, René Thiemann, Peter Schneider-Kamp: Proving and Disproving Termination of Higher-Order Functions
2005-04	Daniel Mölle, Stefan Richter, Peter Rossmanith: A Faster Algorithm for the Steiner Tree Problem
2005-05	Fabien Pouget, Thorsten Holz: A Pointillist Approach for Comparing Honeypots

2005-06	Simon Fischer, Berthold Vöcking: Adaptive Routing with Stale Information
2005-07	Felix C. Freiling, Thorsten Holz, Georg Wicherski: Botnet Tracking: Exploring a Root-Cause Methodology to Prevent Distributed Denial-of-Service Attacks
2005-08	Joachim Kneis, Peter Rossmanith: A New Satisfiability Algorithm With Applications To Max-Cut
2005-09	Klaus Kursawe, Felix C. Freiling: Byzantine Fault Tolerance on General Hybrid Adversary Structures
2005-10	Benedikt Bollig: Automata and Logics for Message Sequence Charts
2005-11	Simon Fischer, Berthold Vöcking: A Counterexample to the Fully Mixed Nash Equilibrium Conjecture
2005-12	Neeraj Mittal, Felix Freiling, S. Venkatesan, Lucia Draque Penso: Efficient Reductions for Wait-Free Termination Detection in Faulty Distributed Systems
2005-13	Carole Delporte-Gallet, Hugues Fauconnier, Felix C. Freiling: Revisiting Failure Detection and Consensus in Omission Failure Environments
2005-14	Felix C. Freiling, Sukumar Ghosh: Code Stabilization
2005-15	Uwe Naumann: The Complexity of Derivative Computation
2005-16	Uwe Naumann: Syntax-Directed Derivative Code (Part I: Tangent-Linear Code)
2005-17	Uwe Naumann: Syntax-directed Derivative Code (Part II: Intraprocedural Adjoint Code)
2005-18	Thomas von der Maßen, Klaus Müller, John MacGregor, Eva Geisberger, Jörg Dörr, Frank Houdek, Harbhajan Singh, Holger Wußmann, Hans-Veit Bacher, Barbara Paech: Einsatz von Features im Software-Entwicklungsprozess - Abschlußbericht des GI-Arbeitskreises “Features”
2005-19	Uwe Naumann, Andre Vehreschild: Tangent-Linear Code by Augmented LL-Parsers
2005-20	Felix C. Freiling, Martin Mink: Bericht über den Workshop zur Ausbildung im Bereich IT-Sicherheit Hochschulausbildung, berufliche Weiterbildung, Zertifizierung von Ausbildungsangeboten am 11. und 12. August 2005 in Köln organisiert von RWTH Aachen in Kooperation mit BITKOM, BSI, DLR und Gesellschaft fuer Informatik (GI) e.V.
2005-21	Thomas Noll, Stefan Rieger: Optimization of Straight-Line Code Revisited
2005-22	Felix Freiling, Maurice Herlihy, Lucia Draque Penso: Optimal Randomized Fair Exchange with Secret Shared Coins
2005-23	Heiner Ackermann, Alantha Newman, Heiko Röglin, Berthold Vöcking: Decision Making Based on Approximate and Smoothed Pareto Curves
2005-24	Alexander Becher, Zinaida Benenson, Maximillian Dornseif: Tampering with Motes: Real-World Physical Attacks on Wireless Sensor Networks
2006-01 *	Fachgruppe Informatik: Jahresbericht 2005
2006-02	Michael Weber: Parallel Algorithms for Verification of Large Systems

2006-03	Michael Maier, Uwe Naumann: Intraprocedural Adjoint Code Generated by the Differentiation-Enabled NAGWare Fortran Compiler
2006-04	Ebadollah Varnik, Uwe Naumann, Andrew Lyons: Toward Low Static Memory Jacobian Accumulation
2006-05	Uwe Naumann, Jean Utke, Patrick Heimbach, Chris Hill, Derya Ozyurt, Carl Wunsch, Mike Fagan, Nathan Tallent, Michelle Strout: Adjoint Code by Source Transformation with OpenAD/F
2006-06	Joachim Kneis, Daniel Mölle, Stefan Richter, Peter Rossmanith: Divide-and-Color
2006-07	Thomas Colcombet, Christof Löding: Transforming structures by set interpretations
2006-08	Uwe Naumann, Yuxiao Hu: Optimal Vertex Elimination in Single-Expression-Use Graphs
2006-09	Tingting Han, Joost-Pieter Katoen: Counterexamples in Probabilistic Model Checking
2006-10	Mesut Günes, Alexander Zimmermann, Martin Wenig, Jan Ritzerfeld, Ulrich Meis: From Simulations to Testbeds - Architecture of the Hybrid MCG-Mesh Testbed
2006-11	Bastian Schlich, Michael Rohrbach, Michael Weber, Stefan Kowalewski: Model Checking Software for Microcontrollers
2006-12	Benedikt Bollig, Joost-Pieter Katoen, Carsten Kern, Martin Leucker: Replaying Play in and Play out: Synthesis of Design Models from Scenarios by Learning
2006-13	Wong Karianto, Christof Löding: Unranked Tree Automata with Sibling Equalities and Disequalities
2006-14	Danilo Beuche, Andreas Birk, Heinrich Dreier, Andreas Fleischmann, Heidi Galle, Gerald Heller, Dirk Janzen, Isabel John, Ramin Tavakoli Kolagari, Thomas von der Maßen, Andreas Wolfram: Report of the GI Work Group "Requirements Management Tools for Product Line Engineering"
2006-15	Sebastian Ullrich, Jakob T. Valvoda, Torsten Kuhlen: Utilizing optical sensors from mice for new input devices
2006-16	Rafael Ballagas, Jan Borchers: Selexels: a Conceptual Framework for Pointing Devices with Low Expressiveness
2006-17	Eric Lee, Henning Kiel, Jan Borchers: Scrolling Through Time: Improving Interfaces for Searching and Navigating Continuous Audio Timelines
2007-01 *	Fachgruppe Informatik: Jahresbericht 2006
2007-02	Carsten Fuhs, Jürgen Giesl, Aart Middeldorp, Peter Schneider-Kamp, René Thiemann, and Harald Zankl: SAT Solving for Termination Analysis with Polynomial Interpretations
2007-03	Jürgen Giesl, René Thiemann, Stephan Swiderski, and Peter Schneider-Kamp: Proving Termination by Bounded Increase
2007-04	Jan Buchholz, Eric Lee, Jonathan Klein, and Jan Borchers: coJIVE: A System to Support Collaborative Jazz Improvisation

2007-05 Uwe Naumann: On Optimal DAG Reversal

2007-06 Joost-Pieter Katoen, Thomas Noll, and Stefan Rieger: Verifying Concurrent List-Manipulating Programs by LTL Model Checking

2007-07 Alexander Nyßen, Horst Lichter: MeDUSA - MethoD for UML2-based Design of Embedded Software Applications

2007-08 Falk Salewski and Stefan Kowalewski: Achieving Highly Reliable Embedded Software: An empirical evaluation of different approaches

2007-09 Tina Kraußer, Heiko Mantel, and Henning Sudbrock: A Probabilistic Justification of the Combining Calculus under the Uniform Scheduler Assumption

2007-11 Klaus Wehrle: 6. Fachgespräch Sensornetzwerke

2007-12 Uwe Naumann: An L-Attributed Grammar for Adjoint Code

2007-13 Uwe Naumann, Michael Maier, Jan Riehme, and Bruce Christianson: Second-Order Adjoints by Source Code Manipulation of Numerical Programs

2007-14 Jean Utke, Uwe Naumann, Mike Fagan, Nathan Tallent, Michelle Strout, Patrick Heimbach, Chris Hill, and Carl Wunsch: OpenAD/F: A Modular, Open-Source Tool for Automatic Differentiation of Fortran Codes

* These reports are only available as a printed version.
Please contact `biblio@informatik.rwth-aachen.de` to obtain copies.