Combining Monitoring with Run-time Assertion Checking

Gouw, C.P.T. de

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.