Combining Monitoring with Run-time Assertion Checking

Stijn de Gouw

Combining Monitoring with Run-time Assertion Checking

Proefschrift

ter verkrijging van

de graad van doctor aan de Universiteit Leiden

op gezag van de Rector Magnificus prof. mr. C.J.J.M. Stolker,

volgens besluit van het College voor Promoties

te verdedigen op woensdag 18 december 2013

klokke 10.00 uur

door

Stijn de Gouw
geboren te 's-Hertogenbosch, in 1985

PhD committee

Promotor:	Prof. Dr. F.S. de Boer	
Co-promotor:	Dr. M. M. Bonsangue	
Other members:		
	Prof. Dr. F. Arbab	
	Prof. Dr. J.N. Kok	
	Prof. Dr. O. Owe	University of Oslo, Norway
	Prof. Dr. K.R. Apt	University of Amsterdam

The work reported in this thesis has been carried out at the Center for Mathematics and Computer Science (CWI) in Amsterdam and Leiden Institute of Advanced Computer Science at Leiden University, under the auspices of the research school IPA (Institute for Programming research and Algorithmics). This research was supported by the European FP7-231620 project HATS on Highly Adaptable and Trustworthy Software using Formal Models.

Copyright © 2013 by Stijn de Gouw. All rights reserved.

Contents

Introduction 1

According to a study in 2002 commisioned by a US Department, software bugs annually costs the US economy an estimated $59 billion[1]. A more recent study in 2013 by Cambridge University estimated that the global cost has risen to $312 billion globally[2].

1.1 Prevention, Isolation and Fixing Bugs

There exists various ways to prevent, isolate and fix software bugs, ranging from lightweight methods that are (semi)-automatic, to heavyweight methods that require significant user interaction. To put our own proposal in the right context, we first briefly look at the main existing approaches. The ones we consider all are based on some form of annotation of the source code of the program by the user. The annotations can also be used in their own right as a form of documentation of the source code (to varying levels of detail, depending on the exact nature of the annotations).

[1] http://web.archive.org/web/20090610052743/http://www.nist.gov/public_affairs/releases/n02-10.htm

[2] http://www.prweb.com/releases/2013/1/prweb10298185.htm

Type-Checking

A relatively successfull and widely adopted method is *type-checking* [73]. The programmer annotates source-code[3], specifying for each variable and function over which values they may range, this is called the type. Subsequently there is a check, the 'type check', which determines whether the values assigned to the variable matches the type of the variable (and similarly for functions and expressions in general) and prevents further compilation and displays a type error when they do not. It is clearly desirable to perform this check early and automate it as much as possible, so that errors in the development development are caught at an early stage. The extend to which automation and early checks are possible depends on the *expressiveness* of the type system: what types can be used. Type systems with limited expressiveness either reject programs which in reality do not contain errors, or accept programs which actually contain errors, but the type system cannot determine this due to the limited expressiveness. However, type systems with strong expressiveness tend to be undecidable, which means roughly that no terminating algorithm exists to perform the type check. Compilers for the most used imperative languages, C, C++ and Java all perform the type checks at compile-time. In general there is a trade-off between the expressiveness of the type system, and the degree of automation of the corresponding type checks. Current research focuses on finding more expressive typesystems which are also efficiently decidable.

Static Verification

Given a formal specification of a program, a static verifier proves (or disproves) whether all executions of the program satisfy that specification. The specification is a formal description that expresses what the program (or parts of the program) is supposed to do. There are two main branches of static verifiers: model checkers and theorem provers (though static type checkers can also been seen as a form of static verification).

[3] There are also type systems which automatically infer types, without requiring annotations. Such type inference is typically supported by compilers for functional languages.

Model checkers do not check whether the given property holds on the actual program. Instead, they determine whether it holds on a (typically finite) model of the program. The most used specification languages in model checking are based on temporal logics [74], which express whether a property holds at certain points in time. A simple example is: whenever a request is made, eventually access will be granted. The model is a simplified version of the program and usually only models are allowed for which it is possible to decide fully automatically whether the property holds. The reason that model checkers work on a model of the program instead of the actual program is that even for seemingly very simple properties (like the halting problem, which asks whether it is possible to decide whether a program terminates), it is undecidable whether that property holds of the actual program. However, since the model is different from the program, this raises the question whether the program satisfies a given property if the model does. There is ongoing research on constructing the smallest possible models which only abstract away parts of the program irrelevant for the given property [53]. Another challenge in this field is the development of algorithms which check the properties as efficiently as possible.

Theorem provers work on the actual program and determine correctness of the program by repeatedly applying proof rules. The problem of determining whether the program satisfies a given property reduces in this setting to checking whether that property is derivable by applying finitely many proof rules. In general, even for very weak specification languages, this will be undecidable (see the next section), though there is a much wider class of specification languages that are semi-decidable (i.e. the true properties of the program are recursively enumerable). For efficiency reasons, usually much user interaction and an in-depth knowledge of the program is needed to guide the proof search. Specification languages used in theorem provers include first-order logic, higher-order logic, dynamic logic and separation logic. These are further discussed in Chapter 3.

Run-Time Checking

Given a program and a specification, a run-time verifier inserts checks in the code which determine whether the specification is satisfied. The check is triggered during an actual execution of the program. Thus in contrast to static verification, where properties are checked with respect to *all* executions (possibly there are infinitely many), run-time checkers only consider a single execution of the program. There is a wide range of specification languages used in run-time verification. They can be partitioned into two categories: languages that focus on the control-flow (these approaches are also called "monitoring"), and those focussing on data-flow.

As an example, one can use regular expressions to specify the order in which functions or methods in a program should be called [21]. Such specifications describe the control-flow of the program. Other formalisms for specifying control-flow are temporal logics, various kinds of automata and context-free grammars. For these formalisms, checking whether a given property holds of the current execution involves parsing a word (where the word is some representation of the trace of method calls in the current execution) in an automata. Generally only formalisms are chosen with a decidable parsing problem (in particular, this is the case for regular expressions, context-free grammars and most automata), so that everything can be automated. Specification languages for monitoring are discussed in more detail in the next chapter.

Approaches that specify data-flow usually do so by annotating the source code with assertions: logical formulas that must be true whenever control passes them. The formulas constrain the values of the program variables. If assertions are expressed in first-order logic with arithmetic, it is in general undecidable due to unbounded quantification (i.e. ranging over an infinite number of values) whether the assertion is true, thus usually the assertions are restricted in some way. For instance, Java contains an `assert`-statement which restricts to quantifier-free formulas (i.e. Boolean expressions). *Design by Contract* [65] provides a systematic way of using assertions to specify classes, interfaces and methods with respectively class invariants and pre- and postconditions. It was first used in the programming language Eiffel, and subsequently has also been applied to many other programming languages. For example, JML [17] is one of the most popular specification languages for Java and supports Design by Contract. JML also supports unbounded quantification, though assertions containing unbounded quantifiers are not checked by the JML run-time assertion checker.

While type checking for the most used imperative languages is done fully automatically at compile-time, run-time checking is done (also fully automatically) during execution, and properties are only checked for the current execution. This generally allows more expressive specifications compared to type checkers. Static verification cannot be automated. In particular, even if one restricts pre- and postconditions to just the formulas *true* and *false*, the resulting specification language is still undecidable (such assertions suffice to express the halting problem).

> Our own proposal is a method for run-time checking of object-oriented programs. We discuss below in more detail how run-time checking applies to the specific context of object-oriented programming, focussing first on single-threaded Java, and then describe an extension to concurrency.

1.2 Object Orientation

Two of the basic features of object-oriented programming are data abstraction and encapsulation. In the design of software, these features support the methodology of *programming to interfaces* [40]. This methodology allows the developer of client code to abstract from irrelevant implementation details. Combined with the *design by contract* principle [65], programming by interfaces is one of the main approaches to mastering the complexity of software today.

One of the main formal behavioral interface specification languages for Java, the Java Modeling Language (JML) [17], is inherently *state-based*; i.e., JML mainly provides support for the specification of classes in terms of their fields, including so-called *model* fields that represent certain aspects of the data structures underlying the implementation. JML does not provide explicit support for the specification of the *interaction* between objects, in contrast to other formalisms such as message sequence charts and UML sequence diagrams [27, 50].

On the other hand, the very semantic foundations of object-oriented programming are defined in terms of sequences of messages. In [52], a *fully abstract* trace semantics for a core Java-like language is given, where traces (or *communication histories*) are (finite) sequences of messages. A fully abstract semantics in general captures the observable behavior abstracting from implementation details. Such an abstraction is required in for example a proper semantic definition of *behavioral subtyping* as is illustrated by the *fragile base class problem* [66]: According to the initial/final state semantics the class B (Figure 1.1) and its revised version in Figure 1.2 below are behaviorally equivalent.

```
class B {
  int x = 0;

  void m() {
    x = x+1;
  }

  void n() {
    x = x+1
  }
}
```

Figure 1.1: First version of a base class B

However the behavior of the subclass `M` defined in Figure 1.3 is clearly different for the two versions of the base class. In fact, when using the revised version of the base class, the definitions of the methods `m` and `n` in the subclass `M` are mutually recursive, giving rise to a non-terminating loop.

It is worthwhile to observe the analogy between this anomaly with repect to the substitutivity of (behaviorally) equivalent classes and the following basic counter-example to the compositionality of the initial/final state semantics for *multi-threaded* programs. Both threads T_1 and T_2 of Figure 1.4 have the same initial/final state semantics, however the initial/final state

```
class B {
  int x = 0;

  void m() {
    this.n();
  }

  void n() {
    x = x+1;
  }
}
```

Figure 1.2: New version of a base class B

```
class M extends B {
  void n() {
    this.m();
  }
}
```

Figure 1.3: Subclass of the base class

semantics of the *interleaving* of T_1 and thread T clearly differs from that of T_2 and T, if assignments are treated atomically.

```
thread T_1 { x=x+1; x=x+1 }
thread T_2 { x= x+2; }
thread T { x=0 }
```

Figure 1.4: Multi-Threaded Programs

This counter-example shows that for a compositional semantics of multi-threaded programs we need more specific information about the underlying implementation, namely information about *how* the final state is generated from the initial state. The *minimal* information needed is captured by a fully abstract semantics (see [67] for a definition of the *full abstraction problem*). In general fully abstract semantics of concurrent systems are based on some form of *trace* semantics. Of interest here is that the above work on fully abstract semantics for a core Java-like language shows that some form of trace semantics is needed even for sequential (single threaded) programs. More specifically, [52] shows that a form of trace semantics for object-oriented programs indeed guarantees substitutivity assuming encapsulation of the object state. Consequently, also the fragile base class problem, as shown above, can only be resolved by some form of trace semantics of behavioral subtyping. In this case, the sequences of internal communication distinguishes the classes in Figure 1.1 and Figure 1.2. Fischer and Wehrheim [37] further investigate behavioral subtyping based on histories for object-oriented languages.

1.3 Extension to Concurrency

The standard Java concurrency model, based on threads and locks, is too low-level, error-prone and insufficiently modular for many applications areas [80]. Instead of extending our run-time checker for single threaded Java programs to the usual multithreading[4], we investigate instead how to run-time check programs that use the actor-like concurrency model of [80]. In that paper, Schaefer et al. extend Java with a concurrency model based on the notion of concurrently running object groups, so-called coboxes, which provide a powerful generalization of the concept of active objects. Coboxes can be dynamically created and objects within a cobox have only direct access to the fields of the other objects belonging to the same cobox. Since one of the main requirements of the design of coboxes is a smooth integration with object-oriented languages like Java, coboxes themselves do not have an identity, e.g., all communication between coboxes refer to the objects within coboxes. Communication between coboxes is based on asynchronous method calls with standard objects as targets. An asynchronous method call spawns a local thread within the cobox to which the targeted object belongs. Such a thread

[4] A simple way to extend our results to standard mutlithreading would consider histories *per* thread (i.e. project the global history upon each thread). This does not require significant modifications in either the theory or the tool described in the next chapters.

consists of the usual stack of internal method calls. Coboxes support multiple local threads which are executed in an interleaved manner. The local threads of a cobox are scheduled cooperatively, along the lines of the Creol modeling language described in [55]. This means that at most one thread can be active in a cobox at a time, and that the active thread has to give up its control explicitly to allow other threads of the same cobox to become active.

The following question arises: how to bridge the gap between the semantic foundations of Java and the abstraction level of formal behavioral interface specification languages like JML? To this end we aim to find a formalism and corresponding tool support which:

1. Integrates properties of the control-flow and data-flow.
2. Is at the same abstraction level as the object-oriented programming model.
3. Is sufficiently expressive.
4. Is user-friendly, i.e., fairly close to the familiar surface syntax of the programming language.
5. Supports automated run-time checking.
6. Adds as little overhead as possible.
7. Contains some form of error reporting.

Outline

Chapter 2 contains a survey of existing formalisms and tools for specifying object-oriented programs.

Chapter 3 presents our own formalism for single-threaded object-oriented programs. The basic notions of a communication view, attribute grammars and assertions in attribute grammars are introduced. The chapter concludes with a motivation for the design choices that were taken during the development of the specification language.

Chapter 4 describes the architecture of SAGA, a tool for run-time checking the previously presented formalism. First, the components of a generic tool architecture are identified. Second, each component is instantiated with different tools which are then evaluated.

Chapter 5 contains two case studies. First we specify a small but very common Java library: a `Stack`. Subsequently we consider a larger industrial case from the e-commerce company Fredhopper. The chapter finishes with an evaluation based on the two cases.

Chapter 6 contains an extension to concurrency. The underlying concurrency model is based on concurrent object groups, also known as coboxes. First, the semantics of concurrent programs, which is based on histories, is formalized. The rest of the chapter explains how such programs can be specified and checked at run-time. To this end, we extend the formalism in Chapter 3 to deal with concurrency, and discuss the corresponding tool support.

In the final Chapter 7 we specify various properties of the previous case studies using the tools PQL, Jassda, LARVA and MOP. We directly compare the results with our own from Chapter 5, dicussing expressivity, learnability and adoptability.

The work reported in this book is based on the following selection of my publications: [30, 29, 32, 34, 31]. Other publications [70, 33, 4] are less relevant in the context of this book.

> "Run-Time Assertion Checking of Data- and Protocol-Oriented Properties of Java Programs: An Industrial Case Study" [29]

Is a journal paper which forms the basis of Chapter 3 and Chapter 5. This paper also introduces the implementation based on aspect-oriented programming as described in Chapter 4.

> "Prototyping a tool environment for run-time assertion checking in JML with communication histories" [30]

Reports on the work in Chapter 3 and the Stack case study in Chapter 5.

> "Run-Time Verification of Black-Box Components using Behavioral Specifications: An Experience Report on Tool Development" [32]

Forms the basis of Chapter 4.

> "Run-time checking of data- and protocol-oriented properties of Java programs: an industrial case study" [34]

Reports on the Fredhopper case study and forms the basis of Chapter 5 and Chapter 7.

> "Run-Time Verification of Coboxes" [31]

Describes the extension to concurrency given in Chapter 6.

Specifying Object-Oriented Programs: Formalisms and Tools 2

In this chapter we give an overview of existing specification languages for object-oriented programs. The specification languages can be roughly partitioned into those which focus on formalizing protocol-oriented properties (all but the last three categories listed below), and those focussing on data. All specification languages for protocol properties are based on some form of histories (also known as traces): sequences of method calls or returns. Languages focussing on data restrict the values of variables and fields in a program by means of logical formulas. We describe whether the specification languages are used in actual tools for static verification or run-time checking.

Sequence Diagrams A sequence diagram[1] shows how multiple objects interact with each other over time. The diagram depicts the messages exchanged between the objects, and the order in which they are sent. In the context of object-oriented programs, the messages in a sequence diagram correspond to method calls. Since sequence diagrams visualize a single in-

[1] See `http://www.omg.org/spec/UML/` for the latest UML specification of sequence diagrams.

teraction, one could select a set of sequence diagrams as a specification of the behaviour of an object-oriented program, by requiring that the methods in the program are executed in the order specified by one of the sequence diagrams in the set. The resulting specification language describes properties of the protocol of the program.

While sequence diagrams have been used in theoretical studies for verification purposes [28, 62], to the best of our knowledge, sequence diagrams as a specification language have not been used in actual tools for static or run-time verification. There are several reasons for this. First, any specification based on visualization tends to become unclear and even infeasible for describing large interactions. Second, the number of interactions exhibited in programs are often unbounded due to loops and recursion. Thus one would need an additional language for characterizing infinite sets of sequence diagrams.

Regular Expressions A regular expression [56] is a declarative notation for a regular language. A language is a set of words. The words are usually (finite) strings of characters, though more complex objects can be used as well. The regular languages are those that can be obtained from a finite language by union, concatenation and Kleene star (an infinite union of finite concatenations of a language). If r_1 and r_2 are regular expressions, the notation for these three operations is respectively $r_1 + r_2$ (union), $r_1 r_2$ (concatenation) and r_1* (Kleene star). As an example, the regular expression $(ab)*$ denote the language of all words starting with "a" in which "a" and "b" alternate. The formal properties of regular languages have been widely studied in the field of formal languages and theory of computation, see for example the books [81, 63].

As a specification language for object-oriented programs, regular expressions can be used to denote valid histories [21]. In this setting, the alphabet symbols correspond to method names, histories are represented as sequences of such alphabet symbols, and the valid histories are the words of the regular language. Note that in contrast to the previous sequence diagrams, regular expressions support a convenient notation for an infinite set of histories with the Kleene star.

There are various tools for run-time checking which support regular expressions: JmSeq [70], Tracematches [3] and JavaMOP [20]. The run-time check corresponds to solving the word problem (or parsing problem): decide whether the history is a word of the language denoted by a given regular expression. This can be done efficiently. In particular, if a history is valid according to a given regular expression, then parsing algorithms exist that decide in constant time whether the history resulting from appending a single call is also valid according to the regular expression (for the full history, this leads to parsing algorithms which are linear in the size of the history), see [41]. Moreover one does not need to store the full history, only the "state" of the parser for the previous history, and the method call which is added to the previous history are needed to determine validity of the new history.

Context-Free Grammars A context-free grammar G is a quadruple $G = \langle V, \Sigma, P, S \rangle$ where V is a set of non-terminals, Σ is a set of terminal symbols, S is the start-symbol of the grammar (a non-terminal), and P is a set of production rules. The production rules specify how each non-terminal (independent of the context in which that non-terminal occurs, hence the name context-free) is allowed to be rewritten into a sequence of terminals and non-terminals. The grammar generates a context-free language, namely the set of all strings of terminal symbols that can be obtained by repeatedly applying the production rules of the grammar, starting from the start symbol of the grammar. For example, the grammar below (the used notation for the grammar is BNF [6]) with the non-terminal S as its start symbol, and "a" and "b" as terminal symbols generates all words of the form $\mathrm{a}^k\mathrm{b}^k$, $k \geq 0$ (in words: k a's, followed by k b's). The symbol ϵ denotes the empty word.

$$\boxed{\begin{array}{lcl} S & ::= & \mathrm{a}\ S\ \mathrm{b} \\ & | & \epsilon \end{array}}$$

Context-free grammars are strictly more expressive than regular expressions. Using the so-called pumping lemma [81], one can prove that there is no regular expression which denotes the same language as the grammar above. However it is more complex to parse a string in a given context-free grammar, than in a regular expression. The currently best known practical algorithms can parse a string of length n in (worst case) $\mathcal{O}(n^3)$ time.

When used as a specification language for object-oriented programs, the terminal symbols are the method names, and the grammar specifies the valid orderings in which these methods are allowed to be called (in other words, the context-free grammar generates the valid histories). The run-time check which decides whether a history is valid consists of parsing the current history in the given grammar. PQL [64] and JavaMOP [20] are examples of tools that support run-time checking based on context-free grammars.

Automata There are too many kinds of automata too list them here exhaustively, but all of them contain at least two things: a notion of a state, and a transition function between states. A finite automaton, one of the simplest automata, contains additionally a set of accepting states and a start state, with the requirement that the set of states must be finite. Finite automata are equivalent in expressive power to regular expressions. A push-down automaton is an extension of a finite automaton with a stack of infinite size. Push-down automata are equivalent in expressive power to context-free grammars.

In general, automata can be seen as a representation of a formal language: it takes a string as input, and accepts or rejects it based on an acceptance condition (the specific acceptance condition varies greatly between the different kinds of automata). However, unlike the above declarative formalisms of regular expressions and context-free grammars, automata tend to have an imperative flavor, focussing on *how* to parse a formal language, as opposed to directly specifying the language itself.

As a specification language for object-oriented programs, JavaMOP [20] supports finite automata. LARVA [26] supports a kind of automata called timed automata with stopwatches.

Temporal Logics Temporal logic [74] is a variant of *Modal Logic* [39]. As the name indicates, the basis for temporal logics is a notion of time on which the truth of a formula may depend. In particular, as the system described a temporal logic formula evolves from one state to the next, the truth value of the formula *can* change. There are many kinds of temporal logics, but they can roughly be classified as being linear-time or branching-time. In linear-time logics, time is viewed as a set of paths (the paths being sequences of "time instances"). LTL [74] is a widely used linear-time logic. Branching-time logics represent time as a tree in which the current time is the root, and the branches are considered as "possible futures". CTL [24] is the main branching-time logic.

Temporal logics have been used extensively in model checking [25], for example in the tools (there are too many others to fully list here): BLAST [46] Java Pathfinder [87] NuSMV [22] PRISM [60] SPIN [49] UPPAAL [11]. Temporal logics have also been used in run-time checking, even for the functional language Haskell [83]. Examples of run-time checkers of temporal logic formulas for Java are JavaMOP [20] and Java Pathfinder [5].

Process Algebras Process algebras [7, 45] have been used to formally model concurrent systems. There exist a wide variety of process algebras (or process calculi), but all approaches share some basic characteristics.

Each approach has a notion of a basic process from which larger processes are built using various operators (for example, for parallel composition, sequential composition and recursion). Message passing is used as the only way two different actors or processes can interact (instead of for example, shared variable concurrency). Finally, all approaches come with a set of algebraic laws (hence the name "process algebra") which for example can be used to show that syntactically different processes are semantically equal (i.e. have the same behavior).

For reference we list some of the most used process algebras here: CSP [48, 1], LOTOS [86], CCS [68], ACP [14] and the more recent π-calculus [69, 78]. CSP has been used in the tool Jass [8] for run-time checking object-oriented programs.

First-Order Logic First-order logic is a formal system for specifying and reasoning about formulas about objects (or values) that range over some domain of discourse. All variables and terms in a first-order formula range over objects of the domain of discourse.

First-order logic can be used to specify programs by means of assertions: a logical formula in which the free variables (i.e. all variables not bound by $\forall$ and $\exists$) are program variables. Assertions are written in the source code of the program and must be true whenever control passes over them. Floyd describes in [38] a method for proving properties using first-order assertions. His work was extended by Hoare in [47]. First-order logic also forms the basis for dynamic logic and second- and higher-order logic described below.

The popular tool-suite for JML [17] supports first-order assertions for both static verification and run-time checking of Java programs. The run-time checker for JML only checks formulas involving bounded quantifiers: quantified variables that range over a finite set of values. Validity of formulas involving unbounded quantifiers is in general undecidable, as already noted in the previous chapter.

Dynamic Logic Like temporal logic, Dynamic Logic (DL) [76, 42] is a variant of *modal logic* [39] which allows the direct expression of program equivalence and weakest preconditions. DL extends full first-order logic with two additional (mix-fix) relations: $< . > .$ (diamond) and $[.] .$ (box). In both cases, the first argument is a *statement*, whereas the second argument is another DL formula. A formula $< s > p$ is true if there exists a terminating execution of s after which the formula p is true. A formula $[s]p$ is true after all terminating executions of s, the formula p is true. For example, the formula <`x=x-1;`> $(x == 0)$ is equivalent to $x = 1$. Dynamic logic has been used as a specification language in the static verifiers KeY [10] and KIV [44].

Second- and Higher-Order Logic Second-Order logic is a highly expressive formalism which allows quantification over predicates and functions over the values of the underlying domain. This contrasts with first-order logic, in which only quantification over values of the domain is allowed. The expressiveness comes at a price: no sound and complete proof systems (with decidable proof rules and axioms) can exist for full second-order logic. Higher-Order logic is a generalization of second-order and first-order logic which allows quantification over objects of an arbitrary higher type (i.e. quantification over predicates of predicates, and so on). There exist various theorem provers for programs that support higher-order logic: Isabelle/HOL [57], Why3 [36], PVS [85] and Coq [15].

Another relatively recent approach is Separation Logic [77], which extensively uses inductively defined predicates (i.e. second-order logic), but adds several non-standard logical connectives to reason about heap properties, such as the separating conjunction and the points-to predicate. These connectives support modularity, though they complicate proof theory (they cannot be axiomatized [18]). Tools that support separation logic for static verification of programs include: VeriFAST [51], jStar [35], Slayer [13] and Smallfoot [12].

Trace Specifications for Control- and Data-Flow 3

The formalisms described in the previous chapter for specifying object-oriented programs can be categorized in roughly two categories: those focussing on the control-flow of the program, and those focussing on the data-flow of the program. Formalisms focussing on the control-flow specify the allowed orderings between method calls, for example using regular expressions, context-free grammars or temporal logics. Formalisms for describing the data-flow generally use assertions to restrict the values of fields, parameters or local variables, possibly enhanced by constructs such as pre-post conditions and class invariants for supporting design by contract. But none of described specification languages were developed to *combine* the specification of the control-flow with the data-flow in a single formalism. In contrast, the behavior of almost all Java programs depends on both control-flow and data-flow: for example, the behavior of a stack is fully characterized by the sequence of method calls to `push` and `pop` it receives (the control-flow), together with the parameter and return values (the data-flow). For Java programs that encapsulate their internal state[1] an execution can be represented by the *global communication history* of the program: the sequence of *messages* corresponding to the invocation and completion of (possibly static) methods, including actual parameters and return values. Similarly, the execution of a single object can be represented by its *local communication history*, which consists of all messages sent and received by that object. The behavior of a program (or object) can then be defined as the set of its allowed histories. Jeffrey and Rathke [52] develop a fully abstract semantics based on histories which coincides with the standard operational semantics.

Let us call the orderings between method-calls and returns the *control-flow* of a history, and the actual parameters and return values the *data-flow* of the history. In this chapter we develop a single formalism which allows combining data-oriented properties of the history with protocol-oriented properties. To be of practical use, such a formalism should be *user-friendly*, amenable to (at least) *automated run-time verification* and sufficiently *expressive*. Below we propose attribute grammars extended with assertions and conditional productions for the specification of histories, and compare several alternatives approaches with respect to expressiveness, usability and automation.

Specifications can be used in two different ways: as a description of how an API (in our case, a set of Java classes and interfaces) must be used by a client (this can be seen as a kind of formalized user manual), or as an internal specification for developers of a class to test the class which is being developed. In the first case, only methods visible to clients can be used in the specification (i.e. public methods and no self-calls, since the user has no control over private methods and self-calls), in the second case for internal use we must also monitor self-calls and calls to private methods.

[1] Encapsulation means that objects do not have direct access to the fields of other objects. If access to a field x is needed, the programmer instead adds two methods `T getx()` and `void setX(T val)`.

3.1 Modeling Framework

The modeling framework consists of three basic ingredients: communication views, grammars with conditional productions, and assertions. We use the interface of the Java `BufferedReader` (Figure 3.1) as a running example to explain these modeling concepts. In particular, we formalize the following property of the `BufferedReader`:

> The `BufferedReader` may only be closed by the same object which created it, and read actions may only occur between the creation and closing of the `BufferedReader`.

Note that the above property constrains the clients that use the `BufferedReader`; in other words, it is a kind of "user manual" for the reader, but does not guarantee that the reader itself works properly (since this property does not restrict the behavior of the reader itself). The property is a little unusual in that the reader actually cannot even detect whether a client uses it according to the above specification, since the reader has no way to detect whether the caller of `close` is the same object that constructed it. This last part can be seen as a form of dynamically checked ownership: the client which created the reader owns it, and the above property can serve as a first step to ensure that no information about the reader is leaked to other clients.

As a naive first step one might be tempted to define the behavior of `BufferedReader` objects simply in terms of 'call-m($\overline{T}$)' and 'return-m($\overline{T}$)' messages of all methods 'm' in its interface, where the parameter types $\overline{T}$ are included to distinguish between overloaded methods (such as `read`). However, interfaces in Java contain only signatures of provided methods: methods where the `BufferedReader` is the callee. Calls to these methods correspond to messages received by the object. In general the behavior of objects also depends on messages sent by that object (i.e. where the object is the caller), and on the particular constructor (with parameter values) that created the object. Moreover it is often useful to select a particular subset of method calls or returns, instead of using calls and returns

```
interface BufferedReader {
  void close();
  void mark(int readAheadLimit);
  boolean markSupported();
  int read();
  int read(char[] cbuf, int off, int len);
  String readLine();
  boolean ready();
  void reset();
  long skip(long n);
}
```

Figure 3.1: Methods of the BufferedReader Interface

```
local view BReaderView specifies java.util.BufferedReader {
  BufferedReader(Reader in) open,
  BufferedReader(Reader in, int sz) open,
  call void close() close,
  call int read() read,
  call int read(char[] cbuf, int off, int len) read
}
```

Figure 3.2: Communication view of a BufferedReader

to all methods (a partial or incomplete specification). Finally in referring to messages it is cumbersome to explicitly list the parameter types. A *communication view* addresses these issues.

Communication View

A communication view is a partial mapping which associates a name to each message. Partiality makes it possible to filter irrelevant events and message names are convenient in referring to messages.

Suppose we wish to formally specify the property on page 23. This is a property which must hold for the local history of all instances of `java.util.BufferedReader`. The communication view in Figure 3.2 selects the relevant messages and associates them with intuitive names: *open*, *read* and *close*.

All return messages and call messages methods not listed in the view are filtered. Note how the view identifies two different messages (calls to the overloaded `read` methods) by giving them the same name *read*. Though the above communication view contains only provided methods (those listed in the `BufferedReader` interface), required methods (e.g. methods of other interfaces or classes) are also supported. Since such messages are sent to objects of a different class (or interface), one must include the appropriate type explicitly in the method signature. For example consider the following message:

```
call void C.m() out
```

If we would additionally include the above message in the communication view, all call-messages to the method `m` of class `C` sent by a `BufferedReader` would be selected and named *out*. In general, incoming messages received by an object correspond to calls of provided methods and returns of required methods. Outgoing messages sent by an object correspond to calls of required methods and returns of provided methods. Incoming call-messages of local histories never involve static methods, as such methods do not have a callee.

Besides normal methods, communication views can contain signatures of constructors (i.e. the messages named *open* in our example view). As such, the set of signatures that occur in a communication view is not necessarily a subset of the signatures in the interface it specifies (since Java interfaces do not contain constructors). In this case, the view selects all calls/returns to an object of a class that implements that interface.

Incoming calls to provided constructors raise an interesting question: what would happen if we select such a message in a local history? At the time of the call, the object has not even been created yet, so it is unclear which `BufferedReader` object receives the message. We therefore only allow return-messages of provided constructors (clearly constructors of other objects do not pose the same problem, consequently we allow selecting both calls and returns to required constructors), and for convenience omit `return`. Alternatively one could treat constructors like static methods, disallowing incoming call-messages to constructors in local histories altogether. However this makes it impossible to express certain properties (including the desired property of the BufferedReader) and has no advantages over the approach we take.

Java programs can distinguish methods of the same name only if their parameter types are different. Communication views are more fine-grained: methods can be distinguished also based on their return type or their access modifiers (such as public). For instance, consider a scenario with suggestively named classes Base and three subclasses Sub1, Sub2 and Sub3, all of which provide a method m. The return type of m in the Base, Sub1 and Sub2 classes is the class itself (i.e. Sub1 for m provided by Sub1). In the Sub3 class the return type is Sub1. To monitor calls to m only with return type Sub1, simply include the following event in the view:

```
call Sub1 C.m() messagename
```

One may ask: why allow private methods to appear in specifications? After all, private methods cannot be used by an outside client of the class. The same question arises when considering whether to monitor self-calls or not. By allowing to monitor private methods and self-calls, the modeling framework and corresponding tool support can also be used by developers of the class, to test the current implementation of the class in development. Communication views include an optional excludeSelfCalls keyword which indicates per event whether self-calls must be tracked (for self-calls, the caller and the callee are the same). While typically developers do not want to exclude self-calls for the purpose of internal tests, this keyword is especially useful in public specifications for other clients, that describe how the class must be used by the client.

Local communication views, such as 3.2, selects messages sent and received by *a single object* of a particular class, indicated by 'specifies java.util.BufferedReader'. In contrast, global communication views select messages sent and received by *any* object during the execution of the Java program. This is useful to specify global properties of a program. In addition to instance methods, calls and returns of static methods can also be selected in global views. Figure 3.3 shows a global view which selects all returns of the method m of a class or interface

```
global view PingPong {
  return void Ping.m() ping,
  call void Pong.m() pong
}
```

Figure 3.3: Global communication view

Constructors
Inheritance
Dynamic Binding
Overloading
Static Methods
Required Methods
Access Modifiers

Table 3.1: Supported Java features that require special care.

(or any of its subclasses) called `Ping`, and all calls to `m` on a subtype of a class or interface called `Pong`. Note that communication views do not distinguish instances of the same class (e.g. calls to 'Ping' on two different objects of class 'Ping' both get mapped to the same terminal 'ping'). Different instances *can* be distinguished in the grammar using the built-in attributes 'caller' or 'callee', see the next two sections.

In contrast to interfaces of the programming language, communication views can contain constructors, required methods, static methods (in global views) and can distinguish methods based on return type or method modifiers such as 'static', or 'public'. See table 3.1 for a list of supported features which require special care. For example, to support dynamic binding, the actual run-time type of the callee must be used, instead of the static type of the variable or field in which the callee is stored. This means that the correspondence between the messages named in the communication view, and actual method calls in the program source code must be made at runtime. The other features listed in the table have been discussed above.

Context-Free Grammars

Now that we have identified the basic messages using the communication view, the question arises how we can specify the valid orderings between these messages: *the protocol*. More specifically, we want to find a notation for the set of the valid histories (where a history is a finite sequence of messages). While the histories in this set will be finite (since at any point during execution, the then current history is finite), the set itself usually contains an infinite number of histories due to recursion or loops, so we cannot simply write it down explicitly. We can consider the set to be a language in which each history is a word, and each message is an alphabet symbol. This suggests we can use existing formalisms for defining languages, in particular the ones surveyed in Chapter 2. We use context-free grammars to specify the protocol behavior of histories.

Definition A history is valid with respect to a given context-free grammar if and only if all prefixes of the history (including the history itself) are generated by the grammar.

The discussion in Section 3.2 provides a motivation for choosing grammars over the other formalisms, and a justification for our definition of a valid history.

The grammar below specifies the valid histories of the `BufferedReader`:

$$\boxed{\begin{array}{lll} S & ::= & \mathit{open}\ C \\ & | & \epsilon \\ C & ::= & \mathit{read}\ C \\ & | & \mathit{close}\ S \\ & | & \epsilon \end{array}}$$

Figure 3.4: Context-Free Grammar which specifies that 'read' may only be called in between 'open' and 'close'.

This grammar describes the prefix closure of sequences of the terminals 'open', 'read' and 'close' as given by the regular expression $((\mathit{open}\ \mathit{read}* \ \mathit{close})*)$. In general, the message names given by a communication view form the terminal symbols of the grammar, whereas the non-terminal symbols specify the structure of valid sequences of messages (in particular, the start symbol S generates the valid histories).

Attribute Grammars and Assertions

While context-free grammars provide a convenient way to specify the *protocol structure* of the valid histories, they do not take data such as parameters and return values of method calls and returns into account. Thus the question arises how to specify the *data-flow* of the valid histories. To that end, we first extend the above context-free grammars with so-called attributes.

Definition Terminal Attributes. Given a terminal T, an attribute of T assigns a value to each instance[2] of T (i.e. to each token of T).

For example, consider a terminal INT_LITERAL, and suppose the string “33” is an instance of INT_LITERAL. One could define an attribute *val* for INT_LITERAL, which assigns the number 33 to the string “33”. Note that terminal attributes can assign different values to different instances of the same terminal.

In the previous section we saw that (instances of) terminals correspond to call or return messages. The question arises: what are sensible attributes for such terminals? Several objects are involved in the sending of the messages: the *caller*, the *callee*, and the actual data being sent in the form of actual parameters or a return value *result.* We define *built-in* attributes (named *callee*, *caller*, and so on) to capture precisely those objects involved in the message. In summary, attributes of terminals are determined (i.e., built-in) from the method signatures given in the communication view.

Next we define attributes for non-terminals. Unlike attributes for terminals, they are defined by the user in the grammar. Given a context-free grammar G and a non-terminal V, let us denote by $L(V)$ the language generated from the non-terminal V by using the productions of G.

[2] A token is a string of symbols. A terminal can be seen as a token type, whose tokens are considered to be syntactically “similar”

Definition Non-terminal Attributes. Given a set of values D and a context-free grammar with a non-terminal V, an attribute for V is a function $f : L(V) \rightarrow D$.

Intuitively the above definition states that a non-terminal attribute assigns values to all of the words generated by that non-terminal. The value of non-terminal attributes is user-defined: the user must associate with each production, source code that computes the attribute values of all non-terminals involved in the production. There are two kinds of non-terminal attributes: synthesized attributes and inherited attributes. In each production the user defines the value of the synthesized attributes of the non-terminal on the left-hand side of the production, and the values of the inherited attributes of the non-terminals appearing on the right-hand side of the production. In general this does not rule out circular attribute definitions The seminal paper [59] in which Knuth first introduced attribute grammars contains an algorithm which detects circular definitions. Using actual source code for the attribute definitions ensures that all attribute values of non-terminals are computable. Of course this source code may not terminate, we rely on the user to make sure that it does.

In our setting, the grammar non-terminals generate sequences of call/return messages. Hence, a non-terminal attribute can be seen as a property of the data-flow of that sequence and hence, as an important special case, the attributes of the start symbol of the grammar can be considered as properties of the data-flow of the history. We are now ready to define attribute grammars:

Definition An attribute grammar is a pair (G, F), where G is a context-free grammar, and F is a set of attributes for G.

Note that the attributes themselves do not alter the language generated by the attribute grammar, they only *define* properties of data-flow of the history. We extend the attribute grammar with assertions to specify properties of attributes. For example, in the attribute grammar in Figure 3.5 a user-defined synthesized attribute ‘c’ for the non-terminal ‘C’ is defined to store the identity of the object which closed the

```
S  ::=  open C1   {assert (open.caller == null ||
                           open.caller ==C1.c ||
                           C1.c == null);}
   |    ε
C  ::=  read C1   (C.c =C1.c;)
   |    close S   (C.c = close.caller;)
   |    ε         (C.c = null;)
```

Figure 3.5: Attribute Grammar which specifies that ‘read’ may only be called in between ‘open’ and ‘close’, and the reader may only be closed by the object which opened it.

`BufferedReader` (and is `null` if the reader was not closed yet). Synthesized attributes define the attribute values of the nonterminals on the left-hand side of each grammar production, thus the ‘c’ attribute is not set in the productions of the start symbol ‘S’. The extension of context-free grammars to attribute grammars with assertions and conditional productions (next called “extended attribute grammars”) naturally gives rise to the following modification in the definition of a valid history.

Definition A history is valid with respect to a given extended attribute-grammar if and only if all prefixes of the history (including the history itself) are generated by the grammar, and all assertions in the grammar were true for every prefix of the history.

The assertion in the attribute grammar of the `BufferedReader` allows only those histories in which the object that opened (created) the reader is also the object that closed it. Throughout the paper the start symbol in any grammar is named ‘S’. For clarity, attribute definitions are written between parentheses ‘(’ and ‘)’ whereas assertions over these attributes are surrounded by braces ‘{’ and ‘}’.

Assertions can be placed at any position in a production rule and are evaluated at the position they were written. Note that assertions appearing directly before a terminal can be seen as a precondition of the terminal, whereas post-conditions are placed directly after the terminal. This is in fact a generalization of traditional pre- and post-conditions for methods as used in design-by-contract: a single terminal ‘call-m’ can appear in multiple productions, each of which is followed by a different assertion. Hence different preconditions (or post-conditions) can be used for the same method, depending on the context (grammar production) in which the event corresponding to the method call/return appears. Traditional pre- and post-conditions are still useful if in every context, the same assertion must be used: in that case, the assertions in the grammar would be duplicated at every occurence of the appropriate terminal. In Section 5.1 we show an example which uses traditional pre- and post-conditions.

It is important to note that for a meaningful semantics we have to restrict the attribute grammars to those grammars which are side-effect free (with respect to the heap) so that they don’t affect the flow of control of the tested program, and which do not involve dereferencing of the built-in attributes of the grammar terminal (the formal parameters of the corresponding methods as specified by the communication view) because these refer to the *current* heap (and not to the past one corresponding to the occurrence of the message). This latter restriction is a fairly natural requirement as the method call which generated the grammar terminal only passed the the object identities of the actual parameters, but not the values of the fields of these objects. Note also that this requirement is automatically satisfied by using encapsulation.

Attribute grammars in combination with assertions cannot express *protocol that depend on data*. To express such protocols we consider attribute grammars enriched by *conditional productions* [72]. In such grammars, a production is chosen only when the given condition (a `boolean` expression over the inherited attributes) for that production is true. Hence conditions are evaluated before any of the symbols in the production are parsed, before synthesized attributes of the non-terminals appearing in the production are set and before assertions are evaluated. In contrast to assertions, conditions in productions affect the parsing process. The Worker grammar in Figure 5.17 in the case study contains a conditional production for the ‘T’ non-terminal.

In summary, a communication view selects and names the relevant messages. Selection allows to focus just on the relevant messages while names allow the identification of different messages, and enable the user to refer to the messages in a user-friendly manner. Context-free grammars specify the allowed orderings of the messages. The terminals of the grammars are the names as introduced by the communication view. These names are not just simple strings, but also contain various attributes such as the sender, receiver and the data sent in the message. The non-terminals are user-defined and generate sets of sequences of messages (i.e. histories), as given by the grammar productions. The start symbol of the grammar generates the valid histories. A context-free grammar can thus be seen as specifying a kind of invariant of the control-flow. Attribute grammars allow defining data properties of sequences of terminals, and in particular of the whole history. To this end, the user defines attributes of the grammar non-terminals in terms of the attributes of the grammar terminals. The values of non-terminal attributes are defined by Java code, which ensures that the attribute definitions are computable. The extension of attribute grammars with assertions makes it possible to specify data-oriented properties of the history, by constraining the value of the non-terminal attributes.

Finally, conditional productions can be used for protocols that *depend* on data. In general, it is possible to specify a single interface or class with multiple communication views (and corresponding grammars). This increases expressiveness: it makes it possible to specify the intersection of two context-free languages (if the user specifies two grammars, the history must satisfy both), and context-free languages are not closed under intersection. Furthermore multiple communication views and grammars can be used as partial specifications for the class or interface, to focussing on a particular behavioral aspect. If it is possible to decompose a single complete specification into multiple partial specifications, the resulting specifications are often simpler. This stems from the fact that a complete specification formalizes various properties, and care must be taken to avoid unwanted interference between these properties. In contrast, partial specifications can be used to formalize each property individually.

3.2 Discussion

We now briefly motivate our choice of attribute grammars extended by assertions as specifications and discuss its advantages over alternative formalisms.

Instead of context-free grammars, we could have selected push-down automata to specify protocol properties (formally these have the same expressive power). Unfortunately push-down automata cannot handle attributes. An extension of push-down automata with attributes results in a kind of Turing machine. From a user perspective, the declarative nature and higher abstraction level of grammars (compared to the imperative and low-level nature of automata) makes them much more suitable than automata as a *specification* language. In fact, a push-down automaton which recognizes the same language as a given grammar is an *implementation* of a parser for that grammar.

Both the BufferedReader above and the case study use only regular grammars. Since regular grammars simplify parsing compared to context-free grammars, the question arises if we can reasonably restrict to regular grammars. Unfortunately this rules out many real-life use cases. For instance, the following grammar in EBNF[3] specifies the valid protocol behavior of a stack:

$$S \quad ::= \quad (\mathit{push}\ S\ \mathit{pop}\ ?)^*$$

It is well-known that the language generated by the above grammar is not regular (apply the pumping lemma for regular languages [81]), so regular grammars (without attributes) cannot be used to enforce the safe use of a stack. It is possible to specify the stack using an attribute which counts the number of pushes and pops:

[3] EBNF is an extension of the usual BNF notation for context-free grammars which allows using the operators on regular expressions (such as the Kleene star ‘*’ and the ‘?’ operator standing for an optional occurrence, i.e., ‘r?’ stands for ‘r + ϵ’) directly inside grammars.

S	::=	S_1 *push*	(`S.cnt = S_1.cnt+1;`)
	|	S_1 *pop*	(`S.cnt = S_1.cnt-1;`) {`assert S.cnt >=0;`}
	|	ϵ	(`S.cnt = 0;`)

The resulting grammar is clearly less elegant and less readable: essentially it encodes (instead of directly expresses, as in the grammar above) a protocol-oriented property as a data-oriented one. The same problem arises when using regular grammars to specify programs with recursive methods. Thus, although theoretically possible, we do not restrict to regular grammars for practical purposes.

Ultimately the goal of run-time checking safety properties is to prevent unsafe ongoing behavior. To do so, errors must be detected as soon as they occur; this is known as *fail-fast*, and the monitor must *immediately* terminate the system: it cannot wait until the program ends to detect errors. In other words, the monitor must decide *after every event* whether the current history is still valid. The simplest notion of a valid history (one which should not generate any error) is that of a word generated by the grammar. One way of fulfilling the above requirement, assuming this notion of validity, is to restrict to prefix-closed grammars. Unfortunately it's not possible to decide whether a context-free grammar is prefix-closed. The following lemmas formalize this result:

Lemma 3.2.1 *Let L_M be the set of all accepting computation histories*[4] *of a Turing Machine M. Then the complement $\overline{L_M}$ is a context-free language.*

Proof See [81].

Lemma 3.2.2 *It is undecidable whether a context-free language is prefix-closed.*

[4] A computation history of a Turing Machine is a sequence $C_0\#C_1\#C_2\#\ldots$ of configurations C_i. Each configuration is a triple consisting of the current tape contents, state and position of the read/write head. Due to a technicality, the configurations with an odd index must actually be encoded in reverse.

Proof We show how the halting problem for M (which is undecidable) can be reduced to deciding prefix-closure of $\overline{L_M}$. To that end, we distinguish two cases:

1. M does not halt. Then L_M is empty so $\overline{L_M}$ is universal and hence prefix-closed.

2. M halts. Then there is an accepting history $h \in L_M$ (and $h \notin \overline{L_M}$). Extend h with an illegal move (one not permitted by M) to the configuration C, resulting in the history $h\#C$. Clearly $h\#C$ is not a valid accepting history, so $h\#C \in \overline{L_M}$. But since $h \notin \overline{L_M}$, $\overline{L_M}$ is not prefix-closed.

Summarizing, M halts if and only if $\overline{L_M}$ is not prefix-closed. Thus if we could decide prefix-closure of the context-free language (lemma 3.2.1) $\overline{L_M}$, we could decide whether M halts.

Since prefix-closure is not a decidable property of grammars (not even if they don't contain attributes) we propose the following alternative definition for the valid histories. A communication history is valid if and only if all its prefixes are generated by the grammar. Note that this new definition naturally fulfills the above requirement of detecing errors after every event. And furthermore this notion of validity is decidable assuming the assertions used in the grammar are decidable. As an example of this new notion of validity, consider the following modification of the above grammar:

$$
\begin{array}{llll}
T & ::= & S & \{\texttt{assert S.cnt >=0;}\} \\
S & ::= & S_1\ \mathit{push} & (\texttt{S.cnt = S\_1.cnt+1;}) \\
 & | & S_1\ \mathit{pop} & (\texttt{S.cnt = S\_1.cnt-1;}) \\
 & | & \epsilon & (\texttt{S.cnt = 0;})
\end{array}
$$

Note that the history *push pop* is a word generated by this grammar, but not its prefix *pop*, which as such will generate an error (as required). Note that thus in general invalid histories are guaranteed to generate errors. On the other hand, if a history generates an error all its extensions are therefore also invalid.

Observe that our approach monitors only safety properties ('prevent bad behavior'), not liveness ('something good eventually happens'). This restriction is not specific to our approach: liveness properties in general cannot be rejected on any finite prefix of an execution, and monitoring only checks finite prefixes for violations of the specification. Most liveness properties fall in the class of the non-monitorable properties [75, 9]. However it *is* possible to ensure liveness properties for terminating programs: they can then be reformulated as safety properties. For instance, suppose we want to guarantee that a method `void m()` is called before the program ends. Introduce the following global view

```
global view livenessM {
  call void C.m() m,
  return static void C.main(String[]) main
}
```

The occurence of the 'main' event (i.e. a return of the main method of the program) signifies the program is about to terminate. Define the EBNF grammar

$$\begin{array}{lll} S & ::= & \epsilon \\ & | & m \\ & | & m+\ main \end{array}$$

(where '+' stands for one or more repetitions). This grammar achieves the desired effect since the only terminating executions allowed are those containing `m`. In local views a similar effect is obtained by including the method `finalize` (which is called once the object will be detroyed) instead of `main`.

Implementation 4

Given a Java interface specified with an attribute grammar, we would like to test whether an object implementing the interface satisfies the properties defined in the grammar at every point in its lifetime. In this chapter we first describe the generic architecture of our tool SAGA [34] which achieves this. Four different components are combined: a state-based assertion checker, a parser generator, a debugger and a general tool for metaprogramming. Traditionally these tools are used for very diverse purposes and don't need to interact with each other. We therefore investigate requirements needed to achieve a seamless integration of these components, motivated by describing the workflow of the run-time checker. In the next section we instantiate the four components with concrete state-of-the-art tools.

Suppose that during execution of a Java program, a method of a class (subsequently referred to as CUT, the 'class under test') which implements an interface specified by an attribute grammar is called. The new history of the object on which the method was called should be updated to reflect the addition of the method call. To represent the history of an object of CUT, the **Meta-Programming** tool generates for each method `m` in CUT two classes `call-m` and `return-m`. These classes con-

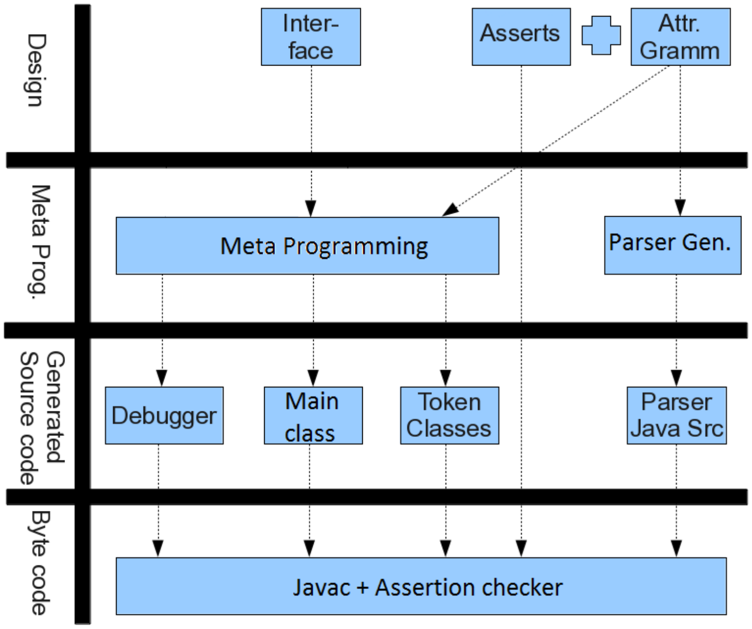


Figure 4.1: Generic Tool Architecture

tain the following fields: the object identitity of the *callee*, the identity of the *caller* and the actual parameters. Additionally `return-m` contains a field `result` containing the return value. A Java `List` containing instances of `call-m` and `return-m` then stores the history of an object of CUT.

The meta-programming tool further generates code for a wrapper class which replaces the original main class. We will refer to this class as the “history class”. This history class contains a field `H`, a Java `map` containing pairs (`id`, `h`) of an object identity `id` and its local history `h`. Moreover it stores the current values of the synthesized attributes of the start symbol, these can be used in assertion languages supporting design by contract (See Section 5.1 for an example of this usage). The history class executes the original program inside the **Debugger**. The Debugger is responsible for monitoring execution of the program. It must be capable of temporarily ‘pausing’ the program whenever a call or return occurs, and execute user-defined code to update `H` appropriately . Moreover the Debugger must be able to read the identity of the callee, caller and parameters/return-value.

After the history is updated the run-time checker must decide whether it still satisfies the specification (the attribute grammar). Observe that a communication history can be seen as a sequence of tokens (in our setting: communication events). Since the attribute grammar together with the assertions generate the language of all valid histories, checking whether a history satisfies the specification reduces to deciding whether the history can be parsed by a parser for the attribute grammar, where moreover during parsing the assertions must evaluate to true. Therefore the **Parser Generator** creates a parser for the given attribute grammar. Since the history is a heterogenous list of `call-m` and `return-m` objects, the parser must support parsing streams of tokens with user-defined types. Assertions in general describe properties of Java objects, and the grammar contains assertions over attributes, the attributes must be normal Java variables. Consequently the parser generator must allow arbitrary user-defined java code (to set the attribute value) in rule actions. The use of Java code ensures the attribute values are computable. Since assertions are allowed in-between any two (non)-terminals, the parser generator should support user-defined actions between arbitrary grammar symbols. At run-time, the parser is triggered whenever the history of an object is updated. The result is either a parse error, which indicates that the current communication history has violated the protocol structure specified by the attribute grammar, or a parse tree with new attribute values. During parsing, the **Assertion Checker** evaluates the assertions in the grammar on the newly computed attribute values. To avoid parsing the whole history of a given object each time a new call or return is appended, ideally the parser should support incremental parsing [43]. An incremental parser computes a parse tree for the new history based on the parse trees for prefixes of the history. In our setting, the attribute grammar specifies invariant properties of the ongoing behavior. Hence the parser constructs a new parse tree after each call/return, consequently parse trees for all prefixes of the current history can be exploited for incremental parsing.

To illustrate how the tools described above interact with each other at run-time, the UML sequence diagram in Figure 4.2 shows the run-time environment of a successful method invocation of a (single-threaded) Java program, containing a class Class Under Test (CUT) whose local history is specififed by an attribute grammar. The actors in the sequence diagrams are:

- 'User Prog': A client class that instantiates and uses CUT.
- 'Debugger': Java debugger that intercepts all method calls and corresponding returns from 'User Prog' to CUT.
- 'History (instance)': an instance of the history class. This class stores the local history of each object of CUT.
- 'Parser': an instance of a parser for the given attribute grammar. The source code of the Parser was generated by the Parser Generator.
- 'Assertion Checker': provides facilities to check assertions at run-time.
- 'Class Under Test (CUT)': The class which was specified using an attribute grammar.
- 'stderr': the standard error stream of the system. Error reports (such as an assertion failure or protocol violation) can be sent to this stream.

Figure 4.3 shows a scenario in which a method return causes the updated history to violate the grammar rules. In this case, the parser detects a parse error and outputs a protocol violation to 'stderr'. The scenario in which parsing is successfull, but the assertions cause an error, is not shown but very similar.

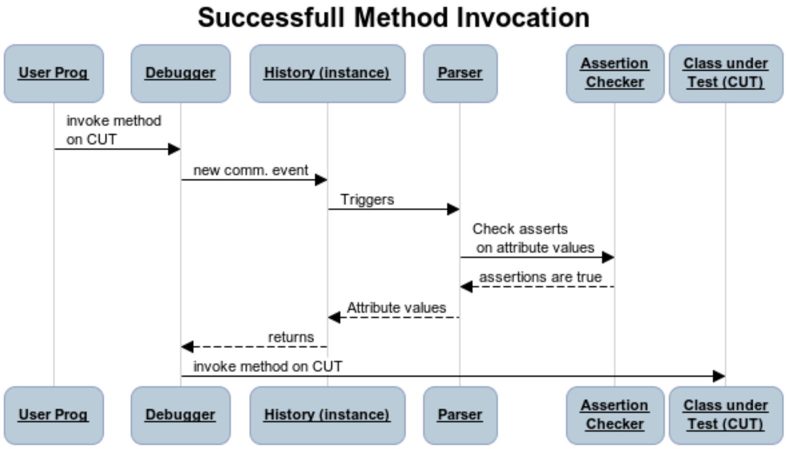


Figure 4.2: Run-time environment of successfull method invocation

4.1 Instantiating the Tool Architecture

The previous section introduced the generic tool architecture, which was based on four different components: meta-programming, debugger, parser generator and state-based run-time assertion checker. Here we instantiate these four components with particular (state of the art) tools, and report our experiences to what extent the requirements stated in the previous section are satisfied by these current tools. The main overhead of the run-time checker is caused by the parser, hence we discuss performance (both theoretical and in practice) in the paragraph on parser generators.

Meta-Programming Rascal [58] is a tool-supported domain specific language for meta programming. We use its parsing, source code analysis, source-to-source transformation and source code generation features. A ± 1000 line Rascal program[1] takes care of:

[1] Excluding the grammar for Java.

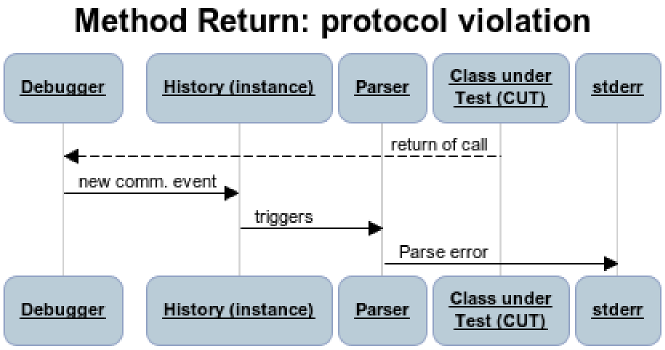


Figure 4.3: Run-time environment of successfull method invocation

- parsing and analyzing the Java method signatures in the communication view.
- generating Java source for a debugger. The debugger should intercept any method call and return, and inform the History class that an event occured.
- generating the token classes `call-m` and `return-m` for each call and return event in the view.
- generating the History class, which specifically accepts new events from the provided methods in the interface and acts as a token stream for the generated parser.

The full source code which Rascal generates for the above tasks contains about 50 times the number of events + 100 lines of code, in other words, the size of the generated code depends mainly on the number of events in the communication view.

Note that we require general meta programming features for several input languages, not just Java. This application of Rascal has three languages as input (ANTLR grammars, View declarations and Java), and one output language (Java). Rascal runs on a JVM, such that it integrates into any Java environment.

In the following Rascal snippet we generate update methods in the history class which are called whenever a method returns.

```
return "
<for (`<mods> <return> <id> (<formals>)` <- methods) {
  r = "return_<id>";>
public void update(return_<id> e) {
<if (r in tokens){>
  e.setType(<grammarName>Lexer.<tokens[r]>);
  addAndParse(e);<}>
}
<}>";
```

This return statement contains three levels. The Rascal language level (in boldface) provides the return statement, the string, and embedded in the string expressions marked by <...> angular brackets. The string that is generated represent an (unparsed) Java fragment. The fragments embedded in back ticks (`) represent parsed Java fragments from the input interface. Inside those fragments Rascal expressions occur again between angular brackets.

The string template language of Rascal allows us to instantiate a number of methods called `update` using a `for` loop and an `if` statement. The data that is used in the for loop is extracted directly from the parse trees of the methods in a Java interface file. The concrete Java source pattern between the back ticks (`) matches the declaration of a method in the interface, extracting the name of the method (`<id>`). Note that this snippet uses variables declared earlier, such as `tokens` which is a map from method names to token names taken from the view declaration in the interface and `grammarName` which was also extracted from the view earlier. Albeit complex code due to the many levels required for this task, the code is short and easy to adapt to other kinds of analysis and generation patterns.

The main disadvantages of Rascal are that it is still in an alpha stage, it is not fully backwards compatible and we discovered numerous bugs in Rascal during development of the Rascal program. However overall our experience was quite positive. The identified bugs were fixed quickly by the Rascal team, and its powerful parsing, pattern matching and transforming concrete syntax features proved indispensable.

Debugger We evaluated Sun's implementation of the Java Debugging Interface for the debugger component. It is part of the standard Java Development Kit, hence maintenance of the debugger is practically guaranteed. The Sun debugger starts the original user program in separate a virtual machine which is monitored for occurences of `MethodEntryEvent` (method calls) and `MethodExitEvent` (method returns). It allows defining event handlers which are executed whenever such events occur. It also allows retrieving the caller, callee, parameters values and return value of events using `StackFrames`. No actual Java source code for the class under test is needed for the debugging. The approach is safe in that no source code nor bytecode is modified for the monitoring. The Sun debugger meets all requirements for the debugger stated above. As the main disadvantage, we found that the current implementation of the debugger is very slow. In fact it was responsible for the majority of the overhead of the run-time checker. This is not necessarily problematic: as testing is done during development, the debugger will typically not be present in performance critical production code. Moreover, one usually wants to test only up to a certain bound (for instance, in time, or in the number of events), and report on results once the bound is exceeded. Nonetheless, for testing up to huge bounds, a different implementation for the debugger is needed.

As an alternative we have also tested AspectJ, a Java compiler which supports aspect-oriented programming. Aspect-oriented programming is tailored for monitoring. AspectJ can intercept method calls and returns conveniently with pointcuts, and weave in user-defined code (advices) which is executed before or after the intercepted call. In our case the pointcuts correspond to the calls and returns of the messages listed in the communication view. The advice consists of code which updates the history. The code for the aspect is generated from the communication view automatically by the Rascal meta-program. Advice can either be woven into Java source code, byte code or at class load-time fully automatically by AspectJ. Note that in contrast to the above Java Debugger approach this step involves changing the source or bytecode, which may be deemed as less safe. We use the inter-type declarations of AspectJ to store the local history of an object as a field in the

```
    /* call int read(char[] cbuf, int off, int len); */
before(Object clr, BufferedReader cle,
      char[] cbuf, int off, in len):
 (call( int *.read(char[], int, int))
  && this(clr) && target(cle) && args(cbuf, off, len)
  && if(BReaderHistoryAspect.class.desiredAssertionStatus() ))
 {
   cle.h.update(new call_push(clr, cle, cbuf, off, len));
 }
```

Figure 4.4: Aspect for the event ‘call int read(char[] cbuf, int off, int len)’

object itself. This ensures that whenever the object goes out of scope, so does its history and consequently reduces memory usage. Clearly the same does not hold for global histories, which are stored inside a separate Aspect class. Figure 4.4 shows a generated aspect. The second and third line specify the relevant method, in this case `BufferedReader.read`. The fourth line binds variables (‘clr’, ‘cle’, ...) to the appropriate objects. Note that to support dynamic binding, it is not possible to statically match method calls to in the Java source to the below pointcut: the dynamic type of the callee, which is determined at run-time, determines whether the pointcut matches. The fifth line ensures that the aspect is applied only when Java assertions are turned on. Assertions can be turned on or off for each communication view individually. The fifth line contains the advice that updates the history. Note that since the event came was defined in a local view, the history is treated as a field of the callee and will not persist in the program indefinitely but rather is garbage collected as soon as callee object itself is.

As a third alternative, we also tested the meta-programming tool Rascal to generate code which intercepts the method calls and returns appropriately. This can be done by defining a transformation on the actual Java source code of the class under test, which requires a full Java grammar (which must be kept in sync with the latest updates to Java). To capture the identity of the callee, parameter values and return value of a method, one only needs to transform that particular method (i.e. locally). But inside the method there is no way to access the identity of the caller. Java does offer facilities to inspect stack frames, but these frames contain only static entities, such as the name of the method which called the currently executing method, or the type of the caller, but not the caller itself. To capture the caller, a global transformation at all call-sites is needed (and in particular one needs to have access to the source code of *all* clients which call the method). The same problem arises in monitoring calls to required methods.

Finally it proved to quickly get very complex to handle all Java features listed in Table 3.1. We wrote an initial version of a weaver in Rascal which already took over 150 lines (over half of the full checker at the time) without supporting method calls appearing inside expressions, inheritance, dynamic binding, constructors and overloading. Moreover the meta-programming approach is also unsuitable if the Java source code is not available (which happens frequently for libraries) ing where only byte code is available, limiting the applicability of the tool. In summary, while it is possible to implement monitoring by defining a code transformation in Rascal, this rules out bytecode only libraries, and quickly gets complex due to the need for a full (up to date) Java grammar and the complexity of the full Java language.

Parser Generator For the the parser generator component we tested ANTLR v3, a state of the art parser generator. It generates fast recursive descent parsers for Java and allows grammar actions and custom token streams. It even supports conditional productions: productions which are only chosen during parsing whenever an associated Boolean expression (the condition) is true and allow for a degree of context-senstitiveness. Attribute grammars with conditional productions express protocols that depend on data which are typically not context-free. ANTLR also supports EBNF, a notation grammars which extends context-free grammars with the operations from regular expressions, for example the Kleene star. Though EBNF does not strictly increase expressiveness (the language generated by such grammars is still context-free), it is convenient for practical purposes: sometimes a regular expression is simpler and more natural than a full-fledged grammar.

Due to the power of general context-free grammars extended with attributes (as introduced in the seminal paper [59] by Knuth), they can be quite expensive to parse. In particular, the currently best known algorithm [84] to parse context-free grammars has a time complexity of $\mathcal{O}(n^{2.38})$ (with very huge constants), where n is the number of terminals to parse. The current best practical algorithms (with reasonably sized constants) require cubic time. Clearly parsing n tokens cannot be done in less than $\mathcal{O}(n)$ steps, since the entire input must be read. Besides this trivial linear lower bound, no non-trivial lower bounds are known [41], though Lee [61] showed that multiplication of two square Boolean matrices can be reduced at a certain cost to parsing context-free grammars. In particular, she showed that if parsing n tokens can be done in $\mathcal{O}(n^{3-\epsilon})$ steps, then we can multiple two n by n Boolean matrices in $\mathcal{O}(n^{3-(\epsilon/3)})$ steps, with small constants. This means that any practical (i.e. small constants) sub-cubic parsing algorithm also can be used as a practical sub-cubic matrix multiplication algorithm. However no such fast practical algorithm is known for matrix multiplication.

ANTLR avoids the cubic-time parsing inefficiency by only supporting LL(*) grammars[2]. Due to the restriction, the parsing algorithm used by ANTLR is for most grammars linear, and quadratic in the worst case. A major disadvantage of ANTLR is that it lacks support for incremental parsing: each time the history is updated (i.e. a single terminal is added), the full history has to be reparsed. Additionally the full history has to be saved. Support for incremental parsing is planned by the ANTLR developers. We have not been able to find any Java parser generator which supports incremental parsing of attribute grammars.

Assertion Checker We tested two state-based assertion languages: standard Java assertions and the Java Modeling Language (JML). Both languages suffice for our purposes. A Java

[2] A strict subset of the context-free grammars. Left-recursive grammars are not LL(*).

assertions is a statement `assert b;` where `b` is a standard `boolean` expressions. As a consequence, note that Java assertions can contain calls to methods that return a `boolean`. Though Java assertions can not contain quantifiers, it is to some degree possible to simulate those using a method containing a loop. Java does not enforce assertions to be side-effect free: one needs to check manually that only 'pure' assertions are used.

JML is far more expressive than the standard Java assertions. It allows unbounded quantification, in general any first-order formula can be expressed in JML, and supports Design by Contract (see also Section 5.1). JML also ensures that assertions are side-effect free. Unfortunately the JML tool support is not ready yet for industrial usage. In particular, the last stable version of the JML run-time assertion checker dates back over 8 years, when for instance generics were not supported yet. The main reason is that JML's run-time assertion checker only works with a proprietary implementation of the Java compiler, and unsurprisingly it is costly to update the proprietary compiler each time the standard compiler is updated. This problem is recognized by the JML developers [19]. OpenJML, a new alpha version of the JML run-time assertion checker integrates into the standard Java compiler, and initial tests with it provided many valuable input for real industrial size applications. See the Sourceforge tracker of OpenJML at `http://sourceforge.net/tracker/?group_id=65346&atid=510629` for the kind of issues we have encountered when using OpenJML.

Case Studies 5

In this chapter we use the formalism described in chapter Chapter 3 and the extension to design by contract described in chapter Chapter 4 to specify a Java library, and an industrial-sized case from the e-commerce company Fredhopper. The Java library we consider is a (last-in-first-out) Stack. The Stack example illustrates how the *Design by Contract* methodology as supported by JML can be used to specify the `push` and `pop`-methods purely in terms of histories in an elegant manner. In particular, this example shows how synthesized attributes of the start-symbol can be used conveniently inside method pre- and postconditions. Based on the case study, we discuss our experiences with SAGA.

5.1 Design by Contract: Stack

A Stack is an abstract data type which has only two operations push and pop. The operation push adds an object to the stack, while pop returns and removes the last element from the stack which was pushed but not yet removed. The operation pop is not allowed on an empty Stack. Figure 5.1 shows an interface for the Stack in Java.

```
public interface Stack {
        void push(Object item);
        Object pop();
}
```

Figure 5.1: Stack Interface

Our task is to find a specification for the Stack which ensures that pop is never called by the user on an empty stack, and moreover that pop returns the right object when called on a non-empty stack. The communication view in Figure 5.2 selects three events. The returns of push are needed to keep track of the elements which have been pushed onto the Stack. Note that it would incorrect be to consider the calls to push instead: suppose some strange implementation of push would itself call pop as its first action, before restoring the removed element and adding the element which was passed to push. Then calling push on an empty stack would fail (since that results in calling pop on an empty stack), but the history would be 'PUSH POP' (which seemingly looks valid for a Stack). Selecting returns of push avoids this problem. The calls to pop, which are referred to by the terminal 'POP', are needed to ensure that pop is never called on an empty Stack. In this case it would not suffice to track only returns of pop, since whenever pop is executed on an empty stack, the run-time checker would only detect the failure after executing of pop (which fails), and thus does not *prevent* unsafe behavior.

The protocol behavior of this view can be defined in terms of sequences of the *terminals* 'PUSH' and 'POP' generated by the context-free grammar given in Figure 5.3, where 's' is the

```
local view StackHistory specifies Stack {
       return push PUSH;
       call pop POP;
}
```

Figure 5.2: Communication View of a Stack

start symbol.

$$
\begin{array}{lcl}
\mathrm{s} & ::= & \mathrm{PUSH}\ s \\
 & | & s\ s \\
 & | & b \\
\mathrm{b} & ::= & \mathrm{PUSH}\ \ b\ \mathrm{POP} \\
 & | & \epsilon \\
 & | & b\ b
\end{array}
$$

Figure 5.3: Abstract Stack Behavior

The non-terminal ‘s’ generates the *prefix closure* of the standard grammar for *balanced* sequences of ‘PUSH’ and ‘POP’ (which are generated by the non-terminal ‘b’). This ensures that `pop` is never called on an empty stack.

In order to specify the relation between the actual parameters of calls to the `push` method and the return values of the `pop` method, we introduce a synthesized attribute ‘stack’ of type `JMLListValueNode` for the non-terminal ‘s’. `JMLListValueNode` is a JML class for a singly-linked list with *side-effect free* implementations of the methods `JMLListValueNode append(Object item)` , which appends an item to the list, and `JMLListValueNode concat(JMLListValueNode ls2)` which concatenates two lists. The intended value of the ‘stack’ attribute is a list of the elements which are pushed but have not yet been popped. Since balanced Stacks are empty, associating the ‘stack’ attribute also to the b-non-terminal would be redundant. Figure 5.4 shows how ‘stack’ is updated in each production of the non-terminal s. Intuitively the value of ‘stack’ at the root of the parse-tree (i.e. an occurence of the start-symbol s) is a list containing the current contents of the

Stack. Figure 5.5 shows the parse tree for the history resulting from the program `s.push(5); s.push(7); s.pop();`. Note that this does not mean that an actual implementation of the stack interface works correctly: the attribute grammar can be considered as a 'reference implementation' of the stack, but we still need to ensure that an actual implementation of the Stack matches (in the sense that calling `pop` returns the right value) this reference implementation.

$$\begin{array}{llll}
\text{s} & ::= & \text{PUSH } s_1 & (\texttt{stack =}s_1\texttt{.stack.append(PUSH.item);}) \\
 & | & s_1\ s_2 & (\texttt{stack =}s_1\texttt{.stack.append(}s_2\texttt{.Stack);}) \\
 & | & \text{b} & (\texttt{stack = stack.clear();}) \\
\text{b} & ::= & \text{PUSH } \ b \text{ POP} & \\
 & | & \epsilon & \\
 & | & b\ b &
\end{array}$$

Figure 5.4: Attribute Grammar Stack Behavior

In order to specify the method contracts for the Stack, the JML implementation of SAGA (described in Section 4.1) allows referring to the synthesized attributes of the root of the parse tree. Since the start symbol in the parse tree gener-

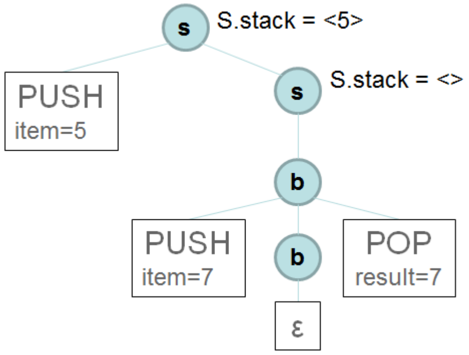


Figure 5.5: Parse tree annotated with attribute values for the history push(5) push(7) pop() in the grammar of Figure 5.4 (irrelevant attributes ommitted)

ates the whole history, intuitively the synthesized attributes of the start symbol can be thought of as a property of the entire history. In order to use the attribute ‘stack’ of this grammar in assertions for specifying the contracts of the push and pop methods of the ‘Stack’ interface (Figure 5.1) in terms of communication histories, the modeling framework provides a class StackHistory which corresponds to the communication view of Figure 5.2. This class contains a ’getter’ method JMLListValueNode stack() which retrieves the value of the attribute ‘stack’ of the root of the parse tree of the current history.

```
interface Stack {
  //@ public model instance StackHistory history;

  //@ ensures history.stack().equals(
  //@           \old(history.stack()).append(item));
  void push(Object item);

  //@ ensures history.stack().equals(
  //@           \old(history.stack()).tail());
  //@ ensures \result == \old(history.stack()).head();
  Object pop();
}
```

Figure 5.6: JML Specification Stack Interface

Figure 5.6 illustrates how the StackHistory class can be used to specify the desired contracts. The JML keyword model indicates that history (of type StackHistory) can be used only in specifications. The keyword instance specifies that history will be added as a (non-static) field to any class that implements the Stack interface. The ensures and requires clauses specify the method contracts in terms of the ‘stack’ attribute (whose value is defined in the attribute grammar). Summarizingly, the property that pop may not be called on an empty stack is ensured by the productions of the grammar (the grammar productions can be considered to be an interface invariant for the protocol behavior), and the property that pop returns the right object is guaranteed by the method contracts and the definition of the attribute ‘stack’.

Note that alternatively we could have avoided the method contracts by instead adding appropriate assertions in the attribute grammar before and after *every* occurence of 'PUSH' and 'POP' in the grammar. This leads to duplication since 'PUSH' occurs multiple times in the grammar. Moreover, for this alternative solution, we should also have added to the communication view that we intend to capture returns of `pop`: otherwise there would be no way to check that `pop` returned the right value. For the above example, we favour the above design-by-contract solution over the assertions-in-grammar, since it avoids duplication of specifications and additionally avoids adding the extra terminal for returns of `pop`. This increases readability of the grammar, and results in less overhead for the run-time check since the sequence of tokens to parse is shorter.

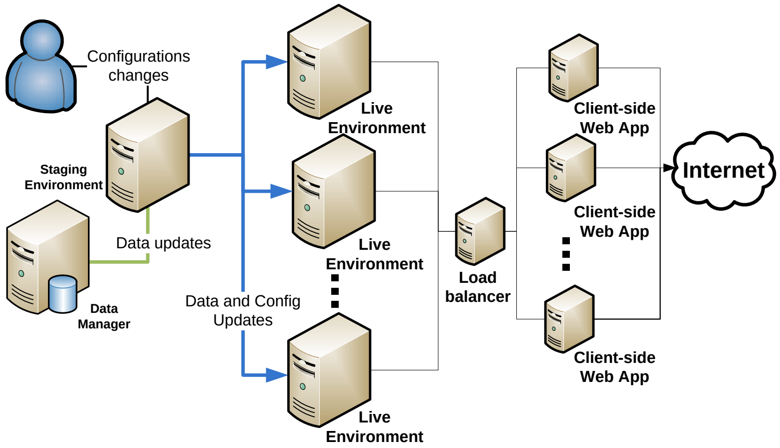


Figure 5.7: An example FAS deployment

5.2 Fredhopper Case-Study

Fredhopper[1] is a search, merchandising and personalization solution provider, whose products are tailored to the needs of online businesses. Fredhopper operates behind the scenes of more than 100 of the largest online shops [2]. It provides the Fredhopper Access Server (FAS), which is a distributed concurrent object-oriented system that provides search and merchandising services to eCommerce companies. Briefly, FAS provides to its clients structured search capabilities within the client's data. Each FAS installation is deployed to a customer according to the FAS deployment architecture (See Figure 5.7).

FAS consists of a set of live environments and a single staging environment. A live environment processes queries from client web applications via web services. FAS aims at providing a constant query capacity to client-side web applications. A staging environment is responsible for receiving data updates in XML format, indexing the XML, and distributing the resulting indices across all live environments according to the *Replica-*

[1] http://www.sdl.com/products/fredhopper/

[2] http://www.sdl.com/campaign/wcm/gartner-maqic-quadrant-wcm-2013.html?campaignid=70160000000fSXu

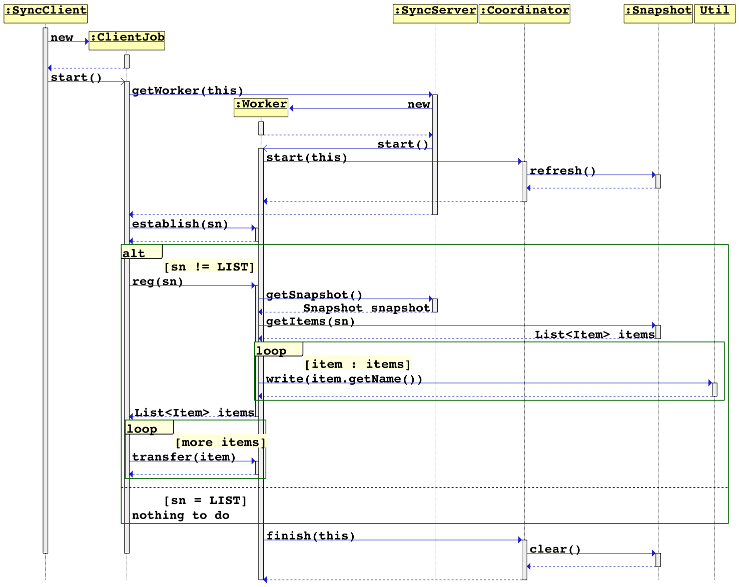


Figure 5.8: Replication interaction

tion Protocol. The Replication Protocol is implemented by the *Replication System*. The Replication System consists of a *SyncServer* at the staging environment and one *SyncClient* for each live environment. The SyncServer determines the schedule of replication, as well as its content, while SyncClient receives data and configuration updates according to the schedule.

Replication Protocol

The SyncServer communicates to SyncClients by creating *Worker* objects. Workers serve as the interface to the server-side of the Replication Protocol. On the other hand, SyncClients schedule and create *ClientJob* objects to handle communications to the client-side of the Replication Protocol. When transferring data between the staging and the live environments, it is important that the data remains *immutable*. To ensure immutability without interfering the read and write accesses of the staging environment's underlying file system,

```
interface Snapshot {
 void refresh();
 void clear();
 List<Item> items(String sn);
}

interface Worker {
 void establish(String sn);
 List<Item> reg(String sn);
 void transfer(Item item);
 SyncServer server();
}
```

Figure 5.9: SnapShot and Worker interfaces of Replication System

the SyncServer creates a *Snapshot* object that encapsulates a snapshot of the necessary part of the staging environment's file system, and periodically *refreshes* it against the file system. This ensures that data remains immutable until it is deemed safe to modify it. The SyncServer uses a *Coordinator* object to determine the safe state in which the Snapshot can be refreshed. Figure 5.8 shows a UML sequence diagram concerning parts of the replication protocol with the interaction between a SyncClient, a ClientJob, a Worker, a SyncServer, a Coordinator and a Snapshot. the diagram also shows a *Util* class that provides static methods for writing to and reading from `Stream`. The figure assumes that SyncClient has already established connection with a SyncServer and shows how a ClientJob from the SyncClient and a Worker from a SyncServer are instantiated for interaction. For the purpose of this paper we consider this part of the Replication Protocol as a *replication session.*

In this section we show how to modularly decompose object interaction behavior depicted by the UML sequence diagram in Figure 5.8 using SAGA. Figures 5.9 and 5.10 shows the corresponding interfaces and classes, note that we do not consider SyncClient as our interest is in object interactions of a replication session, that is after `ClientJob.start()` has been invoked.

The protocol descriptions and specifications considered in this case study have been obtained by manually examining the

```
interface SyncServer {
 Snapshot snapshot();
}

interface Coordinator {
 void start(Worker t);
 void finish(Worker t);
}

class Util {
 static void write(String s) { .. }
}
```

Figure 5.10: SyncServer and Coordinator interfaces of Replication System

behavior of the existing implementation, by formalizing available informal documentations, and by consulting existing developers on intended behavior. Here we first provide such informal descriptions of the relevant object interactions:

- Snapshot: at the initialization of the Replication System, `refresh` should be called first to refresh the snapshot. Subsequently the invocations of methods `refresh` and `clear` should alternate.
- Coordinator: neither of methods `start` and `finish` may be invoked twice in a row with the same argument, and method `start` must be invoked before `finish` with the same argument can be invoked.
- Worker: `establish` must be called first. Furthermore `reg` may be called *if* the input argument of `establish` is not “LIST” but the name of a specific replication schedule, and that `reg` must take that name as an input argument. When the `reg` method is invoked and before the method returns, the Worker must obtain the replication items for that specific replication schedule via method `items` of the Snapshot object. The Snapshot object must be obtained via method `snapshot` of its SyncServer, which must be obtained via the method `server`. It must notify the name of each replication item to its

```
local view SnapshotHistory
grammar Snapshot.g
specifies Snapshot {
 call void refresh() rf,
 call void clear() cl
}
```

Figure 5.11: Snapshot Communication View

```
local view CoordinatorHistory
grammar Coordinator.g
specifies Coordinator {
 call void start(Worker t) st,
 call void finish(Worker t) fn
}
```

Figure 5.12: Coordinator Communication View

```
local view WorkerHistory grammar Worker.g
specifies Worker {
 call void establish(String sn) et,
 call List<Item> reg(String sn) rg,
 return List<Item> reg(String sn) is,
 call void transfer(Item item) tr
}
```

Figure 5.13: Worker Communication View

interacting SyncClient. This notification behavior is implemented by the static method `write` of the class `Util`. The method `reg` also checks for the validity of each replication item and so the method must return a subset of the items provided by the method `items`. Finally `transfer` may be invoked after `reg`, one or more times, each time with a unique replication item, of type `Item`, from the list of replication items, of type `List<Item>`, returned from `reg`.

Figures 5.11 to 5.14 specifies communication views. They provide partial mappings from message types (method calls and returns) that are local to individual objects to grammar terminal symbols. Note that the specification of the Worker's behavior is modularly captured by two views: `WorkerHistory` and

```
local view WorkerRegHistory grammar WorkerReg.g
specifies Worker {
 call List<Item> reg(String sn) rg,
 return List<Item> reg(String sn) is,
 return Snapshot SyncServer.snapshot() sp,
 call List<Item> Snapshot.items(String sn) ls,
 return List<Item Snapshot.items(String sn) li,
 call static void Util.write(String s) wr
}
```

Figure 5.14: WorkerReg Communication View

$$\begin{array}{lcll} S & ::= & \epsilon & \mid rf\ T \\ T & ::= & \epsilon & \mid cl\ S \end{array}$$

Figure 5.15: Snapshot Attribute Grammar

$$\begin{array}{lllll} S & ::= & & T & (T\texttt{.ts} = \texttt{new HashSet();}) \\ T & ::= \epsilon & \mid & st & \{\texttt{assert !}\ T\texttt{.ts.contains(}st\texttt{.t);}\} \\ & & & \multicolumn{2}{l}{(T\texttt{.ts.add(}st\texttt{.t);})\ T_1\ (T_1\texttt{.ts} = T\texttt{.ts;})} \\ & & \mid & fn & \{\texttt{assert}\ T\texttt{.ts.contains(}fn\texttt{.t);}\} \\ & & & \multicolumn{2}{l}{(T\texttt{.ts.remove(}fn\texttt{.t);})\ T_1\ (T_1\texttt{.ts} = T\texttt{.ts;})} \end{array}$$

Figure 5.16: Coordinator Attribute Grammar

`WorkerRegHistory`. The view `WorkerHistory` exposes methods **`establish`**, **`reg`** and **`transfer`**. Using this view we would like to capture the overall valid interaction in which Worker is the callee of methods, and at the same time the view helps abstracting away the implementation detail of individual meth-

$$\begin{array}{llll} S & ::= \epsilon & \mid & et\ T \quad (T\texttt{.d} = et\texttt{.sn;}) \\ T & ::= \epsilon & \mid & \{\texttt{!"LIST".equals(}T\texttt{.d);}\}? \\ & & & rg\ \{\texttt{assert}\ rg\texttt{.sn.equals(}T\texttt{.d);}\}\ U \\ U & ::= \epsilon & \mid & is\ V \quad (V\texttt{.m} = \texttt{new ArrayDeque(}is\texttt{.result);}) \\ V & ::= \epsilon & \mid & tr \quad \{\texttt{assert}\ V\texttt{.m.peek().equals(}tr\texttt{.item);}\} \\ & & & (V\texttt{.m.pop();})\ V_1\ (V_1\texttt{.m} = V\texttt{.m;}) \end{array}$$

Figure 5.17: Worker Attribute Grammar

```
/*S accepts call to Worker.reg() and, records */
/*the input schedule name, also S allows */
/*arbitary calls to SyncServer.snapshot() */
/*and Util.write() */
S    ::= ε   |   wr S | sp S | rg T    (T.d = et.sn;)

/*T accepts and stores the return */
/*snapshot object from SyncServer.snapshot() */
T    ::= ε   |   sp V (V.d = T.d; U.s = sp.result;)

/*U ensures call items() is called on the same */
/*snapshot object, and the replication items */
/*for the correct schedule are retrieved */
U    ::= ε   |   ls {assert ls.callee.equals(U.s);
                     assert ls.sn.equals(U.d);}
                 V (V.s = U.s;)

/*V records replication items and their name */
/*returned from item() */
V    ::= ε   |   li W (W.is = new HashSet(li.result);
                       W.ns = new HashSet();
                       for (Item i :W.is) {
                         W.ns.add(i.name()); })

/*W ensures all replication */
/*items are processed */
W    ::= ε   |   wr (W.ns.remove(wr.s);)
                 W1 (W1.ns =W.ns; W1.is =W.is;)
             |   is {assert W.is.containsAll(is.result);
                     assert W.ns.isEmpty();}
                 X

X    ::= ε   |   sp X | rg X
```

Figure 5.18: WorkerReg Attribute Grammar

ods. The view WorkerRegHistory, on the other hand, captures the behavior inside reg. According to the informal description above, the view projects incoming method calls and returns of reg, outgoing method calls to server and items, and as well as the outgoing static method calls to write.

We now define the abstract behavior of the communication views, that is, the set of allowable sequences of interactions of objects restricted to those method calls and returns mapped in the views. Each local view also defines the file containing the attribute grammar, whoses terminal symbols the view maps method invocations and returns to. Specifically, Figures 5.15 to 5.18 shows the attribute grammars Snapshot.g, Coordinator.g, Worker.g and WorkerReg.g for views SnapshotHistory, CoordinatorHistory, WorkerHistory and WorkerRegHistory respectively.

The simplest grammar Snapshot.g specifies the interaction protocol of Snapshot. It focuses on invocations of methods refresh and clear per Snapshot object. The grammar essentially specifies the (prefix-closure of the) regular expression (refresh clear)$*$.

The grammar Coordinator.g specifies the interaction protocol of Coordinator. It focuses on invocations of methods start and finish, both of which take a Worker object as the input parameter. These method calls are mapped to terminal symbols *st* and fn, while their inherited attribute is a HashSet, recording the input parameters, thereby enforcing that for each unique Worker object as an input parameter only the set of sequences of method invocations defined by the reqular expression (start finish)$*$ is allowed.

The grammar Worker.g specifies the interaction protocol of Worker It focuses on invocations and returns of methods establish, reg and transfer. The grammar specifies that for each Worker object, establish must be first invoked, then followed by reg and then zero or more transfer, that is, the regular expression (establish reg transfer$*$). We use the attribute definition of the grammar to ensure the following:

- The input argument of establish and reg must be the same;

- reg can only be invoked if the input argument of establish is not “LIST”;
- The return value of reg is a list of Item objects such that transfer is invoked with each of Item in that list from position 0 to the size of that list.

The grammar WorkerReg.g specifies the behavior of the method reg of Worker. It focuses on the invocations and returns of method reg of Worker as well as the outgoing method calls and returns of Util.write and SyncServer.snapshot and Snapshot.items. At the protocol level the grammar specifies the regular expression (snapshot items write∗) inside the invocation method reg. We use attribute definition to ensure the following:

- Snapshot.items must be called with the input argument of reg and it must be called on the Snapshot object that is identical to the return value of SyncServer.snapshot;
- The static method Util.write must be invoked with the value of Item.name for each Item object in the Collection returned from Snapshot.items;
- The returned list of Item objects from reg must be a subset of that returned from Snapshot.items.

Notice that methods Util.write and SyncServer.snapshot may be invoked outside of the method reg. However, this particular behavioral property does not specify the protocol for those invocations. The grammar therefore abstracts from these invocations by allowing any number of calls to Util.write and SyncServer.snapshot before and after reg.

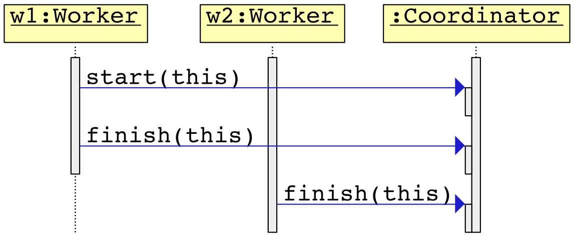


Figure 5.19: Violating histories

5.3 Experiment

We applied SAGA to the Replication System. The current Java implementation of FAS has over 150,000 lines of code, and the Replication System has approximately 6400 lines of code, 44 classes and 5 interfaces.

We have successfully integrated SAGA into the quality assurance process at Fredhopper. The quality assurance process includes automated testing that includes automated unit, integration and system tests as well as manual acceptance tests. In particular system tests are executed twice a day on instances of FAS on a server farm. Two types of system tests are scenario and functional testing. Scenario testing executes a set of programs that emulate a user and interact with the system in predefined sequences of steps (scenarios). At each step they perform a configuration change or a query to FAS, make assertions about the response from the query, etc. Functional testing executes sequences of queries, where each query-response pair is used to decide on the next query and the assertion to make about the response. Both types of tests require a running FAS instance and as a result we may leverage SAGA by augmenting these two automated test facilities with runtime assertion checking using SAGA.

To integrate of SAGA with the system tests, we employ Apache Maven tool[3], an open source Java based tool for managing dependencies between applications and for building dependency artifacts. Maven consists of a project object model

[3] maven.apache.org

```
class WKImpl extends Thread
implements Worker {
 final Coordinator c;
 WKImpl(Coordinator c) {
   this.c = c; }
 public void run() {
  try { .. c.start(this); ..
  } finally {
    c.finish(this); .. }}}
```

Figure 5.20: Incorrect behavior of `WKImpl`

(POM), a set of standards, a project lifecycle, and an extensible dependency management and build system via plug-ins. We use its build system to automatically generate and package the parser/lexer of attribute grammars as well as aspects from views and grammars. We expose the packaged aspects, parser and lexer to FAS instance on the server farm and employ Aspectj using load-time weaver for monitoring method calls/returns during the execution of FAS instances on the server farm. Table 5.1 shows the number of join point matches during the execution of 766 replication sessions over live client data. Figure 5.21 shows the exection time of the 766 replication sessions with and without the integration of SAGA in milliseconds. At some points (for example, around 261 events), the figure seemingly indicates that the system runs faster with SAGA than without. In reality this is not the case: the dependence of the case study on user input (i.e., to start replication sessions) means that it is impossible to replicate an execution exactly (with the only difference being SAGA turned on and off respectively) and leads to small errors in the measurements. However, despite the fact that we cannot control the exact flow of control of the replication sessions (due to this dependence on user input), the graph clearly shows that the integration of SAGA has minimal performance impact on the execution time.

During this session we have found an assertion error at join point `call finish` due to the condition T`.ts.contains(`fn`.t)` not being satisfied at non-terminal T of the grammar Coordinator.g. Specifically, the implementation of Worker (`WKImpl`) that invoke `finish` before `start`. Figure 5.19 shows the sequence diagram of an invalid history

Join point	Terminal	Match
`call static write`	*wr*	247446
`return snapshot`	*sp*	3061
`call transferItem`	*tr*	1101
`return reg` (`WorkerHistory`)	*is*	765
`return reg` (`WorkerRegHistory`)	*is*	765
`call establish`	*et*	766
`call reg` (`WorkerHistory`)	*rg*	765
`call reg` (`WorkerRegHistory`)	*rg*	765
`return items`	*li*	765
`call start`	*st*	766
`call finish`	*fn*	766
`call items`	*ls*	765
`call refresh`	*rf*	766
`call clear`	*cl*	766

Table 5.1: Join point matches in 766 replication sessions

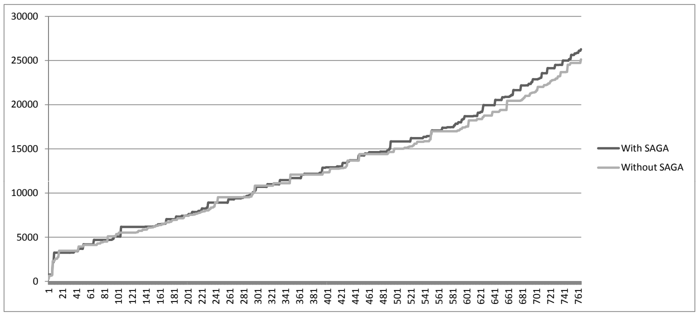


Figure 5.21: Comparison of the execution time (milliseconds) of the replication sessions with and without the integration of SAGA

causing the error, fully automatically generated from the output of SAGA. Figure 5.20 shows part of the implementation of `WKImpl`. It turns out that in the `run` method of `WKImpl`, the method `start` is invoked inside a `try` block while the method `finish` is invoked in the corresponding `finally` block. As a result when there is an exception being thrown by the execution preceding the invocation of `start` inside the `try` block, for example a network disruption, `finish` would be invoked without `start` being invoked.

Concurrent Object Groups 6

In [80] Java is extended with a concurrency model based on the notion of concurrently running object groups, so-called coboxes, which provide a powerful generalization of the concept of active objects. Coboxes can be dynamically created and objects within a cobox have only direct access to the fields of the other objects belonging to the same cobox. Since one of the main requirements of the design of coboxes is a smooth integration with object-oriented languages like Java, coboxes themselves do not have an identity, e.g., all communication between coboxes refer to the objects within coboxes. Communication between coboxes is based on asynchronous method calls with standard objects as targets. An asynchronous method call spawns a local thread within the cobox to which the targeted object belongs. Such a thread consists of the usual stack of internal method calls. Coboxes support multiple local threads which are executed in an interleaved manner. The local threads of a cobox are scheduled cooperatively, along the lines of the Creol modeling language described in [55]. This means, that at most one thread can be active in a cobox at a time, and that the active thread has to give up its control explicitly to allow other threads of the same cobox to become active.

ABS (Abstract Behavioral Specification language) is a novel language based on coboxes for modeling and analysis of complex distributed systems. It is a fully executable language with code generators for Java, Maude and Scala. In [54] a formal semantics of ABS was introduced based on asynchronous messages between coboxes. However, as of yet, no formal method for specifying and run-time verifying traces of such asynchronous messages has been developed. In this chapter, we develop tool support for the efficient run-time verification of asynchronous message passing between coboxes, independent from any backend. This latter requirement is important because in general the analysis of a particular backend is complicated by low-level implementation details. Further, it allows to generalize the analysis to all (including future) backends.

We show how to use attribute grammars extended with assertions to specify and verify (at run-time) properties of the messages sent between coboxes. To this end, we first improve the efficiency of the run-time verification tool SAGA [34], which smoothly integrates both data- and protocol-oriented properties of message sequences. Both time and space complexity of SAGA is linear in the size of the message sequence. Further we extend it to support design-by-contract for coboxes. We illustrate the effectiveness of our method by an industrial case study from the eCommerce software company Fredhopper.

6.1 Language

We formally describe coboxes by means of a modeling language which is based on the *Abstract Behavioral Specification* language [54]. We refer to our own modeling language by ACOG (pure Actor-based Concurrent Object Groups). ACOG is designed with a layered architecture, at the base are functional abstractions around a standard notion of parametric algebraic data types (ADTs). Next we have an OO-imperative layer similar to (but much simpler than) JAVA. ACOG generalizes the concurrency model of Creol [55] from single concurrent objects to concurrent object groups (coboxes). As in [80] coboxes encapsulate synchronous, multi-threaded, shared state computation on a single processor. In contrast to thread-based concurrency, task scheduling is *cooperative*, i.e., switching between tasks of the same object happens only at specific scheduling points during the execution, which are explicit in the source code and can be syntactically identified. This allows writing concurrent programs in a much less error-prone way than in a thread-based model and makes ACOG models suitable for static analysis. In fact, the standard Java concurrency model, based on threads and locks, is too low-level, error-prone and insufficiently modular for many applications areas [80]. In our dialect, unlike in [80], for simplicity we restrict to coboxes that communicate only via pure asynchronous messages, and as such form an actor-based model as initially introduced by [2] and further developed in [82].

Fig. 6.1 shows some data types and parts of interfaces used in the case study. The interface `ClientJob` models a ClientJob, the interface `Worker` models a Worker, and the interface `Coordinator` models a Coordinator. The algebraic data types (ADT) `Content` models the file system of environments in ACOG. ADTs allow specifying immutable values in functional expressions and to abstract away from implementation details such as hardware environment, file content, or operating system specifics. Specifically, `Content` is either a `File`, where an integer (e.g., its size) is taken to represent the content of a single file, or it is a directory `Dir` with a mapping of names to `Content`, thereby, modelling a file system structure with hierarchical name space. Note that an ADT may have

```
data Content = File(Int content)
           | Dir(Map<String,Content>);

interface Worker {
  Unit acceptCoordinator(Coordinator coord);
  Unit sendCurrentId(Int id);
  Unit replyRegisterItems(Bool register);
  Unit acceptItems(Set<Item> items);
  Unit acceptEntries(Set<Map<String,Content>> contents);
}

interface Coordinator {
  Unit startReplication(Worker w);
}
interface ClientJob {
  Unit registerItems(Worker w, Int id);
}
```

Figure 6.1: Data types and Interfaces

type parameters. For example, `Map` is a built-in ADT where its key and value type parameters are instantiated to `String` and `Content`.

In this subsection we describe the core constructs of our dialect of the ABS in some detail. Specifically, we describe

- algebraic data types and functions;
- interfaces
- synchronous method calls and objects creation;
- asynchronous method calls and cobox creation;
- cooperative scheduling using `await` statements.

To illustrate synchronous and asycnhronous communication we look at the implementation of how a ClientJob connects to a Worker and receives the next set of replication schedules.

Data types and Functions ACOG supports *algebraic data types* (ADT) to model data in a software system. ADTs abstract away from implementation details such as hardware environment, file content, or operating system specifics. For example in the Replication System, the following ADT `Content` models the file system of environments.

```
data Content = File(Int content)
             | Dir(Map<String,Content>);
```

ACOG supports first-order functional programming with ADT. Functional code is guaranteed to be free of side effects. One consequence of this is that functional code may not use object-oriented features. For example, the following function `isFile` checks if the given `Content` value records a file.

```
def Bool isFile(Content c) =
    case {
       File(_) => True;
             _ => False;
    };
```

Interfaces ACOG has a nominal type system with interface-based subtyping. Interfaces define types for objects. They have a name, and define a set of method signatures, that is, the names and types of callable methods. The following shows interface `Worker` that models a Worker.

```
interface Worker {
  Unit execute();
  Unit command(Command c);
  Unit acceptCoordinator(Coordinator coord);
  Unit sendCurrentId(Int id);
  Unit replyRegisterItems(Bool register);
  Unit acceptItems(Set<Item> items);
  Unit acceptEntries(Set<Map<String,Content>> contents);
}
```

Classes ACOG also supports class-based, object-oriented programming with standard imperative constructs. Classes define the implementation of objects. In contrast to Java, for example, classes do *not* define a type. Classes can implement arbitrarily many interfaces. These interfaces define the type of instances of that class. A class has to implement all methods of all its implementing interfaces. Classes are instantiated by constructors. The following class `WorkerImpl` implements `Worker`:

```
class WorkerImpl(ClientJob job,
                 SyncServer server,
                 Coordinator coord)
      implements Worker {
  Maybe<Command> cmd = Just(ListSchedule);
  WorkerImpl(ClientJob job,
             SyncServer server,
             Coordinator coord) {
       ...
  }
  Unit execute() {
       ...
  }
  Unit command(Command c) {
       ...
  }
  Unit acceptCoordinator(Coordinator coord) {
       ...
  }
  Unit sendCurrentId(Int id) {
       ...
  }
  Unit replyRegisterItems(Bool register) {
       ...
  }
  Unit acceptItems(Set<Item> items) {
       ...
  }
  Unit acceptEntries(Set<Map<String,Content>> contents) {
       ...
  }
}
```

It defines the fields job, server, cmd and coord. Those fields are typically initialized by a constructor method, or for simple initializations such as cmd, in the class definition itself.

Thread-based computation Basic statements describing the flow of control of a single thread include the usual (synchronous) method invocations, object creation, and field and variable reads and assignments. These statements can be composed by the standard control structures (sequential composition, conditional and iteration constructs). The following shows the part of class ClientJobImpl that a ClientJob connecting to a Worker and acquiring the next schedules:

```
class ClientJobImpl(SyncServer server)
      implements ClientJob {
  Unit sendSchedules(Set<Schedule> ss) {
       ...
  }
  Unit executeJob() {
       ...
  }
  Unit acceptConnection(Worker w) {
       if (w != null) {
              ...
              this.scheduleJob();
       }
  }
  Unit scheduleJobs() {
    Scheduler sr = new SchedulerImpl(...);
    sr.schedule();
  }
}
```

The method acceptConnection invokes synchronously the (private) method scheduleJob, which in turn creates an object of SchedulerImpl (by invoking the appropriate constructor method) and invokes its method schedule.

Coboxes The concurrency model of ACOG is based on the concept of *Coboxes*. A typical ACOG system consists of multiple, concurrently running coboxes. Coboxes can be regarded as autonomous run-time components that are executed concurrently, share no state and communicate via method calls. A new object cobox is created by using the `new cog` expression. It takes as argument a class name and optional parameters and returns a reference to the initial object of the new cobox. Communication between coboxes may solely be done via *asynchronous method calls*. The difference to the synchronous case is that an asynchronous call immediately returns to the caller without waiting for the message to be received and handled by the callee. Asynchronous method calls are indicated by an exclamation mark (!) instead of a dot.

The following fragment of `ClientJobImpl` illustrates cobox creation and asynchronous communications.

```
class ClientJobImpl(SyncServer server,
                  SyncClient client,
                  Schedule s)
     implements ClientJob {

  Set<Schedule> schedules = EmptySet;

  Unit executeJob() {
      server!getConnection(this);
  }
  Unit acceptConnection(Worker w) {
      ...
  }
  Unit sendSchedules(Set<Schedule> ss) {
      ...
  }
  Unit scheduleJobs() {
      ...
  }
}

class SyncServerImpl(Coordinator coord)
     implements SyncServer {

  Unit getConnection(ClientJob job) {
      Bool shutdown = this.isShutdownRequested();
      if (shutdown) {
             job!acceptConnection(null);
      } else {
            Worker w = new cog WorkerImpl(job,
                                       this,
                                       coord);
            job!acceptConnection(w);
      }
  }
}
```

The class SyncServerImpl implements the SyncServer and ClientJobImpl implements a ClientJob. ClientJobImpl has a field server that holds the reference to the SyncServer that is assigned to a different cobox. The method executeJob invokes SyncServer's method getConnection asynchronously to connect with a Worker. In the implementation of SyncServer, a new object cobox is created with the WorkerImpl object being the initial object in that cobox.

Cooperative scheduling Each asynchronous method call results in a *task* in the cobox of the target object. Tasks are scheduled *cooperatively* within the scope of a object cobox. Cooperative scheduling means that switching between tasks of the same object cobox happens only at specific *scheduling points* during program execution and that at no point two tasks in the same cobox are active at the same time. Using the await statement, one can create a *conditional* scheduling point, where the running task is suspended until a Boolean condition over the object state becomes true. The following shows the implementation of ClientJobImpl after connecting with a Worker.

```
class ClientJobImpl(SyncServer server,
                  SyncClient client,
                  Schedule s)
     implements ClientJob {

  Set<Schedule> schedules = EmptySet;

  Unit sendSchedules(Set<Schedule> ss) {
      schedules = ss;
  }
  Unit acceptConnection(Worker w) {
      if (w != null) {
            w!command(Schedule(s));
            await schedules != EmptySet;
            this.scheduleJobs();
      }
  }
  ...
}

class WorkerImpl(ClientJob job,
               SyncServer server)
     implements Worker {
  Unit command(Command c) {
      ...
      job!sendSchedules(schedules);
  }
}
```

The method acceptConnection invokes method command on the worker and suspends using the statement await schedules != EmptySet to wait for the next set of schedules to arrive. The next set of schedules is set by invoking the method sendSchedules on the ClientJob.

Fig. 6.2 shows a part of a class implementation of Worker that provides an implementation of method acceptCoordinator. The method takes a reference of the Coordinator. It first sets the instance variable coord to the input reference, it invokes the statement await cmd != Nothing, which suspends the current task until the side-effect free expression cmd != Nothing is satisfied. Instance variable cmd is a Maybe value which is either a Command value representing the next command to the worker from the ClientJob, or the value Nothing if no command has yet been given. After this, the method makes an asynchronous method call either to the server's requestListSchedules, requesting to get all configured replication schedules, or requestSchedule, requesting only the schedule with the name specified by the given command. Both fromJust and ssname are functions on data types.

```
class WorkerImpl(ClientJob job,
                 SyncServer server)
      implements Worker {

  Maybe<Command> cmd = Nothing;
  Coordinator coord = null;

  Unit acceptCoordinator(Coordinator coord) {
       this.coord = coord;
       await cmd != Nothing;
       if (cmd == Just(ListSchedule)) {
              server!requestListSchedules(this);
       } else {
              server!requestSchedule(this,
                               ssname(fromJust(cmd)));
       }
  }
}
```

Figure 6.2: Method `acceptCoordinator`

6.2 Semantics

In this section we describe the formal semantics of systems of coboxes compositionally in terms of the behavior of the coboxes individually. The behavior of a cobox itself is described compositionally in terms of its threads. In this section we abstract from the functional part of the modeling language. We further abstract from variable declarations and typing information, and simply assume given a set of variables $x, y, \ldots$. We distinguish between simple and instance variables. The set of simple variables is assumed to include the special variable "this". Simple variables are used as formal parameters of method definitions.

Throughout this section we assume a given program which specifies a set of classes and a (single) inheritance relation. We start with the following basic semantic notions. For each class C we assume given a set of O_C, with typical element o, of (abstract) objects which belong to class C at run-time. A *heap* h is formally given as a set of (uniquely) labelled object states $o : s$, where s assigns values to the instance variables of the object o. An object o exists in a heap h if and only it has a state in h, that is, $o : s \in h$, for some object state s. A heap thus represents the values of the instance variables of a group of objects. A heap is "open" in the sense that $s(x)$, for $o : s \in h$, may refer to an object that does not exist in h, i.e., that belongs to a different group. We denote $s(x)$, for $o : s \in h$, by $h(o.x)$. By s_{init} we denote the object state which results from the initialization of the instance variables of a newly created object. Further, by $h[o.x = v]$ we denote the heap update resulting from the assignment of the value v to the instance variable x of the object o. Next we introduce a *thread configuration* as a pair $\langle t, h\rangle$ consisting of a thread t. and heap h. A thread itself is a stack of closures of the form (S, τ), where S is a statement and τ is a local environment which assigns values to simple variables. By $\tau[x = v]$ we denote the update of the local environment τ resulting from the assignment of the value v to the variable x. We denote by $V(e)(\tau, h)$ the value of a side-effect free expression e in the local environment τ and global heap h. In particular we have that $V(x)(s, h) = s(x)$, for a simple variable x, and $V(x)(\tau, h) = h(\tau(\text{this}).x)$, for an instance variable x. It is important to observe that since heaps are "open" (as discussed above) $V(e)(\tau, h)$ can be undefined in case e refers to instance variables of objects that do not belong to the group represented by h.

Thread Semantics A transition

$$\langle t, h\rangle \longrightarrow \langle t', h'\rangle$$

between thread configurations $\langle t, h\rangle$ and $\langle t', h'\rangle$ indicates

- the execution of an assignment $x = e$ or
- the evaluation of a boolean condition b of an if-then-else or while statement,
- or the execution of a synchronous call.

A *labelled* transition

$$\langle t, h\rangle \xrightarrow{l} \langle t', h'\rangle$$

indicates for

$l =$ **await:** the successfull execution of an await statement,

$l = o!m(\bar{v})$**:** an asynchronous call of the method m of the object o with actual parameters $\bar{v}$.

In the following structural operational semantics for the execution of single threads $(S, s) \cdot t$ denotes the result of pushing the closure (S, s) into the stack t. We omit the transitions for sequential composition, if-then-else and while statement since they are standard.

Assignment simple variables

$$\langle (x = e; S, \tau) \cdot t, h \rangle \longrightarrow \langle (S, \tau') \cdot t, h \rangle$$

where $\tau' = \tau[x = V(e)(\tau, h)]$.
The assigment to a simple variable thus only affects the (active) local environment.

Assignment instance variables

$$\langle (x = e; S, \tau) \cdot t, h \rangle \longrightarrow \langle (S, \tau) \cdot t, h'] \rangle$$

where $h' = h[\tau(\text{this}).x = V(e)(\tau, h)]$ (assuming that $V(e)(\tau, h)$ is defined).
The assignment to an instance variable only affects the heap. Note that the assignment thus fails in case $V(e)(\tau, h)$ is undefined (such failures can be prevented by a suitable typing system, see [80]).

Await

$$\langle (\text{ await } b; S, \tau) \cdot t, h \rangle \overset{\text{await}}{\longrightarrow} \langle (S, \tau) \cdot t, h \rangle$$

where $V(b)(\tau, h) = \text{ true }$.
If the boolean condition of an await statement evaluates to true this transition thus additionally generates a label which will be used for synchronization with other threads (see the corresponding rule in the semantics of coboxes below).

Asynchronous method call

$$\langle (x!m(\bar{e}); S, \tau) \cdot t, h\rangle \xrightarrow{o!m(\bar{v})} \langle (S, \tau) \cdot t, h\rangle$$

where $o = V(x)(s, h)$, $\bar{e} = e_1, \ldots, e_n$, $\bar{v} = v_1, \ldots, v_n$, and $v_i = V(e_i)(s, h)$, for $i = 1, \ldots, n$. An asynchronous call thus simply generates a corresponding message.

Synchronous method call

$$\langle (y = x.m(\bar{e}); S, \tau) \cdot t, h\rangle \longrightarrow \langle (S', \tau') \cdot (y = r; S, \tau) \cdot t, h\rangle$$

where, assuming that $V(x)(s, h) \in O_C$, $m(\bar{x})\{S'\}$ is the corresponding method definition in class C. Further, $\tau'(\text{this}) = V(x)(\tau, h)$ and $\tau'(x_i) = V(e_i)(\tau, h)$, for $i = 1, \ldots, n$, (here $\bar{e} = e_1, \ldots, e_n$ and $\bar{x} = x_1, \ldots, x_n$). We implicitly assume here that τ' initializes the local variables of m, i.e., those simple variables which are not among the formal parameters $\bar{x}$. Upon return for each type a distinguished simple variable r (which is assumed not to appear in the given program) will store the return value (see the transition below for returning a value).

Class instantiation

$$\langle (y = \text{ new } C(\bar{e}); S, \tau) \cdot t, h\rangle \longrightarrow \langle (y = r.C(\bar{e}); S, \tau') \cdot t, h \cup \{o' : s_{init}\}\rangle$$

where $\tau' = \tau[r = o']$, $o' \in O_C$ is a fresh object identity (i.e., not in h), where C is the type of the variable y. For each type, the distinguished variable r is used to store temporarily the identity of the new object. We implicitly assume that the constructor method returns the identity of the newly created object (by the statement "return this").

Cobox instantiation

$$\langle (y = \text{ new cog } C(\bar{e}); S, \tau) \cdot t, h\rangle \longrightarrow \langle (y = r; y!C(\bar{e}); S, \tau') \cdot t, h\rangle$$

where $\tau' = \tau[r = o']$, $o' \in O_C$ is a fresh object identity, (C is the type of the variable y). As above, the distinguished variable r is used to store temporarily the identity of the new object (here it allows to circumvent a case distinction on whether y

is a simple or an instance variable). Note that the main difference with class instantiation is that the newly created object is *not* added to the heap h and the constructor method is called asynchronously.

In contrast to [54] and [80] we allow for very flexible scheduling policies (no assumptions are made about scheduling policies at all, even for constructors, besides the fact that `await` statements are respected), it is possible that the constructor method is executed at a later stage than a normal method called on the newly created object. If this is not desired, the user can synchronize explicitly using `await`.

Return

$$\langle(\text{ return } e; S, \tau) \cdot (S', \tau') \cdot t, h\rangle \longrightarrow \langle(S', \tau'[r = v]) \cdot t, h\rangle$$

where $v = V(e)(\tau, h)$. The distinguished variable r here is used to store temporarily the return value.

In the above transitions for the creation of a class instance or a new cobox we assume a thread-local mechanism for the selection of a fresh object identity which avoids name clashes between the activated threads, the technical details of which are straightforward and therefore ommitted.

Semantics of coboxes A cobox is a pair $\langle T, h\rangle$ consisting of a set T of threads and a heap h. An object o belongs to a cobox $\langle T, h\rangle$ if and only if it has a state in h, that is, $o : s \in h$, for some object state s.

Internal computation step

An unlabelled computation step of a thread is extended to a corresponding transition of the cobox by the following rule:

$$\frac{\langle t, h\rangle \longrightarrow \langle t', h'\rangle}{\langle \{t\} \cup T, h\rangle \longrightarrow \langle \{t'\} \cup T, h'\rangle}$$

External call

A computation step labelled by an asynchronous method call is extended to a corresponding transition of the cobox by the following rule:

$$\frac{\langle t, h\rangle \overset{o!m(\bar{v})}{\longrightarrow} \langle t', h'\rangle}{\langle \{t\} \cup T, h\rangle \overset{o!m(\bar{v})}{\longrightarrow} \langle \{t'\} \cup T, h'\rangle}$$

Synchronization

The execution of an await statement by a thread within a given cobox is formally captured by the rule

$$\frac{\langle t, h\rangle \overset{\text{await}}{\longrightarrow} \langle t', h'\rangle}{\langle \{t\} \cup T, h\rangle \longrightarrow \langle \{t'\} \cup T, h'\rangle}$$

provided all threads in T executing an await statement, that is, the top of each thread in T consists of a closure of the form (await $b; S, \tau$) (we implicitly assume that terminated threads are removed). Note that thus the await statement enforces a barrier synchronization of all the threads of a cobox. This synchronization ensures that at most one thread in a cobox is executing.

Input-enabledness

We further have the following transition which describes the *reception* of an asynchronous method call to an object o which belongs to the cobox $\langle T, h\rangle$:

$$\langle T, h\rangle \overset{o?m(\bar{v})}{\longrightarrow} \langle T \cup \{t\}, h\rangle$$

where, assuming that $o \in O_C$, $m(\bar{x})\{S\}$ is the corresponding method definition in class C. Further, t consists of the closure $\langle$ await b ; $S, \tau\rangle$, where $V(b)(\tau, h) =$ true and τ assigns the actual parameters $\bar{v}$ to the formal parameters $\bar{x}$ of m (as above, the object identity o is assigned to the implicit formal parameter "this") and initializes all local variables of m.

The added await statement enforces synchronization between the other threads. Since coboxes are *input-enabled* this transition thus models *an assumption* about the environment. This assumption is validated in the context of coboxes as described next.

Semantics of systems of coboxes Finally, a system configuration is simply a set G of coboxes. For technical convenience we assume that all system configurations contain an infinite set of *latent* coboxes $\langle \emptyset, \{o : s_{\ init}\ \}\rangle$ (for $o \in O_C$ for all classes C) which have not yet been activated. The fresh object generated by the creation of a new cobox, as described above in the thread semantics (transition 6.2), at this level is assumed to correspond to a latent cobox.

Interleaving

An internal computation step of a cobox is extended to a corresponding transition of the global system as follows.

$$\frac{g \longrightarrow g'}{\{g\} \cup G \longrightarrow \{g'\} \cup G}$$

Message passing

Communication between two coboxes is formalized by

$$\frac{g_1 \xrightarrow{o?m(\bar{v})} g_1' \quad g_2 \xrightarrow{o!m(\bar{v})} g_2'}{\{g_1, g_2\} \cup G \longrightarrow \{g_1', g_2'\} \cup G}$$

Here it is worthwhile to observe that for an asynchronous call $o!m(\bar{v})$ to an object o belonging to the *same* cobox there does not exist a matching reception $o?m(\bar{v})$ by a *different* cobox because coboxes have no shared objects.

Trace Semantics A trace is a finite sequence of input and output messages, e.g., $o?m(\bar{v})$ and $o!m(\bar{v})$, respectively. For each coboxes g we define its trace semantics $T(g)$ by

$$\{\langle \theta, g' \rangle \mid \ g \xrightarrow{\theta} g'\}$$

where $\xrightarrow{\theta}$ denotes the reflexive, transitive closure of the above transition relation between coboxes, collecting the input/output messages. Note that the trace θ by which we can obtain from g a cobox g' does *not* provide information about object creation or information about which objects belong to the same cobox. In fact, information about which objects have been created can be inferred from the trace θ. Further, in general a cobox does not "know" which objects belong to the same cobox.

The following compositionality theorem is based on a notion of *compatible* traces which roughly requires for every input message a corresponding output message, and vice versa. We define this notion formally in terms of the following rewrite rule for sets of traces

$$\{o?m(\bar{v}) \cdot \theta, o!m(\bar{v}) \cdot \theta'\} \cup \Theta \Rightarrow \{\theta, \theta'\} \cup \Theta$$

This rule identifies two traces in the given set which have two matching initial messages which are removed from these traces in the resulting set. Note that this identification is nondeterministic, i.e., for a given trace there may be several traces with a matching initial message. A set of traces Θ is compatible, denoted by *Compat* (Θ), if we can derive the singleton set $\{\epsilon\}$ (ϵ denotes the empty trace). Formally, *Compat* (Θ) if and only if $\Theta \Rightarrow^* \{\epsilon\}$, where $\Rightarrow^*$ denotes the reflexive, transitive closure of $\Rightarrow$.

Theorem 6.2.1 *Let $\rightarrow^*$ denote the reflexive, transitive closure of the above transition relation between system configurations. We have*

$$G \longrightarrow^* G'$$

if and only if $G = \{g_i \mid i \in I\}$ and $G' = \{g'_i \mid i \in I\}$, for some index set I such that for every $i \in I$ there exists $\langle \theta_i, g'_i \rangle \in T(g_i)$, with Compat $(\{\theta_i \mid i \in I\})$.

Proof: The proof is straightforward but tedious and proceeds by induction on the derivation.

The above theorem states that the overall system behavior can be described in terms of the above trace semantics of the individual coboxes. This means that for compositionality no further information is required. Next we show in the following section how to specify properties of the externally observable behavior of a cobox, as defined by its traces of input/output messages.

6.3 Behavioral Interfaces for Coboxes

In this section we use the previously introduced *attribute grammars* extended with assertions to specify and verify properties of the traces generated between coboxes. As such, extended attribute grammars provide a new formalism for contracts in general, and coboxes in particular. In contrast to classes or interfaces, coboxes are run-time entities which do not have a single fixed interface[1]. In particular, newly created objects of any type can be added dynamically at any time to a group by executing a `new`-statement without the `cog` keyword. The execution of such a statement expands the group with the new object, which can be of a type different from all the objects in the group so far and consequently provides methods not provided by the other objects in the group.

As a crude approximation of the behavior of a group, we could just take the behavior of the class `C` of the object which is created by a `new cog C` statement. Such an approach requires no modification in specification languages (and corresponding tools) traditionally used for object-oriented languages (be it assertions on states, or trace-based specifications). However this basically corresponds to the assumption that all groups contain only a single object, bypassing the very concept of *groups* of objects and essentially resulting in concurrent objects, not concurrent object *groups*. This can be partly alleviated by assuming that besides the object of type `C`, objects of other types can also be part of the group by having specifications of the form 'if an object of type `D` is part of the group, then ϕ'. However, this means that *all* groups whose first created object is of type `C` must satisfy the property (since the group name in this case does not depend on other objects in the group). We abandon this idea in favor of a more fine-grained specification of groups. We first discuss how we can still refer statically, in the program text, to these run-time entities by communication views.

[1] We consider interfaces here to be a list of all signatures of the methods supported by some object in the cobox

Communication Views for COGs

To be able to refer to coboxes in syntactical constructs (such as specifications), we introduce the following (optional) annotation of cobox instantiations:

$$S \quad ::= \quad y = \text{new cog [Name]} \;\; C(\bar{e})$$

The semantics of the language remain unchanged. Note that the same cobox name can be shared among several coboxes (i.e. is in general not unique) since different cobox creation statements can specify the same cobox name. First, the same group creation statement `y = new cog A C` can appear multiple times in the program. Second, a group creation statement can be surrounded by a loop or recursive method, causing it to be executed multiple times. In both cases, all groups created by the statement receive the same name. However in contrast to the previously sketched abandoned proposal (where different named groups are distinguished at the class level of the first object in the group), in this approach, named groups are distinguished at the finer-grained statement level.

Now that we can refer to groups syntactically, the question arises what kind of specification languages would be suitable to specify the behavior of a named group. Groups communicate with other groups exclusively using asynchronous method calls, and there are no returns. Thus, the behavior of a group as observed by its environment (other groups) is simply a sequence of asynchronous method calls sent and received by objects in the group. Taking the set of all legal sequences of asynchronous method calls (also known as traces) of the group as its specification is a natural choice. We observe asynchronous method calls directly when the method call statement executes in the group, not when the actual method body of the called method begins to execute (note that the latter can happen at a much later time in our concurrency model, or even not at all in the absence of fairness assumptions). This convention allows an orthogonal treatment of scheduling policies. However, we have to face the following problem: Coboxes do not have a fixed interface, as the methods which can be invoked on an object in a cobox (and consequently appear in traces) are not fixed statically. In particular, during execution objects of any type can be added to a cobox, which clearly affects the possible traces of the cobox. Additionally, for practical reasons it is often convenient to focus on a particular subset of methods, leaving out methods irrelevant for specification purposes. This is especially useful for incomplete specifications. To solve both these problems, we use *communication views*. A communication view can be thought of as an interface for a named cobox. Figure 6.3 shows an example communication view associated with all coboxes named WorkerGroup. Formally a communication view is a partial mapping from messages to abstract event names. A communication view thus simply introduces names tailored for specification purposes (see the next subsection about grammars for more details on how this event name is used). Partiality allows the user to select only those asynchronous methods relevant for specification purposes. Names (such as 'st' for the method `startReplication`) are not strictly needed, but can be used to identify calls to different methods. Any method not listed in the view will be irrelevant in the specification of WorkerGroups.

```
view WorkerView grammar Worker.g specifies WorkerGroup {
  send Coordinator.startReplication(Worker w) st,
  send ClientJob.registerItems(Worker w, Int id) pr,
  receive Worker.sendCurrendId(Int id) id,
  receive Worker.replyRegisterItems(Bool reg) ar,
  receive Worker.acceptItems(Set<Item> items) is,
  receive Worker.acceptEntries(
           Set<Map<String, Content>> contents) es
}
```

Figure 6.3: Communication View

Note that in this asynchronous setting we can distinguish three different events: sending a message (at the call-site), receiving the message in the queue (at the callee-site), and scheduling the message for execution (i.e. the point in time when the corresponding method starts executing). By the asynchronous nature of the ABS, we cannot detect in the ABS itself when a message has been put into the queue. Therefore we restrict to the other two events. Since we implement the run-time checker independently from any back-end (see also Section 6.4), we are forced to use the ABS itself for the detection of the observable events. The `send` keyword identifies calls from objects in the WorkerGroup to methods of objects in another cobox (in other words: methods required by an object in the WorkerGroup). Conversely, the keyword `receive` identifies the scheduling of calls from another cobox to an object in a WorkerGroup. It is possible that methods listed in the view actually can never be called in practice (and therefore won't appear in the local trace of a cobox). For example, if WorkerGroups are created by a statement `y = new cog WorkerGroup Worker2`, only objects of the class `Worker2` are guaranteed to be part of the group. Thus messages of the form `receive Worker.*` can only be received in those WorkerGroups in which a `Worker`-object was added. The introduction of names for messages gives rise to a small refinement of our notion of a specification of a group. A specification is not a set of sequences of asynchronous method calls anymore, instead a specification is a set of sequences of names.

Communication views allow the selection of messages essentially just on the basis of the method name. But messages also involve and contain data: they are sent between an object in one group (the caller), to an object in another group (the callee) with the actual parameter values as the contents of the message. Thus the question arises what data (caller, callee, actual parameters) we can observe and use in specifications of groups. Clearly the parameter values sent in a message influence the behavior of the group which receives the message. On the other hand, as can be seen by inspecting the formal (compositional) semantics introduced in the previous section, the identity of the caller of a `receive` message does not influence the behavior of a group. In particular, there is no way to detect whether two messages originate from the same group (or even the same object). Thus it would be unnatural if one could refer in a specification to the identity of the caller: this results in specifications that cannot be satisfied by any implementation. Consequently we disallow any reference to the identity of the caller in specifications, and take the callee and the actual parameter values as the only data that can be observed from a message. A fully abstract semantics allows to determine the *minimum* amount of information that needs to be captured. The introduction of data introduces another refinement of our notion of a specification. Message names are not just strings anymore, they also contain the identity of the callee and the actual parameter values. A specification for a group is still a set of the legal sequences of names (as above), but since names now also contain the callee and parameter values, their values can be restricted by the specification. Note that specifications combine protocol-oriented properties (such as the legal orderings between messages) and data-oriented properties (such as the allowed parameter values). The next subsection discusses how specifications can be defined syntactically in a convenient way.

We are now in the position to formally define when an implementation satisfies a specification.

Definition Let P be a program and S a specification (set of traces). Then $P \models S$ iff For all sets of groups $G = \{g_i \mid \ i \in I\}$: For all $i \in I : \theta_i \in T(g_i)$ and compat$(\{\theta_i | i \in I\})$ implies $\text{Proj}_V(\theta_i) \in S$.

In this definition we assume a mechanism $\text{Proj}_V(\theta_i)$for projecting each trace of a named group on the events listed in the associated communication view V. Informally the definition says that a P satisfies S if the traces of P are a subset of those of S.

Attribute Grammars

In this subsection we describe how properties of the set of allowed traces of a cobox can be specified in a convenient, high-level and declarative manner. We illustrate our approach by partially specifying the behavior depicted by the UML sequence diagram in Figure 6.4. Informally the property we focus on is:

> The Worker first notifies the Coordinator its intention to commence a replication session, the Worker would then receive the last transaction id identifying the version of the data to be replicated, the Worker sends this id to the ClientJob to see if the

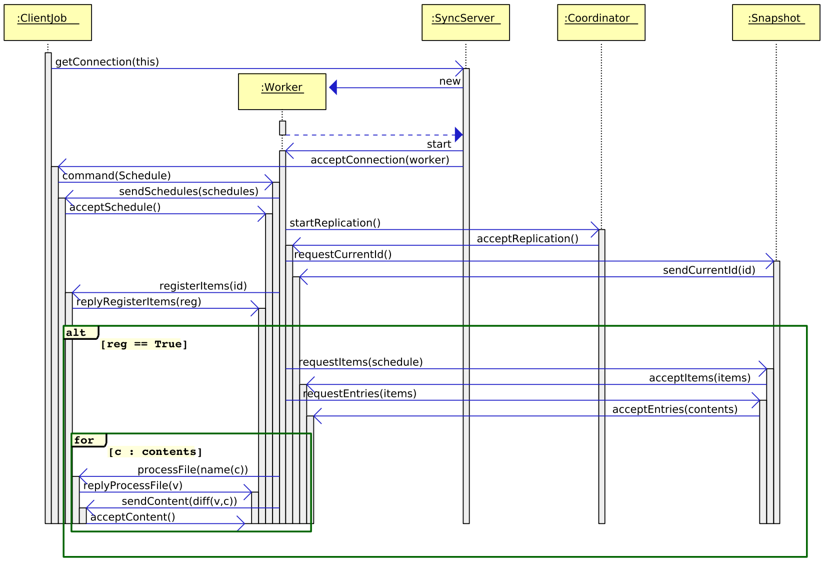


Figure 6.4: Replication interaction

client is required to update its data up to the specified version. The Worker then expects an answer. Only if the answer is positive can the Worker retrieve replication items from the snapshot, moreover, the number of files sets to be replicated to the ClientJob must correspond to the number of replication items retrieved.

The formalization of the above property uses the communication view depicted in Figure 6.3. The productions of the grammar underlying the attribute grammar in Figure 6.5 specify the legal orderings of these messages named in the view. For example, the productions

$$\boxed{\begin{array}{lll} S & ::= & \epsilon \\ & | & st \;\; T \\ T & ::= & \epsilon \\ & | & id \;\; U \end{array}}$$

specify that the message ‘id’ is preceded by the message ‘st’.

The grammars above specify only the *protocol structure* of the valid traces, but do not take the data-flow into account. To that end, we extend the grammar with attributes and assertions over these attributes. Each terminal symbol has *built-in* attributes consisting of the parameter names for referring to the object identities of the actual parameters, and `callee` for referencing the identity of the callee (see the end of the previous subsection for a motivation to include these particular attributes). Non-terminals have *user-defined* attributes to define data properties of sequences of terminals. In each production, the value of the attributes of the non-terminals appearing on the right-hand side of the production is defined.[2] For example, in the following production, the attribute ‘w’ for the non-terminal ‘T’ is defined.

$$\boxed{\begin{array}{lll} S & ::= & \epsilon \\ & | & st\ \ T \quad (T.\mathtt{w} = st.\mathtt{w;}) \end{array}}$$

Attribute definitions are surrounded by ‘(’ and ‘)’. However the attributes themselves do not alter the language generated by the attribute grammar, they only *define* properties of data-flow of the trace. We extend the attribute grammar with assertions to specify properties of attributes. For example, the assertion in the second production of

$$\boxed{\begin{array}{llll} T & ::= & \epsilon & \\ & | & id\ \ U & (U.\mathtt{w} = T.\mathtt{w;}\ U.\mathtt{i} = id.\mathtt{id;}) \\ U & ::= & \epsilon & \\ & | & pr & \{\mathtt{assert}\ U.\mathtt{w} == pr.\mathtt{w} \\ & & & \qquad \mathtt{\&\&}\ U.\mathtt{i} == pr.\mathtt{id;}\}\ V \end{array}}$$

expresses that the ‘id’ passed as a parameter to the method ‘registerItems’ (represented in the grammar by the terminal pr.`id`) must be the same as the one previously passed into ‘sendCurrentId’ (terminal id.`id`). Assertions are surrounded by ‘{’ and ‘}’ to distinguish them visually from attribute definitions.

[2] In the literature, such attributes are called inherited attributes.

$$
\begin{array}{llll}
S & ::= \epsilon & \mid st & T\ (T\texttt{.w} = st\texttt{.w;}) \\
T & ::= \epsilon & \mid id & U\ (U\texttt{.w} = T\texttt{.w;}\ U\texttt{.i} = id\texttt{.id;}) \\
U & ::= \epsilon & \mid pr & \{\texttt{assert}\ U\texttt{.w} == pr\texttt{.w} \\
 & & & \texttt{\&\&}\ U\texttt{.i} == pr\texttt{.id;}\}\ V \\
V & ::= \epsilon & \mid ar & W\ (W\texttt{.b} = ar\texttt{.reg;}) \\
W & ::= \epsilon & \mid is & \{\texttt{assert}\ W\texttt{.b;}\}\ X\ (X\texttt{.s} = \texttt{size(}is\texttt{.items);}) \\
X & ::= \epsilon & \mid es & \{\texttt{assert}\ X\texttt{.s} == \texttt{size(}es\texttt{.contents);}\}
\end{array}
$$

Figure 6.5: Attribute Grammars

The full attribute grammar Figure 6.5 formalizes the informal property stated in the beginning of this subsection. The grammar specifies that for each Worker object, in its own object cobox, the Coordinator must be notified of the start of the replication by invoking its method `startReplication` (*st*). Only then can the Worker receive (from an unspecified cobox) the identifier of the current version of the data to be replicated (*id*). Next the Worker invokes the method `registerItems` on the corresponding ClientJob about this version of the data (*pr*). The grammar here asserts that the identifier is indeed the same as that received via the method call `sendCurrendId`. The Worker then expects to receive a method call `replyRegisterItems` indicating if the replication should proceed, the Worker then can recieve method call `acceptItems` for the data items to be replicated. The grammar here asserts that this can only happen if the previous call indicated the replication should proceed. The Worker then can receive method call `acceptEntries` for the set of Directories, each identified by a data item. Since each data item refers to a directory, the grammar here asserts the number of items is the same as the number of directories.

To further illustrate the above concepts, we consider an additional behavioral interface for the WorkerGroup cobox. To allow users to make changes to the replication schedules during the run-time of FAS, every ClientJob would request the next set of replication schedules and send them to SyncClient for scheduling. Here is an informal description of the property, where Figure 6.6 presents the communication view capturing

```
view ScheduleView grammar Schedule.g
specifies WorkerGroup {
  receive Worker.command(Command c) cm,
  send ClientJob.sendSchedules(Set<Schedule> ss) sn,
  send SyncServer.requestListSchedules(Worker w) lt,
  send SyncServer.requestSchedule(Worker w, String name)
       gt,
  send Coordinator.requestStartReplication(Worker w) st
}
```

Figure 6.6: Communication View for Scheduling

$$\begin{array}{llll}
S & ::= \epsilon & \mid cm & T \; (T\texttt{.c} = cm\texttt{.c;}) \\
T & ::= \epsilon & \mid gt & \{\texttt{assert } T\texttt{.c} \mathrel{!=} \texttt{ListSchedule \&\&} \\
 & & & \quad gt\texttt{.n} == \texttt{name(}T\texttt{.c);}\} \; U \; (U\texttt{.c} = T\texttt{.c;}) \\
 & & \mid lt & \{\texttt{assert } T\texttt{.c} == \texttt{ListSchedule;}\} \; U \; (U\texttt{.c} = T\texttt{.c;}) \\
U & ::= \epsilon & \mid sn & \{\texttt{assert } sn\texttt{.ss} \mathrel{!=} \texttt{EmptySet;}\} \; V \; (V\texttt{.c} = U\texttt{.c;}) \\
V & ::= \epsilon & \mid st & \{\texttt{assert } V\texttt{.c} \mathrel{!=} \texttt{ListSchedule;}\}
\end{array}$$

Figure 6.7: Attribute Grammar for Scheduling

the relevant messages and Figure 6.7 presents the grammar that formalizes the property:

> A ClientJob may request for either all replication schedules or a single schedule. The ClientJob does this by sending a command to the Worker (*cm*). If the command is of the value `ListSchedule`, the Worker is to acquire all schedules from the SyncServer (*lt*) and return them to the ClientJob (*sn*). Otherwise, the Worker is to acquire only the specified schedule (*gt*) and return it to the ClientJob (*sn*). If the ClientJob asks for all schedules, it must not proceed further with the replication session and terminate (*st*).

In summary, a communication view provides an interface of a named cobox. The behavior of such an interface is specified by means of an attribute grammar extended with assertions. This grammar represents the legal traces of the named cobox as words of the language generated by the grammar, which gives rise to a natural notion of the satisfaction relation between programs and specifications. Properties of the control-flow and data-flow are integrated in a single formalism: the grammar productions specify the valid orderings of the messages (the control-flow of the valid traces), whereas assertions specify the data-flow.

6.4 Implementation

In this section we discuss the architecture of the run-time checker for coboxes, identify crucial design decisions and its performance. The cobox version of SAGA is implemented as a run-time checker for ABS models. ABS is basically an extension of the modeling language considered in this paper. It is tool-supported by various analysis tools [88] and automated code generation has been implemented to various lower-level languages including Java, Maude and Scala. SAGA tests whether an actual execution of a given ABS model satisfies its specification given by attribute grammars, and stops the running program in case of a violation to prevent unsafe behavior. It is implemented as a meta-program in Rascal. The full meta-program consists of approximately 1100 lines of code.

Design The design of the cobox version of SAGA was guided by several requirements.

1. *All* back-ends (even future ones) which generate code from ABS models to lower-level target languages should be supported, without having to update SAGA when any of the back-ends is updated (for example, to generate more efficient code). Consequently we need a parser-generator which generates ABS code, and therefore cannot use existing parser generators.

2. The overhead induced by SAGA must be kept to a minimum. In particular, whenever the trace of a cobox is updated with a new message, SAGA should be able to decide *in constant time* whether the new trace still satisfies the specification (the attribute grammar). This is determined by parsing the trace (then considered as a sequence of tokens) in a parser for the attribute grammar.

3. Because of the intrinsic complexity of developing efficient and user-friendly parser generators, we require that the implementation of the parser-generator should be decoupled from the rest of the implementation of SAGA.

These requirements are far from trivial to satisfy. For example JML, a state-of-the-art specification language for Java, has no stable version of the run-time checker which supports all back-ends (and future ones) for Java, violating the first requirement. This is due to the fact that the JML run-time checker was designed as an extension of a proprietary Java compiler. Other tools for run-time verification such as MOP and LARVA satisfy the requirement to a certain extent. Their implementation is based on AspectJ, a compiler which extends Java with aspect-oriented programming. AspectJ can transform Java programs in bytecode form. Hence all back-ends which generate bytecode compatible with AspectJ are also supported by MOP and LARVA. This includes most, though not all, versions of the standard Sun Java compiler. However aspect-oriented programming is currently not supported by the ABS. We choose an approach based on pre-processing. Specifications (consisting of a communication view and attribute grammar) are not added to the formal syntax of the programming language, they are put in separate files. This avoids creating multiple branches of the ABS language. In JML, specifications are added to the actual source, but in comments (so they are not part of the ”logic” of the program). In MOP and LARVA, specifications are also separated from the programming language.

The input of SAGA consists of three ingredients: a communication view, an attribute grammar extended with assertions and an ABS model. The output is an ordinary ABS model which behaves the same as the input program, except that it throws an assertion failure when the current execution violates the specification. Since the resulting ABS model is an ordinary ABS model, all analysis tools[88] (including a debugging environment with visualization and a state-of-the-art cost analyzer) and back-ends which exist for the ABS can be used on it directly. The third requirement (a separation of concerns between the parser-generator and the rest of the implementation) has lead to a component-based design consisting of a parser-generator component and source-code weaving component. We discuss these components, and the second requirement on performance of the generated parser, in more detail below.

Parser generator component The parser-generator component processes only the attribute grammar and generates a parser for it, with ABS as the target language. Parsers for attribute grammars in general take a stream of terminals as input, and output a parse tree according to the grammar productions (where non-terminal nodes are annotated with their attribute values). In our case, the attribute grammars also contains assertions, and the generated parser additionally checks that all assertions in the grammar are true.

In our case, whenever a new message (asynchronous call) is added to the trace, all parse trees of all prefixes have been computed previously. The question arises how efficient the new parse trees can be computed by exploiting the parse trees of the prefixes. Unfortunately, for general context-free grammars, this cannot be done in constant time using currently known algorithms (violating the second requirement on performance). For if this was possible in constant time, parsing the full trace results in a parser which works in linear time (n terminals which all take a constant amount of time), and no linear time algorithm for general context-free grammars is known. We therefore restrict to deterministic regular attribute grammars with only inherited attributes. All grammars used in the case study have this form and parsing the new trace in such grammars can be done in constant time, since they can be translated to a finite automaton with conditions (assertions) and attribute updates as actions to execute on transitions. Parsing the new message consists of taking a single step in this automaton. Moreover for such grammars, the space complexity is also very low: it is not necessary to store the entire trace, only the attribute values of the previous trace must be stored.

Source-code weaving component The weaving component processes the communication view and the given ABS model, and outputs a new ABS model in which each call to a method appearing in the view is transformed. The transformation checks whether the method call which is about to be executed is allowed by the attribute grammar, and if this is not the case, prevents unsafe behavior by throwing an assertion failure. This transformation is invasive, in the sense that it cannot be done only locally in the body of those methods actually appearing in the view, but instead it has to be done at all call-sites (in client code). To see this, suppose that the transformation *was* done locally, say in the beginning of the method body. Due to concurrency and scheduling policies, other methods which were called at a later time could have been scheduled earlier. In such a scenario, these other methods are checked earlier than the order in which they are actually called by a client, which violates the decision (see also the previous section) to treat scheduling policies orthogonally.

The transformation is done in two steps. First, all calls to methods that occur in a communication view are isolated using pattern matching in the meta-program. We created a Rascal ABS grammar for that purpose. The ABS grammar contains around 240 non-terminals: for comparison, the Java grammar in Rascal has about 120. The main reason for the significantly larger size is that the ABS contains an internal sublanguage (for feature models and delta programming [79, 23]) for designing software product lines. The following snippet from the Rascal ABS grammar descibes the syntax for asynchronous method calls (i.e. `o`m(e1, ..., en)!).

```
syntax AsyncCall
       = PureExpPrefix ! IDENTIFIER ( DataExp ","* );
```

In the second step, all asynchronous call-statements are preceded by code which checks that the current object is part of a named cobox (note that this check really has to be done at run-time due to the dynamic nature of coboxes). If this is the case, the trace is updated by taking a step in the finite automaton where additionally the assertion is checked. If there is no transition for the message from the current state, we throw an assertion error. Intuitively such an error corresponds to a protocol violation. There is one subtle point about updating the trace. If no assumptions are made about the scheduling of received messages, only updates to the trace of the calling cobox (i.e. 'send' messages in the view) can be guaranteed to be executed directly before the actual call happens. For 'receive' messages the history is updated whenever the corresponding method begins executing. Thus this assymmetry between 'send' and 'receive' events is natural when one takes into account that the actual behavior of the program only depends on the order in which the 'receive' events (or rather, the associated methods) are actually executed.

Related- and Future Work 7

In this chapter we present a direct comparison of our own framework and corresponding tool support with PQL [64], Jassda [16], LARVA [26] and MOP [20]. We model the properties of the Fredhopper case study described in the previous section in these respective tools. We consider the **learnability** of the frameworks: that is, does the framework provide a specification language with a surface syntax close to existing formal/modeling languages? Is the semantics of the specifications properly documented? We test how easily the frameworks can be **adopted** or integrated into the the software development cycle in an industrial context such as at Fredhopper. This includes operational steps like installation, execution, and documentation and support.

We now provide a brief overview of these three frameworks.

PQL Program Query Language, PQL, is developed by Martin et al. [64], it is a query language for pattern matching (possibly recursive) sequences of method invocations of Java programs. Unlike all the other approaches in the evaluation, PQL queries express invalid behavior rather than all possible valid behavior. Figure 7.1 shows a PQL query expressing part of the `Worker` property. Specifically, it matches invalid behavior of invoking `reg` after invoking `establish` with the input argument "LIST".

Jassda Java with assertions Debugger Architecture, Jassda, is a framework developed by Brörkens et al. [16]. Trace assertions are given in a CSP-like notation. The CSP-like notation maps invocations and returns of method calls of Java programs as events. For example, Figure 7.2 shows that the event $w.rg.begin$ is mapped to the invocation of method `Worker.reg` The framework takes such assertions, combined with the Java Debugger Architecture to monitor calls and returns of methods. It then generates a program for a state machine, and attaches to the Java Virtual Machine running the system under test that is accepting debug connections. The framework then monitors invocations and returns of method calls and output log messages to a separate file on state transitions.

LARVA Logical Automata for Runtime Verification and Analysis, LARVA, is developed by Colombo et al. [26]. LARVA provides a modeling language for specifying both valid and invalid method invocations of Java programs by enumerating transitions in a state transition function of an abstract machine. Each transition is conditioned on an event, where an event maps to one or more invocations and returns of method calls. The modeling language also permits declaring global variables of any visible Java types (class/interface), and at each transition an optional condition can be made about values of these variables against input arguments or return values of methods as well as any number of Java statements on these variables and values. Figure 7.3 shows how to partially model `Worker` property in LARVA.

JavaMOP Monitoring-Oriented Programming, MOP [20], is a run-time monitoring tool based on aspect-oriented programming which uses context-free grammars to describe properties of the control flow of histories. Properties on the data-flow are *predefined* built-in functions (basically AspectJ functions such as a 'target' to bind the callee and 'this' to bind the caller, comparable to built-in attributes of terminals in our setting). This limits the expression of data properties: there is no support for defining properties of *sequences* of terminals. To circumvent this limitation one may however hack general properties into the tool implementation. Figure 7.4 formalizes the protocol behavior of the Worker.

In contrast, our approach supports a general methodology to introduce systematically *user-defined* properties, by means of attributes of non-terminals. Furthermore SAGA supports conditional productions which are essential to specify protocols dependent on data in a declarative manner. Finally, JavaMOP does not directly support the specification of local histories (i.e. monitoring the messages sent and received by a single object).

Expressiveness

	Snapshot	Coordinator	Worker
PQL	yes	no	no
Jassda	yes	no	no
LARVA	yes	yes	yes
MOP	yes	yes	yes
SAGA	yes	yes	yes

Table 7.1: Comparison of Expressiveness

We investigated the expressiveness of the specification languages of these tools by attempting to express and check the SnapShot, Worker and Coordination properties (see Chapter 5). The resulting specifications are given in Figures 7.1 to 7.4.

Table 7.1 summarizes the results. Neither PQL nor Jassda can express the Coordinator and Worker properties since neither allows user-defined properties of data. In both Coordinator and Worker properties, the validity of a method invocation is dependent on the value of the input arguments as well as the return values. For example in PQL snippet in Figure 7.1, to model the invalid sequence of method invocations establish("LIST") followed by reg(_), we had to encapsulate the string value "LIST" into the method SyncServer.getList() as PQL does support object manipu-

	Specification	**Execution**
PQL	5	2
Jassda	4	2
LARVA	2	1
MOP	5	1
SAGA	3	1

Table 7.2: Duration per Activity in hours

```
query main ()
uses object Worker w; object String s;
matches {
  w = Acceptor.getWorker();
  s = SyncServer.getList();
  w.establish(s); w.reg(s); }
executes Util.printStackTrace(*);
```

Figure 7.1: PQL

```
trace worker {
 eventset w { class="Worker" }
 eventset r { method="reg"}
 ...
 process main() {
  w.ls.begin -> w.ls.end ->
  w.et.begin -> w.et.end -> STOP
  []
  w.os.begin -> w.os.end ->
  w.et.begin -> w.et.end ->
  w.rg.begin -> STOP }}
```

Figure 7.2: JASS

lation. LARVA and MOP, on the other hand, support executing arbitrary Java statements when an event occurs, hence it is possible to define data-oriented properties such as `Coordinator` and `Worker`. As such, user-defined properties of the data of a single event are possible to express. It is not possible to di-

```
IMPORTS{ import java.util.*; }
GLOBAL {
 FOREACH (Worker w) {
  VARIABLES { String c = null; ArrayDeque q = null;}
  EVENTS{
   et(String s, Worker w1) = {
     w1.establish(s);} where {w = w1;}
   is(String s, List is, Worker w1) = {
     w1.reg(s)uponReturning(is);} where {w = w1;}
   tr(Item i, Worker w1) = {
     w1.transfer(i);} where {w = w1;}}
  PROPERTY workers{
   STATES {
    STARTING{ start{} }
    BAD{ regL{} transW{} }
    NORMAL{ est{} regS{} transC{} }}
   TRANSITIONS{
    start -> est [et()\\c = s;]
    est -> regL [is()\"LIST".equals(c)]
    est -> regS [is()\! "LIST".equals(c)\q =
                        new ArrayDeque(is);]
    regS -> transW [tr()\q.pop() != i]
    regS -> transC [tr()\q.pop() == i] }}}}
```

Figure 7.3: LARVA

rectly express properties of *sequences* of events (i.e. the dataflow of the history). In LARVA, non-regular context-free protocols cannot be expressed *directly*: one would have to write the parser for a context-free grammar oneself. The user would then essentially be writing their own run-time checker in Java, bypassing MOP and Larva. This is clearly unfeasible, and the resulting specifications are not declarative anymore. Most importantly, in that degenerative sense of expressiveness, AspectJ (on which MOP and LARVA are based) would already be sufficient.

Moreover, since LARVA supports the manipulation of arbitrary Java objects as global variables, it is as expressive as SAGA. However, unlike SAGA, LARVA requires the specifi-

```
import java.io.*; import java.util.*;
suffix HasNext(Worker w) {
   event et before(Worker w):
      call(* Worker.establish(String)) && target(w) {}
   event rg before(Worker w):
      call(* Worker.reg(String)) && target(w) {}
   event is after(Worker w) returning(List result):
      call(* Worker.reg(String)) && target(w) {}
   event tr before(Worker w):
      call(* Worker.transfer(int)) && target(w) {}
   cfg : S -> epsilon | et U, U -> epsilon | rg V,
         V -> epsilon | is W, W -> epsilon | tr W
   @fail { System.err.println("Protocol violation"); }}
```

Figure 7.4: MOP

cation to be explicit on both valid and invalid method call sequences. For example, the specification in Figure 7.3 would allow `reg()` to be invoked immediately at the start state, as the transition from the start state is not defined, it is simply ignored by the monitoring framework.

Learnability

Learnability is the capability of a software product to enable the user to learn how to use it. Table 7.2 shows the number of hours spent on activities to specify and monitoring properties defined in Figure 6.5.

The most time spent at specification was for PQL; PQL defines a new specification language for expressing queries for (recursively) matching sequences of method invocations. We find the language to be counter-intuitive as it does not match any existing modeling or programming languages. Moreover, it requires the user to specify *invalid* behavior rather than valid ones and it is unclear how to specify method invocations with specific input values. Similarly Jassda lacks an integration into the general context of assertion checking, which is needed to specify properties of variable values.

	Documentation	**Maintenance**	**Support**
PQL	1 paper, examples	2006	Minimal
Jassda	papers, (German) thesis, examples	2006	Minimal
LARVA	papers, manuals, examples	2011	Immediate
MOP	papers, manuals, examples	2011	Immediate
SAGA	papers, examples	2012	Immediate

Table 7.3: Adoptability

LARVA provides an intuitive language for specifying regular protocols. Specifications are finite state automata with optionally actions (arbitrary Java code) on the transitions of the automaton. Actions can be used to express data-oriented properties, though in an imperative style. Context-free protocols are however much more cumbersome to express as noted previously. Despite the fact that the Worker property has only been formalized partially in LARVA due to requirements to express *all invalid* sequences of method invocations, the full specification in SAGA is much more concise.

Though it is no so difficult in MOP to formalize the protocol behavior of the Worker, Figure 7.4 (data-oriented properties are more problematic, as these cannot be expressed directly as mentioned), the meaning of the grammars in MOP is unclear: the failure handler was triggered by MOP even for correct programs. Whether this is due to misunderstanding on our part of the meaning of MOP specifications, or due to a bug in MOP remains unclear even after a thorough reading of the documentation.

For PQL, most time is spent identifying which Java statements are supported and how variables can be manipulated. The actual set-up of the run-time checking (compilation, instrumentation etc.) are carried by mirroring the setting in the toy examples provided by the installation package. For Jassda, time is spent at understanding the Java Debugger Architecture, and in particular the proper settings in the configuration files.

We evaluated how easily the frameworks can be **adopted** or integrated into the the software development cycle in an industrial context such as at Fredhopper. This includes operational steps like installation, execution, and documentation and support. The quality assurance process at Fredhopper (as in many other software companies) includes automated testing. This type of testing requires a running FAS instance and can be augmented with run-time assertion checking techniques. Lack of support and maintenance (Table 7.3) reduces the confidence in PQL and Jassda.

Future Work

The concurrent version of our run-time checker which is described in Chapter 6 currently supports only regular grammars and all attributes must be inherited. As described, this restriction allows efficient run-time checking since one does not need to store the full history (just storing the attribute values of the 'previous' history suffices), and one does not need to re-parse the full history when a message is added to it (instead, one simply executes a single step in a finite automaton). The single threaded version allows more general grammars, but at the cost of a more expensive parsing process (the entire history is stored, and completely re-parsed whenever an event occurs). This suggests a possible direction of future work: investigate if and how more general grammars can be parsed incrementally, and implement efficient parsers for that class of grammars. Some initial theoretical work has already been done in this direction by Hedin [43].

Another direction of future work concerns error reporting (by error, we mean here that a history has been reached during execution which violated a specification as given by the grammars). If the run-time checker detects an error, what information is reported, and how is it presented? Clearly it is cumbersome to read through long stacktraces and low-level details of Java virtual machines that one gets by simply executing the program inside a debugger. On the other hand, the reported error should be sufficient to isolate the incorrect part of the source-code if it is to be of use for fixing the error. As a first step, the current version of the run-time checker outputs a UML sequence diagram depicting an invalid history upon detection, but much work remains to be done. For instance, it is clearly infeasable to visualize large histories (or even any history containing more than 10.000 messages). Thus suitable abstractions must be found. In particular, the question arises: which of the messages in the history are actually relevant for the error that was found? Once this is known, the other messages can simply be filtered away.

A third opportunity for future work concerns off-line monitoring. The current run-time checker parses the current history during execution of the program in real-time and stops (or possibly corrects) the original program once a violation of the specification is caught. Therefore the run-time checker induces an overhead during execution. This can partly be alleviated on multicore machines by running the run-time checker in a separate virtual machine (this is done by default if one uses the Java debugger, as explained in Chapter 4), and running that virtual machine on another processor. However there will still be some communication to report the next method call to be executed between the two virtual machines, and communication between two processes generally decreases performance. An alternative would be to develop a run-time checker which writes the history to disk. This allows to investigate any potential errors at another time, potentially even on a physically completely separate machine! One possible downside of this alternative is that since errors are only detected at a later time, the running system is not prevented from unsafe behavior. To keep the sizes of the stored histories manageable, it is in this regard clearly also important to find efficient representations or abstractions of the history.

While our formalism was based on context-free grammars (extended with attributes and assertions), there are more expressive grammar formalisms, such as Boolean grammars [71] or context-sensitive grammars (see for example [63]). These formalisms still have a decidable parsing problem. Future work in this direction can be done by investigating if (and how) these formalisms can be extended by some form of attributes, similarly to how attribute grammars are an extension of context-free grammars.

A perhaps simpler line of future work would be to extend the tool, for example with wildcards in communication views, or associating some form of time to each communication event. Wildcards are useful for specifying patterns (or sets) of method calls. For instance, if a class contains multiple overloaded methods, and one wants identify all of them in the attribute grammar, using a wildcard as the list of parameters would be a simple solution which avoids explicitly listing all variants of the overloaded method in the comunication view. As another feature, one could store the time at which a method call occured as an additional built-in attribute in the grammar terminals. This additional attribute could be used to specify non-functional properties such as resource requirements. For instance, it allows to express properties like 'the method m should not be called within 1 second after n was called'. In this respect it would be interesting to compare the resulting attribute grammars with existing temporal logics.

Input and output of SAGA A

We show here an example of the input to SAGA (a communication view), and the corresponding generated Java code which SAGA generates for the History class. Figure A.1 shows the communication view of the Stack, which will serve as the example input.

The next pages show the corresponding automatically generated Java output.

```
local view StackHistory grammar Stack.g
 specifies StackInterface {
         call void push(int item) PUSH,
       return int pop() POP
}
```

Figure A.1: Input to the tool: communication view of the Stack

```
import java.util.IdentityHashMap; // stores local
   histories and objToId
import org.antlr.runtime.*; // for use in
   StackHistory
import java.util.ArrayList; // for use in
   StackHistory
import java.util.Iterator; // for use in
   StackHistory
import java.util.HashSet; // for use in StackHistory
import java.util.HashMap; // for use in StackHistory

aspect StackHistoryAspect {
    private StackHistory h = new StackHistory();

//
   /////////////////////////////////////////////////////

//////////////////////// Event classes
   /////////////////
//
   /////////////////////////////////////////////////////


    public class call_push extends org.antlr.runtime
       .CommonToken {
      private static final long serialVersionUID = 3
         L;


      private final Object caller;
      public Object caller() {
        return this.caller;
      }

      private final StackInterface callee;
      public StackInterface callee() {
        return this.callee;
      }

      private final int item;
      public int item() {
        return this.item;
      }


      public String toString() {
        return "o" + StackHistory.objToId.get(caller
           ) + ":o" + StackHistory.objToId.get(
           callee) + ".push(" + item + ")";
      }
```

```
  public call_push(Object caller, StackInterface
      callee, int item) {
    super(-1);


    this.caller = caller;

    this.callee = callee;

    this.item = item;

  }
}

public class return_pop extends org.antlr.
   runtime.CommonToken {
  private static final long serialVersionUID = 3
     L;


  private final Object caller;
  public Object caller() {
    return this.caller;
  }

  private final StackInterface callee;
  public StackInterface callee() {
    return this.callee;
  }

  private final int result;
  public int result() {
    return this.result;
  }


  public String toString() {
    return "o" + StackHistory.objToId.get(caller
       ) + ":o" + StackHistory.objToId.get(
       callee) + ".pop(" + result + ")";
  }

  public return_pop(Object caller,
     StackInterface callee, int result) {
    super(-1);


    this.caller = caller;
```

```
        this.callee = callee;

        this.result = result;

      }
    }



//
   /////////////////////////////////////////////////////
///////////////////////// History class
   /////////////////
//
   /////////////////////////////////////////////////////


    public static class StackHistory implements
       TokenSource {
      public  static IdentityHashMap<Object, Integer
         > objToId = new IdentityHashMap<Object,
         Integer>();
      public  static HashMap<Integer, Object>
         idToObj = new HashMap<Integer, Object>();
      private HashSet<Integer> actors = new HashSet<
         Integer>();

      private ArrayList<CommonToken> _L = new
         ArrayList<CommonToken>();
      private Integer _currentToken;
      private StackParser.start_return _start; //
         Synthesized attributes of start non-
         terminal

      public StackHistory() {
        _L.add(new CommonToken(Token.EOF));
        _L.add(new CommonToken(Token.EOF));

        parse(); // the empty history
      }

      // Implemented for TokenSource interface
      public String getSourceName() {
        return null;
      }

      public void print() {

        System.err.println("===␣ERROR!␣Local␣history
     ␣of␣view␣StackHistory␣(events:␣" +
     Integer.toString(_L.size()-2) + ")␣of␣
     StackInterface␣object␣violates␣protocol␣
     structure␣specified␣in␣Stack.g␣===␣\n");


  // Print actors of the sequence diagram
  Iterator<Integer> it = actors.iterator();
  while(it.hasNext()) {
          Integer objId = it.next();
      if(idToObj.get(objId) != null) {
            System.out.println("o" + objId + "
               :" + idToObj.get(objId).
               getClass().getName());
      } else {
            System.out.println("o" + objId + "
               :Object");
      }
  }
  // Print messages between actors
  System.out.println("");
  for(int i=0; i<_L.size()-2; i++) {
    System.out.println(_L.get(i).toString());
  }
}

public CommonToken nextToken() {
  return _L.get(_currentToken++);
}

// Parse the history in antlr and set
   attribute values
private void parse() {
  _currentToken = 0;
  CommonTokenStream tokens = new
     CommonTokenStream(this);
  StackParser parser = new StackParser(tokens)
     ;

  try {

    _start = parser.start();

  } catch(RecognitionException r) {
      print();
      assert false; // Assertion Failure
  } catch(AssertionError r) {
      print();
      assert false; // Assertion Failure
  }
```

```
}


public EList<Object> stack() {
  return _start.stack;
}

public int bla() {
  return _start.bla;
}



public void update(call_push e) {
     e.setType(StackLexer.PUSH);
     _L.add(_L.size()-2, e);


     if(!objToId.containsKey(e.caller())) { //
         for printing
         objToId.put(e.caller(), objToId.size
            ());
         idToObj.put(idToObj.size(),e.caller()
            );
     }
     if(!objToId.containsKey(e.callee())) { //
         for printing
         objToId.put(e.callee(), objToId.size
            ());
         idToObj.put(idToObj.size(),e.callee()
            );
     }


     if(!objToId.containsKey(e.item())) { //
        for printing
         objToId.put(e.item(), objToId.size())
            ;
         idToObj.put(idToObj.size(),e.item());
     }

     actors.add(objToId.get(e.caller()));
     actors.add(objToId.get(e.callee()));

     parse();
}

public void update(return_pop e) {
     e.setType(StackLexer.POP);
     _L.add(_L.size()-2, e);
```

```
           if(!objToId.containsKey(e.caller())) { //
               for printing
               objToId.put(e.caller(), objToId.size
                  ());
               idToObj.put(idToObj.size(),e.caller()
                  );
           }
           if(!objToId.containsKey(e.callee())) { //
               for printing
               objToId.put(e.callee(), objToId.size
                  ());
               idToObj.put(idToObj.size(),e.callee()
                  );
           }

           if(!objToId.containsKey(e.result())) { //
               for printing
               objToId.put(e.result(), objToId.size
                  ());
               idToObj.put(idToObj.size(),e.result()
                  );
           }


           actors.add(objToId.get(e.caller()));
           actors.add(objToId.get(e.callee()));

           parse();
      }


    }


//
   ///////////////////////////////////////////////////////

/////////////////////// Aspects
   ///////////////////////
//
   ///////////////////////////////////////////////////////


    /* call  void push(int item) */
    before(Object clr, StackInterface cle, int item)
       :
      (call( void *.push(int)) && this(clr) &&
         target(cle) && args(item)
         && if(StackHistoryAspect.class.
            desiredAssertionStatus() )) {
          cle.h.update(new call_push(clr, cle, item));
      }

      before(StackInterface cle, int item): // from
         static method
        (call( void *.push(int)) && !this(Object) &&
           target(cle) && args(item)
         && if(StackHistoryAspect.class.
            desiredAssertionStatus())) {
          cle.h.update(new call_push(null, cle, item))
             ;
      }


      /* return  int pop() */
      after(Object clr, StackInterface cle) returning(
         int ret):
        (call( int *.pop()) && this(clr) && target(cle
           ) && args()
         && if(StackHistoryAspect.class.
            desiredAssertionStatus() )) {
          cle.h.update(new return_pop(clr, cle, ret));
      }

      after(StackInterface cle) returning(int ret): //
          from static method
        (call( int *.pop()) && !this(Object) && target
           (cle) && args()
         && if(StackHistoryAspect.class.
            desiredAssertionStatus())) {
          cle.h.update(new return_pop(null, cle, ret))
             ;
      }



  }
```

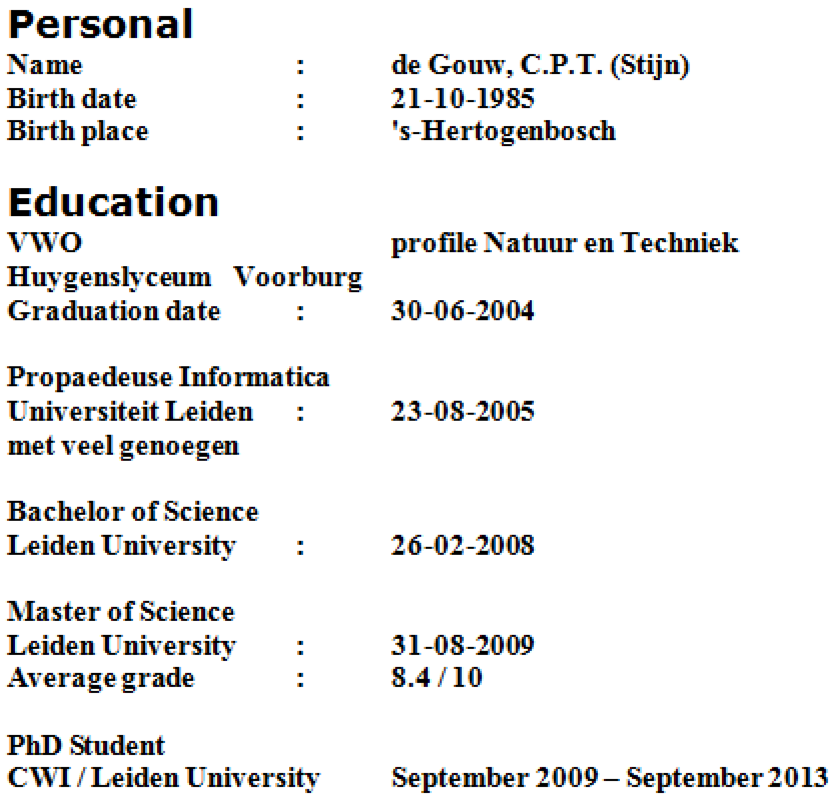
Samenvatting

Fouten in software veroorzaken jaarlijks een schade van 312 miljard dollar volgens een recent onderzoek aan Cambridge University. Type Checking, Static Verification en Run-time Checking zijn bekende manieren om softwarefouten te voorkomen, te isoleren, en op te lossen. Alle drie methoden bepalen of de software naar verwachting werkt op basis van annotaties: een formele beschrijving van de door de gebruiker gewilde werking van de software.

In dit proefschrift hebben we een nieuwe techniek voor Run-Time Checking voor twee object-georiënteerde talen ontwikkeld: Java en de ABS. De ABS is een taal die ontwikkeld is in het Europese HATS project en concurrency ondersteund in de vorm van groepen van objecten, waarbij meerdere groepen tegelijkertijd actief kunnen zijn. In object-georiënteerde talen communiceren objecten door elkaar berichten te sturen. Het gedrag van objecten is volledig bepaald door de volgorde en inhoud van deze berichten. Traditionele methoden voor Runtime Checking focussen *ofwel* exclusief op het beschrijven en testen van deze volgordes (Monitoring), *ofwel* op de beschrijving en het testen van de de gegevens in de berichten (Run-time Assertion Checking, Design by Contract). De methode geïntroduceerd in dit proefschrift **combineert** Monitoring met Runtime Assertion Checking.

Het basisidee van onze techniek is dat het gedrag van objecten formeel beschreven kan worden door een attribuutgrammatica uitgebreid met asserties. De onderliggende (contextvrije) grammatica specificeert de toegestane volgordes van de berichten, de attributen definiëren eigenschappen van de inhoud van de berichten, en de asserties beschrijven de toegestane waarden van deze eigenschappen. Als de asserties geen kwantoren bevatten dan is het beslisbaar of een executie van een programma voldoet aan de specificatie gegeven door zo een attribuutgrammatica. Wij hebben een nieuwe Run-time Checker voor attribuutgrammatica's ontwikkeld in de vorm van een meta-programma in de taal Rascal. Vervolgens hebben we de Run-time Checker toegepast op een industriële case van het bedrijf Fredhopper. Op basis van deze case study hebben we de efficiëntie van de Run-time Checker onderzocht en met succes een aantal fouten in de Fredhopper software ontdekt en opgelost.

Curriculum Vitae



Personal

Name	:	de Gouw, C.P.T. (Stijn)
Birth date	:	21-10-1985
Birth place	:	's-Hertogenbosch

Education

VWO		profile Natuur en Techniek
Huygenslyceum Voorburg		
Graduation date	:	30-06-2004
Propaedeuse Informatica		
Universiteit Leiden	:	23-08-2005
met veel genoegen		
Bachelor of Science		
Leiden University	:	26-02-2008
Master of Science		
Leiden University	:	31-08-2009
Average grade	:	8.4 / 10
PhD Student		
CWI / Leiden University		September 2009 – September 2013

Bibliography

[1] A. E. Abdallah, C. B. Jones, and J. W. Sanders, editors. *Communicating Sequential Processes: The First 25 Years, Symposium on the Occasion of 25 Years of CSP, London, UK, July 7-8, 2004, Revised Invited Papers*, volume 3525 of *Lecture Notes in Computer Science*. Springer, 2005.

[2] G. A. Agha. Actors: A model of concurrent computation in distributed systems. *MIT Press, Cambridge, MA, USA*, 1990.

[3] C. Allan, P. Avgustinov, A. S. Christensen, L. J. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching with free variables to AspectJ. In *OOPSLA*, pages 345–364, 2005.

[4] K. R. Apt, F. S. de Boer, E.-R. Olderog, and S. de Gouw. Verification of Object-Oriented programs: A transformational approach. *J. Comput. Syst. Sci.*, 78(3):823–852, 2012.

[5] 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 *Abstract State Machines*, pages 87–107, 2003.

[6] J. W. Backus. The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM conference. In *IFIP Congress*, pages 125–131, 1959.

[7] J. C. M. Baeten, T. Basten, and M. A. Reniers. *Process Algebra: Equational Theories of Communicating Processes.*

Cambridge University Press, New York, NY, USA, 1st edition, 2009.

[8] D. Bartetzko, C. Fischer, M. Möller, and H. Wehrheim. Jass - Java with assertions. *Electr. Notes Theor. Comput. Sci.*, 55(2), 2001.

[9] A. Bauer, M. Leucker, and C. Schallhart. Comparing LTL semantics for runtime verification. *J. Log. Comput.*, 20(3):651–674, 2010.

[10] B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *Lecture Notes in Computer Science*. Springer-Verlag, 2007.

[11] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. Uppaal — a tool suite for automatic verification of real–time systems. In *Proc. of Workshop on Verification and Control of Hybrid Systems III*, number 1066 in Lecture Notes in Computer Science, pages 232–243. Springer–Verlag, Oct. 1995.

[12] J. Berdine, C. Calcagno, and P. W. O'Hearn. Smallfoot: Modular automatic assertion checking with Separation Logic. In *FMCO*, pages 115–137, 2005.

[13] J. Berdine, B. Cook, and S. Ishtiaq. SLAyer: Memory safety for systems-level code. In *CAV*, pages 178–183, 2011.

[14] J. A. Bergstra and J. W. Klop. Act$_{\text{tau}}$: A universal axiom system for process specification. In *Algebraic Methods*, pages 447–463, 1987.

[15] Y. Bertot, P. Castran, G. Huet, and C. Paulin-Mohring. *Interactive theorem proving and program development : Coq'Art : the calculus of inductive constructions*. Texts in theoretical computer science. Springer, Berlin, New York, 2004.

[16] M. Brörkens and M. Möller. Dynamic event generation for runtime checking using the JDI. *Electr. Notes Theor. Comput. Sci.*, 70(4), 2002.

[17] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 7(3):212–232, 2005.

[18] C. Calcagno, H. Yang, and P. W. O'Hearn. Computability and complexity results for a spatial assertion language for data structures. In *FSTTCS*, pages 108–119, 2001.

[19] P. Chalin, P. R. James, and G. Karabotsos. JML4: Towards an industrial grade IVE for java and next generation research platform for JML. In *VSTTE*, pages 70–83, 2008.

[20] F. Chen and G. Rosu. MOP: an efficient and generic runtime verification framework. In *OOPSLA*, pages 569–588, 2007.

[21] Y. Cheon and A. Perumandla. Specifying and checking method call sequences of Java programs. *Software Quality Journal*, 15(1):7–25, 2007.

[22] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. Nusmv: A new symbolic model verifier. In *CAV*, pages 495–499, 1999.

[23] D. Clarke, M. Helvensteijn, and I. Schaefer. Abstract delta modeling. In *GPCE*, pages 13–22, 2010.

[24] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs*, pages 52–71, 1981.

[25] E. M. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT Press, 2001.

[26] C. Colombo, G. J. Pace, and G. Schneider. LARVA — safer monitoring of real-time java programs (tool paper). In *SEFM*, pages 33–37, 2009.

[27] W. Damm and D. Harel. LSCs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19(1):45–80, 2001.

[28] F. S. de Boer, M. M. Bonsangue, M. Steffen, and E. Ábrahám. A fully abstract semantics for UML components. In *FMCO*, pages 49–69, 2004.

[29] F. S. de Boer, S. de Gouw, E. B. Johnsen, A. Kohn, and P. Y. H. Wong. Run-time assertion checking of data- and protocol-oriented properties of Java programs: An industrial case study. *Transactions on Aspect-Oriented Software Development*, 11 (to appear), 2013.

[30] F. S. de Boer, S. de Gouw, and J. Vinju. Prototyping a tool environment for run-time assertion checking in JML with communication histories. In *Proceedings of the 12th Workshop on Formal Techniques for Java-Like Programs*, FTFJP '10, pages 6:1–6:7, New York, NY, USA, 2010. ACM.

[31] F. S. de Boer, S. de Gouw, and P. Y. H. Wong. Run-time verification of coboxes. In *SEFM*, 2013.

[32] S. de Gouw and F. S. de Boer. Run-time verification of black-box components using behavioral specifications: An experience report on tool development. In *FACS*, 2012.

[33] S. de Gouw, F. S. de Boer, W. Ahrendt, and R. Bubel. Weak arithmetic completeness of Object-Oriented first-order assertion networks. In *SOFSEM*, pages 207–219, 2013.

[34] S. de Gouw, F. S. de Boer, E. B. Johnsen, and P. Y. H. Wong. Run-time checking of data- and protocol-oriented properties of Java programs: an industrial case study. In *SAC*, pages 1573–1578, 2013.

[35] D. Distefano and M. J. Parkinson. jStar: towards practical verification for Java. In *OOPSLA*, pages 213–226, 2008.

[36] J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In *CAV*, pages 173–177, 2007.

[37] C. Fischer and H. Wehrheim. Behavioural subtyping relations for Object-Oriented formalisms. In *AMAST*, pages 469–483, 2000.

[38] R. W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposia in Applied Mathematics*, pages 19–32, Providence, Rhode Island, 1967. American Mathematical Society.

[39] D. M. Gabbay, A. Kurucz, F. Wolter, and M. Zakharyaschev. *Many-Dimensional Modal Logics: Theory and Applications*. Elsevier, 2003.

[40] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[41] D. Grune and C. J. Jacobs. *Parsing Techniques - A Practical Guide (Second Edition)*. Springer-Verlag, 2008.

[42] D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. MIT Press, Cambridge, MA, 2000.

[43] G. Hedin. Incremental attribute evaluation with side-effects. In D. Hammer, editor, *Compiler Compilers and High Speed Compilation, 2nd CCHSC Workshop, Berlin GDR, October 10-14, 1988, Proceedings*, volume 371 of *Lecture Notes in Computer Science*, pages 175–189. Springer, 1988.

[44] M. Heisel, W. Reif, and W. Stephan. Implementing verification strategies in the KIV-system. In *CADE*, pages 131–140, 1988.

[45] M. Hennessy. *Algebraic theory of processes*. MIT Press series in the foundations of computing. MIT Press, 1988.

[46] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with BLAST. In *SPIN*, pages 235–239, 2003.

[47] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[48] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[49] G. J. Holzmann. The model checker SPIN. *IEEE Trans. Software Eng.*, 23(5):279–295, 1997.

[50] International Telecommunication Union. ITU-T Recommendation Z.120: Message Sequence Chart (MSC). Technical report, ITU, Geneva, 2001.

[51] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. VeriFast: a powerful, sound, predictable, fast verifier for C and Java. In *Proceedings of the Third international conference on NASA Formal methods*, NFM'11, pages 41–55, Berlin, Heidelberg, 2011. Springer-Verlag.

[52] A. Jeffrey and J. Rathke. Java Jr: Fully abstract trace semantics for a core Java language. In S. Sagiv, editor, *14th European Symposium on Programming (ESOP'05)*, volume 3444 of *Lecture Notes in Computer Science*, pages 423–438. Springer-Verlag, 2005.

[53] R. Jhala and R. Majumdar. Software model checking. *ACM Comput. Surv.*, 41(4), 2009.

[54] E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In B. Aichernig, F. S. de Boer, and M. M. Bonsangue, editors, *Proc. 9th International Symposium on Formal Methods for Components and Objects (FMCO 2010)*, volume 6957 of *LNCS*, pages 142–164. Springer-Verlag, 2011.

[55] E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. *Software and System Modeling*, 6(1):35–58, Mar. 2007.

[56] S. C. Kleene. Representation of events in nerve nets and finite automata. *Automata Studies*, 1956.

[57] G. Klein and T. Nipkow. A machine-checked model for a Java-like language, virtual machine, and compiler. *ACM Trans. Prog. Lang. Syst.*, 28(4):619–695, 2006.

[58] P. Klint, T. van der Storm, and J. J. Vinju. Rascal: a domain specific language for source code analysis and manipulation. In A. Walenstein and S. Schupp, editors, *Proceedings of the IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2009)*, pages 168–177, 2009.

[59] D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[60] M. Z. Kwiatkowska, G. Norman, and D. Parker. Prism: Probabilistic symbolic model checker. In *Computer Performance Evaluation / TOOLS*, pages 200–204, 2002.

[61] L. Lee. Fast context-free grammar parsing requires fast boolean matrix multiplication. *J. ACM*, 49(1):1–15, 2002.

[62] X. Li, Z. Liu, and J. He. A formal semantics of UML sequence diagram. In *Australian Software Engineering Conference*, pages 168–177, 2004.

[63] J. C. Martin. *Introduction to Languages and The Theory of Computation*. McGraw-Hil, 2010.

[64] M. Martin, B. Livshits, and M. S. Lam. Finding application errors and security flaws using PQL: a Program Query Language. In *OOPLSLA*, 2005.

[65] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2 edition, 1997.

[66] L. Mikhajlov and E. Sekerinski. A study of the fragile base class problem. In *ECOOP*, 1998.

[67] R. Milner. Fully abstract models of typed λ-calculi. *Theoretical Comput. Sci.*, 4:1–22, 1977.

[68] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.

[69] R. Milner. *Communicating and Mobile Systems: The π-Calculus*. Cambridge University Press, New York, NY, USA, 1999.

[70] B. Nobakht, M. M. Bonsangue, F. S. de Boer, and S. de Gouw. Monitoring method call sequences using annotations. In *FACS*, pages 53–70, 2010.

[71] A. Okhotin. Conjunctive and Boolean grammars: The true general case of the context-free grammars. *Computer Science Review*, 9:27–59, 2013.

[72] T. J. Parr and R. W. Quong. Adding semantic and syntactic predicates to LL(k): pred-LL(k). In *In Computational Complexity*, pages 263–277. Springer-Verlag, 1994.

[73] B. C. Pierce. *Types and programming languages.* MIT Press, 2002.

[74] A. Pnueli. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 46–57, 1977.

[75] A. Pnueli and A. Zaks. PSL model checking and run-time verification via testers. In *FM*, pages 573–586, 2006.

[76] V. R. Pratt. Semantical considerations on Floyd-Hoare logic. In *FOCS*, pages 109–121, 1976.

[77] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74, 2002.

[78] D. Sangiorgi and D. Walker. *The Pi-Calculus - a theory of mobile processes.* Cambridge University Press, 2001.

[79] I. Schaefer, L. Bettini, V. Bono, F. Damiani, and N. Tanzarella. Delta-oriented programming of software product lines. In *SPLC*, pages 77–91, 2010.

[80] J. Schäfer and A. Poetzsch-Heffter. JCoBox: Generalizing active objects to concurrent components. In *European Conference on Object-Oriented Programming (ECOOP) (ECOOP'10)*, volume 6183 of *Lecture Notes in Computer Science*, pages 275–299. Springer-Verlag, June 2010.

[81] M. Sipser. *Introduction to the theory of computation.* PWS Publishing Company, 1997.

[82] M. Sirjani, A. Movaghar, A. Shali, and F. S. de Boer. Modeling and verification of reactive systems using Rebeca. *Fundam. Inform.*, 63(4):385–410, 2004.

[83] V. Stolz and F. Huch. Runtime verification of concurrent Haskell programs. *Electr. Notes Theor. Comput. Sci.*, 113:201–216, 2005.

[84] L. G. Valiant. General context-free recognition in less than cubic time. *J. Comput. Syst. Sci.*, 10(2):308–315, 1975.

[85] J. van den Berg and B. Jacobs. The LOOP Compiler for Java and JML. In *TACAS*, pages 299–312, 2001.

[86] P. H. J. van Eijk, C. Vissers, and M. Diaz, editors. *Formal Description Technique Lotos: Results of the Esprit Sedos Project.* Elsevier Science Inc., New York, NY, USA, 1989.

[87] W. Visser, K. Havelund, G. P. Brat, S. Park, and F. Lerda. Model checking programs. *Autom. Softw. Eng.*, 10(2):203–232, 2003.

[88] P. Y. H. Wong, E. Albert, R. Muschevici, J. Proença, J. Schäfer, and R. Schlatte. The ABS tool suite: modelling, executing and analysing distributed adaptable Object-Oriented systems. *STTT*, 14(5):567–588, 2012.

List of Figures

List of Tables

Titles in the IPA Dissertation Series since 2007

H.A. de Jong. *Flexible Heterogeneous Software Systems.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-01

N.K. Kavaldjiev. *A run-time reconfigurable Network-on-Chip for streaming DSP applications.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-02

M. van Veelen. *Considerations on Modeling for Early Detection of Abnormalities in Locally Autonomous Distributed Systems.* Faculty of Mathematics and Computing Sciences, RUG. 2007-03

T.D. Vu. *Semantics and Applications of Process and Program Algebra.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-04

L. Brandán Briones. *Theories for Model-based Testing: Real-time and Coverage.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-05

I. Loeb. *Natural Deduction: Sharing by Presentation.* Faculty of Science, Mathematics and Computer Science, RU. 2007-06

M.W.A. Streppel. *Multifunctional Geometric Data Structures.* Faculty of Mathematics and Computer Science, TU/e. 2007-07

N. Trčka. *Silent Steps in Transition Systems and Markov Chains.* Faculty of Mathematics and Computer Science, TU/e. 2007-08

R. Brinkman. *Searching in encrypted data.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-09

A. van Weelden. *Putting types to good use.* Faculty of Science, Mathematics and Computer Science, RU. 2007-10

J.A.R. Noppen. *Imperfect Information in Software Development Processes.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-11

R. Boumen. *Integration and Test plans for Complex Manufacturing Systems.* Faculty of Mechanical Engineering, TU/e. 2007-12

A.J. Wijs. *What to do Next?: Analysing and Op-*

timising System Behaviour in Time. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2007-13

C.F.J. Lange. *Assessing and Improving the Quality of Modeling: A Series of Empirical Studies about the UML*. Faculty of Mathematics and Computer Science, TU/e. 2007-14

T. van der Storm. *Component-based Configuration, Integration and Delivery*. Faculty of Natural Sciences, Mathematics, and Computer Science,UvA. 2007-15

B.S. Graaf. *Model-Driven Evolution of Software Architectures*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2007-16

A.H.J. Mathijssen. *Logical Calculi for Reasoning with Binding*. Faculty of Mathematics and Computer Science, TU/e. 2007-17

D. Jarnikov. *QoS framework for Video Streaming in Home Networks*. Faculty of Mathematics and Computer Science, TU/e. 2007-18

M. A. Abam. *New Data Structures and Algorithms for Mobile Data*. Faculty of Mathematics and Computer Science, TU/e. 2007-19

W. Pieters. *La Volonté Machinale: Understanding the Electronic Voting Controversy*. Faculty of Science, Mathematics and Computer Science, RU. 2008-01

A.L. de Groot. *Practical Automaton Proofs in PVS*. Faculty of Science, Mathematics and Computer Science, RU. 2008-02

M. Bruntink. *Renovation of Idiomatic Crosscutting Concerns in Embedded Systems*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-03

A.M. Marin. *An Integrated System to Manage Crosscutting Concerns in Source Code*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-04

N.C.W.M. Braspenning. *Model-based Integration and Testing of High-tech Multidisciplinary Systems*. Faculty of Mechanical Engineering, TU/e. 2008-05

M. Bravenboer. *Exercises in Free Syntax: Syntax Definition, Parsing, and Assimi-*

lation of Language Conglomerates. Faculty of Science, UU. 2008-06

M. Torabi Dashti. *Keeping Fairness Alive: Design and Formal Verification of Optimistic Fair Exchange Protocols*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2008-07

I.S.M. de Jong. *Integration and Test Strategies for Complex Manufacturing Machines*. Faculty of Mechanical Engineering, TU/e. 2008-08

I. Hasuo. *Tracing Anonymity with Coalgebras*. Faculty of Science, Mathematics and Computer Science, RU. 2008-09

L.G.W.A. Cleophas. *Tree Algorithms: Two Taxonomies and a Toolkit*. Faculty of Mathematics and Computer Science, TU/e. 2008-10

I.S. Zapreev. *Model Checking Markov Chains: Techniques and Tools*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-11

M. Farshi. *A Theoretical and Experimental Study of Geometric Networks*. Faculty of Mathematics and Computer Science, TU/e. 2008-12

G. Gulesir. *Evolvable Behavior Specifications Using Context-Sensitive Wildcards*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-13

F.D. Garcia. *Formal and Computational Cryptography: Protocols, Hashes and Commitments*. Faculty of Science, Mathematics and Computer Science, RU. 2008-14

P. E. A. Dürr. *Resource-based Verification for Robust Composition of Aspects*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-15

E.M. Bortnik. *Formal Methods in Support of SMC Design*. Faculty of Mechanical Engineering, TU/e. 2008-16

R.H. Mak. *Design and Performance Analysis of Data-Independent Stream Processing Systems*. Faculty of Mathematics and Computer Science, TU/e. 2008-17

M. van der Horst. *Scalable Block Processing Algo-*

rithms. Faculty of Mathematics and Computer Science, TU/e. 2008-18

C.M. Gray. *Algorithms for Fat Objects: Decompositions and Applications*. Faculty of Mathematics and Computer Science, TU/e. 2008-19

J.R. Calamé. *Testing Reactive Systems with Data - Enumerative Methods and Constraint Solving*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-20

E. Mumford. *Drawing Graphs for Cartographic Applications*. Faculty of Mathematics and Computer Science, TU/e. 2008-21

E.H. de Graaf. *Mining Semi-structured Data, Theoretical and Experimental Aspects of Pattern Evaluation*. Faculty of Mathematics and Natural Sciences, UL. 2008-22

R. Brijder. *Models of Natural Computation: Gene Assembly and Membrane Systems*. Faculty of Mathematics and Natural Sciences, UL. 2008-23

A. Koprowski. *Termination of Rewriting and Its Certification*. Faculty of Mathematics and Computer Science, TU/e. 2008-24

U. Khadim. *Process Algebras for Hybrid Systems: Comparison and Development*. Faculty of Mathematics and Computer Science, TU/e. 2008-25

J. Markovski. *Real and Stochastic Time in Process Algebras for Performance Evaluation*. Faculty of Mathematics and Computer Science, TU/e. 2008-26

H. Kastenberg. *Graph-Based Software Specification and Verification*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-27

I.R. Buhan. *Cryptographic Keys from Noisy Data Theory and Applications*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-28

R.S. Marin-Perianu. *Wireless Sensor Networks in Motion: Clustering Algorithms for Service Discovery and Provisioning*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-29

M.H.G. Verhoef. *Modeling and Validating Distributed Embedded Real-*

Time Control Systems. Faculty of Science, Mathematics and Computer Science, RU. 2009-01

M. de Mol. *Reasoning about Functional Programs: Sparkle, a proof assistant for Clean*. Faculty of Science, Mathematics and Computer Science, RU. 2009-02

M. Lormans. *Managing Requirements Evolution*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-03

M.P.W.J. van Osch. *Automated Model-based Testing of Hybrid Systems*. Faculty of Mathematics and Computer Science, TU/e. 2009-04

H. Sozer. *Architecting Fault-Tolerant Software Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-05

M.J. van Weerdenburg. *Efficient Rewriting Techniques*. Faculty of Mathematics and Computer Science, TU/e. 2009-06

H.H. Hansen. *Coalgebraic Modelling: Applications in Automata Theory and Modal Logic*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-07

A. Mesbah. *Analysis and Testing of Ajax-based Single-page Web Applications*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-08

A.L. Rodriguez Yakushev. *Towards Getting Generic Programming Ready for Prime Time*. Faculty of Science, UU. 2009-9

K.R. Olmos Joffré. *Strategies for Context Sensitive Program Transformation*. Faculty of Science, UU. 2009-10

J.A.G.M. van den Berg. *Reasoning about Java programs in PVS using JML*. Faculty of Science, Mathematics and Computer Science, RU. 2009-11

M.G. Khatib. *MEMS-Based Storage Devices. Integration in Energy-Constrained Mobile Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-12

S.G.M. Cornelissen. *Evaluating Dynamic Analysis Techniques for Program Comprehension*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-13

D. Bolzoni. *Revisiting Anomaly-based Network Intrusion Detection Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-14

H.L. Jonker. *Security Matters: Privacy in Voting and Fairness in Digital Exchange.* Faculty of Mathematics and Computer Science, TU/e. 2009-15

M.R. Czenko. *TuLiP - Reshaping Trust Management.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-16

T. Chen. *Clocks, Dice and Processes.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-17

C. Kaliszyk. *Correctness and Availability: Building Computer Algebra on top of Proof Assistants and making Proof Assistants available over the Web.* Faculty of Science, Mathematics and Computer Science, RU. 2009-18

R.S.S. O'Connor. *Incompleteness & Completeness: Formalizing Logic and Analysis in Type Theory.* Faculty of Science, Mathematics and Computer Science, RU. 2009-19

B. Ploeger. *Improved Verification Methods for Concurrent Systems.* Faculty of Mathematics and Computer Science, TU/e. 2009-20

T. Han. *Diagnosis, Synthesis and Analysis of Probabilistic Models.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-21

R. Li. *Mixed-Integer Evolution Strategies for Parameter Optimization and Their Applications to Medical Image Analysis.* Faculty of Mathematics and Natural Sciences, UL. 2009-22

J.H.P. Kwisthout. *The Computational Complexity of Probabilistic Networks.* Faculty of Science, UU. 2009-23

T.K. Cocx. *Algorithmic Tools for Data-Oriented Law Enforcement.* Faculty of Mathematics and Natural Sciences, UL. 2009-24

A.I. Baars. *Embedded Compilers.* Faculty of Science, UU. 2009-25

M.A.C. Dekker. *Flexible Access Control for Dynamic Collaborative Environments.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-26

J.F.J. Laros. *Metrics and Visualisation for Crime Analysis and Genomics.* Faculty of Mathematics and Natural Sciences, UL. 2009-27

C.J. Boogerd. *Focusing Automatic Code Inspections.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2010-01

M.R. Neuhäußer. *Model Checking Nondeterministic and Randomly Timed Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-02

J. Endrullis. *Termination and Productivity.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-03

T. Staijen. *Graph-Based Specification and Verification for Aspect-Oriented Languages.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-04

Y. Wang. *Epistemic Modelling and Protocol Dynamics.* Faculty of Science, UvA. 2010-05

J.K. Berendsen. *Abstraction, Prices and Probability in Model Checking Timed Automata.* Faculty of Science, Mathematics and Computer Science, RU. 2010-06

A. Nugroho. *The Effects of UML Modeling on the Quality of Software.* Faculty of Mathematics and Natural Sciences, UL. 2010-07

A. Silva. *Kleene Coalgebra.* Faculty of Science, Mathematics and Computer Science, RU. 2010-08

J.S. de Bruin. *Service-Oriented Discovery of Knowledge - Foundations, Implementations and Applications.* Faculty of Mathematics and Natural Sciences, UL. 2010-09

D. Costa. *Formal Models for Component Connectors.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-10

M.M. Jaghoori. *Time at Your Service: Schedulability Analysis of Real-Time and Distributed Services.* Faculty of Mathematics and Natural Sciences, UL. 2010-11

R. Bakhshi. *Gossiping Models: Formal Analysis of Epidemic Protocols*. Faculty of Sciences, Department of Computer Science, VUA. 2011-01

B.J. Arnoldus. *An Illumination of the Template Enigma: Software Code Generation with Templates*. Faculty of Mathematics and Computer Science, TU/e. 2011-02

E. Zambon. *Towards Optimal IT Availability Planning: Methods and Tools*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-03

L. Astefanoaei. *An Executable Theory of Multi-Agent Systems Refinement*. Faculty of Mathematics and Natural Sciences, UL. 2011-04

J. Proença. *Synchronous coordination of distributed components*. Faculty of Mathematics and Natural Sciences, UL. 2011-05

A. Moralı. *IT Architecture-Based Confidentiality Risk Assessment in Networks of Organizations*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-06

M. van der Bijl. *On changing models in Model-Based Testing*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-07

C. Krause. *Reconfigurable Component Connectors*. Faculty of Mathematics and Natural Sciences, UL. 2011-08

M.E. Andrés. *Quantitative Analysis of Information Leakage in Probabilistic and Nondeterministic Systems*. Faculty of Science, Mathematics and Computer Science, RU. 2011-09

M. Atif. *Formal Modeling and Verification of Distributed Failure Detectors*. Faculty of Mathematics and Computer Science, TU/e. 2011-10

P.J.A. van Tilburg. *From Computability to Executability – A process-theoretic view on automata theory*. Faculty of Mathematics and Computer Science, TU/e. 2011-11

Z. Protic. *Configuration management for models: Generic methods for model comparison and model co-evolution*. Faculty of Mathematics and Computer Science, TU/e. 2011-12

S. Georgievska. *Probability and Hiding in Concurrent Processes.* Faculty of Mathematics and Computer Science, TU/e. 2011-13

S. Malakuti. *Event Composition Model: Achieving Naturalness in Runtime Enforcement.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-14

M. Raffelsieper. *Cell Libraries and Verification.* Faculty of Mathematics and Computer Science, TU/e. 2011-15

C.P. Tsirogiannis. *Analysis of Flow and Visibility on Triangulated Terrains.* Faculty of Mathematics and Computer Science, TU/e. 2011-16

Y.-J. Moon. *Stochastic Models for Quality of Service of Component Connectors.* Faculty of Mathematics and Natural Sciences, UL. 2011-17

R. Middelkoop. *Capturing and Exploiting Abstract Views of States in OO Verification.* Faculty of Mathematics and Computer Science, TU/e. 2011-18

M.F. van Amstel. *Assessing and Improving the Quality of Model Transformations.* Faculty of Mathematics and Computer Science, TU/e. 2011-19

A.N. Tamalet. *Towards Correct Programs in Practice.* Faculty of Science, Mathematics and Computer Science, RU. 2011-20

H.J.S. Basten. *Ambiguity Detection for Programming Language Grammars.* Faculty of Science, UvA. 2011-21

M. Izadi. *Model Checking of Component Connectors.* Faculty of Mathematics and Natural Sciences, UL. 2011-22

L.C.L. Kats. *Building Blocks for Language Workbenches.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2011-23

S. Kemper. *Modelling and Analysis of Real-Time Coordination Patterns.* Faculty of Mathematics and Natural Sciences, UL. 2011-24

J. Wang. *Spiking Neural P Systems.* Faculty of Mathematics and Natural Sciences, UL. 2011-25

A. Khosravi. *Optimal Geometric Data Structures.* Faculty of Mathematics and Computer Science, TU/e. 2012-01

A. Middelkoop. *Inference of Program Properties with Attribute Grammars, Revisited.* Faculty of Science, UU. 2012-02

Z. Hemel. *Methods and Techniques for the Design and Implementation of Domain-Specific Languages.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2012-03

T. Dimkov. *Alignment of Organizational Security Policies: Theory and Practice.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2012-04

S. Sedghi. *Towards Provably Secure Efficiently Searchable Encryption.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2012-05

F. Heidarian Dehkordi. *Studies on Verification of Wireless Sensor Networks and Abstraction Learning for System Inference.* Faculty of Science, Mathematics and Computer Science, RU. 2012-06

K. Verbeek. *Algorithms for Cartographic Visualization.* Faculty of Mathematics and Computer Science, TU/e. 2012-07

D.E. Nadales Agut. *A Compositional Interchange Format for Hybrid Systems: Design and Implementation.* Faculty of Mechanical Engineering, TU/e. 2012-08

H. Rahmani. *Analysis of Protein-Protein Interaction Networks by Means of Annotated Graph Mining Algorithms.* Faculty of Mathematics and Natural Sciences, UL. 2012-09

S.D. Vermolen. *Software Language Evolution.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2012-10

L.J.P. Engelen. *From Napkin Sketches to Reliable Software.* Faculty of Mathematics and Computer Science, TU/e. 2012-11

F.P.M. Stappers. *Bridging Formal Models – An Engineering Perspective.* Faculty of Mathematics and Computer Science, TU/e. 2012-12

W. Heijstek. *Software Architecture Design in Global and Model-Centric Software Development.* Faculty of

Mathematics and Natural Sciences, UL. 2012-13

C. Kop. *Higher Order Termination.* Faculty of Sciences, Department of Computer Science, VUA. 2012-14

A. Osaiweran. *Formal Development of Control Software in the Medical Systems Domain.* Faculty of Mathematics and Computer Science, TU/e. 2012-15

W. Kuijper. *Compositional Synthesis of Safety Controllers.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2012-16

H. Beohar. *Refinement of Communication and States in Models of Embedded Systems.* Faculty of Mathematics and Computer Science, TU/e. 2013-01

G. Igna. *Performance Analysis of Real-Time Task Systems using Timed Automata.* Faculty of Science, Mathematics and Computer Science, RU. 2013-02

E. Zambon. *Abstract Graph Transformation – Theory and Practice.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2013-03

B. Lijnse. *TOP to the Rescue – Task-Oriented Programming for Incident Response Applications.* Faculty of Science, Mathematics and Computer Science, RU. 2013-04

G.T. de Koning Gans. *Outsmarting Smart Cards.* Faculty of Science, Mathematics and Computer Science, RU. 2013-05

M.S. Greiler. *Test Suite Comprehension for Modular and Dynamic Systems.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2013-06

L.E. Mamane. *Interactive mathematical documents: creation and presentation.* Faculty of Science, Mathematics and Computer Science, RU. 2013-07

M.M.H.P. van den Heuvel. *Composition and synchronization of real-time components upon one processor.* Faculty of Mathematics and Computer Science, TU/e. 2013-08

J. Businge. *Co-evolution of the Eclipse Framework and its Third-party Plugins.* Faculty of Mathematics and Computer Science, TU/e. 2013-09

S. van der Burg. *A Reference Architecture for Distributed Software Deployment*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2013-10

J.J.A. Keiren. *Advanced Reduction Techniques for Model Checking*. Faculty of Mathematics and Computer Science, TU/e. 2013-11

D.H.P. Gerrits. *Pushing and Pulling: Computing push plans for disk-shaped robots, and dynamic labelings for moving points*. Faculty of Mathematics and Computer Science, TU/e. 2013-12

M. Timmer. *Efficient Modelling, Generation and Analysis of Markov Automata*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2013-13

M.J.M. Roeloffzen. *Kinetic Data Structures in the Black-Box Model*. Faculty of Mathematics and Computer Science, TU/e. 2013-14

L. Lensink. *Applying Formal Methods in Software Development*. Faculty of Science, Mathematics and Computer Science, RU. 2013-15

C. Tankink. *Documentation and Formal Mathematics — Web Technology meets Proof Assistants*. Faculty of Science, Mathematics and Computer Science, RU. 2013-16

C. de Gouw. *Combining Monitoring with Run-time Assertion Checking*. Faculty of Mathematics and Natural Sciences, UL. 2013-17