Diogo Miguel Gaspar de Sousa

Licenciado em Engenharia Informática

Preventing Atomicity Violations with Contracts

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática

Orientador: João Lourenço,
Prof. Auxiliar, Universidade Nova de Lisboa

Co-orientadora: Carla Ferreira,
Prof[a]. Auxiliar, Universidade Nova de Lisboa

Júri:

Presidente: Prof[a]. Armanda Rodrigues
Universidade Nova de Lisboa

Arguente: Prof. António Menezes Leitão
Instituto Superior Técnico

Vogal: Prof. João Lourenço
Universidade Nova de Lisboa

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Dezembro, 2013

Preventing Atomicity Violations with Contracts

Aos meus pais e irmã

Acknowledgements

I would like to thank my advisers João Lourenço and Carla Ferreira for their guidance, support, and patience. Lourenço, in particular, has guided me for a few years, and I am thankful for everything I have learned from him in this journey. I also thank Ricardo Dias, Tiago Vale, and João Leitão, who also played a key role in this thesis or preceding, related, projects.

I thank the Departamento de Informática, Faculdade de Ciências e Tecnologia of the Universidade Nova de Lisboa for providing me an amazing environment with a true sense of community. I also extend my gratitude to the following institutions for their financial support: Centro de Informática e Tecnologias da Informação of the FCT/UNL; and Fundação para a Ciência e Tecnologia in the research project Synergy-VM (PTDC/EIA-EIA/113613/2009), and again, the Departamento de Informática and Faculdade de Ciências e Tecnologia of the Universidade Nova de Lisboa.

A huge "thank you" to my family. I thank my father, José Sousa, that inspired me to learn and be curious, to make me fall in love with science, and to have the knowledge and patience to answer my uncountable questions about life, the universe, and everything. To my mother, Anabela Gaspar, for the understanding and for always being there for me. To my sister, Mafalda Sousa, whose cheerfulness make my visits to Leiria even better. To my grandparents, Arnaldo Gaspar, Maria Cardeira, Virgílio Sousa and Amélia Rascão, who gave me a great deal of support.

To my friends, Helder Martins, João Martins, Joana Roque, Andy Goncalves, and João Silva, with whom I shared the laboratory and collectively exorcise some frustrations. I also thank João DaCosta and Nuno Pimenta for many conversations that lead to much more interesting conclusions than 42.

Abstract

Concurrent programming is a difficult and error-prone task because the programmer must reason about multiple threads of execution and their possible interleavings. A concurrent program must synchronize the concurrent accesses to shared memory regions, but this is not enough to prevent all anomalies that can arise in a concurrent setting. The programmer can misidentify the scope of the regions of code that need to be atomic, resulting in atomicity violations and failing to ensure the correct behavior of the program. Executing a sequence of atomic operations may lead to incorrect results when these operations are co-related. In this case, the programmer may be required to enforce the sequential execution of those operations as a whole to avoid atomicity violations. This situation is specially common when the developer makes use of services from third-party packages or modules.

This thesis proposes a methodology, based on the design by contract methodology, to specify which sequences of operations must be executed atomically. We developed an analysis that statically verifies that a client of a module is respecting its contract, allowing the programmer to identify the source of possible atomicity violations.

Keywords: Atomicity Violation, Concurrency, Thread Safety, Design by Contract, Program Analysis

Resumo

A programação concorrente é um tarefa difícil e propensa a erros visto que o programador tem de ter em conta os vários *threads* de execução e as suas possíveis intercalações. O programa tem de sincronizar os acessos concorrentes a regiões de memória partilhada, mas isto não é suficiente para evitar todas as anomalias que podem ocorrer num cenário concorrente. O programador pode identificar incorretamente o âmbito das regiões de código que precisam de ser atómicas, resultando em violações de atomicidade que levam a um comportamento incorreto do programa. Correr uma sequencia de operações atómicas pode produzir resultados incorretos quando existe uma correlação entre essas operações. O programador pode ter que fazer a execução sequencial dessas operações atómicas como um todo para evitar violações de atomicidade. Esta situação é especialmente comum quando o programador usa operações de pacotes ou módulos de terceiros.

Esta tese propõe uma metodologia baseada na programação por contrato para especificar sequências de chamadas que devem ser executadas de forma atómica. Apresentamos uma analise que verifica estaticamente que um cliente de um módulo está a respeitar o seu contrato, permitindo ao programador identificar a causa de potenciais violações de atomicidade.

Palavras-chave: Violações de Atomicidade, Concorrência, *Thread Safety*, Programação por Contrato, Analise de Programas

Contents

1

Introduction

1.1 Context & Motivation

The clock frequency of processors stalled in 2005. Increasing the clock speed of a processor was sustained by advancements in microprocessor manufacturing, in particular reducing power consumption and improving the efficiency of the processor, which is a consequence of making smaller transistors. Subsequently the trade-off between power consumption and clock frequency became more and more expensive. This lead to design of multicore processors, which became a viable way to continue to improve the computing power of processors.

In order for a program to take advantage of multiprocessors it must be written in a concurrent paradigm. To do so the programmer must identify which computational steps are worth executing in separate threads of execution of that program. This allows the operating system's scheduler to assign threads of execution to different cores of the processor, thus achieving true parallelism, and reducing the real execution time of the program.

Concurrent programming introduces new challenges that do not arise in sequential programs and that can compromise the correctness of the program. One of these problems are *data races*, where multiple threads of execution of a process access the same memory location concurrently, which may cause an invalid state to be read or modified. To prevent data races the programmer must identify code regions where these concurrent accesses can occur and use a concurrency control mechanism to ensure mutual exclusion of threads in that region. There are several types of synchronization mechanisms which offer different trade-offs regarding performance, ensured properties and ease of use.

Listing 1 Program containing an atomicity violation (left), and a solution for that anomaly (right).

```
1  void schedule_event(DateTime dt,
2                      Event e)
3  {
4      if (calendar.isFree(dt))
5          calendar.schedule(dt,e);
6  }
```

```
1  void schedule_event(DateTime dt,
2                      Event e)
3  {
4      atomic
5      {
6          if (calendar.isFree(dt))
7              calendar.schedule(dt,e);
8      }
9  }
```

A concurrent program free of data races can still contain concurrent-related anomalies. For example, an operation may be intended to be atomic, but is implemented in two separate atomic regions. While executing that operation another thread may read an intermediate value between the execution of the two atomic phases, possibly obtaining an inconsistent result. Programs that use a modular design, such as in an object-oriented paradigm, are particularly prone to these errors, since the program will compose operations offered by a module[1]. Even if these operations are atomic, the programmer must be careful to compose them in the right way, and their composition may require additional synchronization.

1.2 Problem

One of the most common concurrency related anomalies are caused by atomicity violations [LPSZ08]. Atomicity violations are anomalies where the atomic regions were wrongly or incompletely identified by the programmer, and the program requires additional synchronization to ensure its correctness. This may be caused by the complete lack of synchronization in critical regions or if the scope of some synchronized regions is insufficiently small and must be broader to ensure that the program has the intended semantics.

Consider the example in Listing 1 (left), where we use a calendar module, modeled as an object of the class `DateTime`, that have methods to check if a time slot is free (`isFree()`) and to schedule an event in a time slot (`schedule()`). Here the programmer defined an auxiliary method to schedule events without they ever overlap. The method checks if the requested time slot if free on the calendar, and if so, the event is scheduled. Even if all methods in the calendar class are atomic, the method `schedule_event()` can give an incorrect result, by double booking the same slot with two different events. This anomaly can happen if two threads run this method concurrently. Figure 1.1 represents a thread scheduling where two events (`ea` and `eb`) were scheduled by concurrent threads at the same time slot, contradicting the expected behavior of this method.

[1] By "module" we mean any package of software that can only be access through a well defined interface.

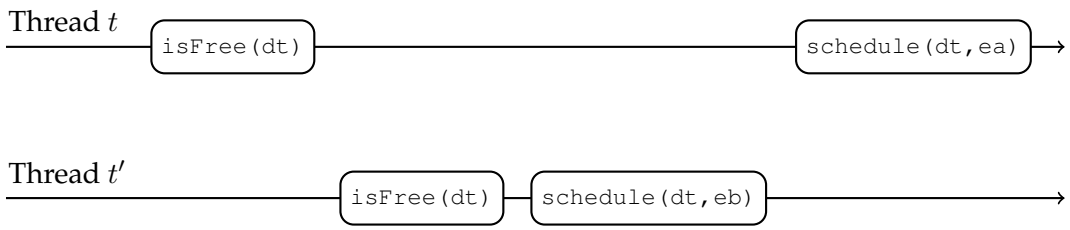


Figure 1.1: Example of concurrent thread scheduling leading to an atomicity violation.

The atomicity violation shown here is common when composing operations of a module, even if its methods are properly synchronized. Moreover, the programmer using the module may not be aware of its implementation and internal state, making it harder to judge what sequence of operations may cause atomicity violations.

1.3 Proposed Approach

To address this problem we propose a solution based on the design by contract methodology. With our solution the developer of the module defines a contract that will clearly specify what might constitute an atomicity violation when using the module. This specification stipulates which sequences of methods must be called in the same atomic scope to ensure that no atomicity violation will happen, preserving the correct behavior that is expected from the module.

The specification will act as a contract that the client code must respect for the module to have the expected behavior. The module's code can be annotated with this specification, providing documentation on how to safely use the module.

We propose a static analysis to verify that the client complies with the module's contract. This will ensure that the program is correct with respect to the module usage, or report the source of the violation if the contract is not met.

Consider again the example of the calendar module shown in Listing 1 (left). The contract of that module should say that calls to `isFree()` followed by `schedule()` should be atomic as a whole. Listing 1 (right) shows the correct implementation of `schedule_event()` respecting this rule.

1.4 Contributions

The contributions of this dissertation can be summarized as:

- Definition of a specification to represent the contract of a module's API. This specification clearly states what methods must be called in an atomic manner to preserve correctness.
- Design of an analysis to statically verify if the atomicity contract of a module is

respected, given the client code. Violations of the contract can easily be fixed with the information gathered by the analysis.

- Introduction of a novel approach to interprocedural control-flow static analysis, based on context-free grammars. The context-free grammar captures the control flow structure of a program and parsing algorithms can then be used to verify control-flow-related properties.
- Implementation of a working prototype that apply the proposed analysis. Our prototype analyses compiled *Java* programs and demonstrates the feasibility of our analysis in real-world programming languages.
- Evaluation of the accuracy and efficiency of the proposed analysis.

1.5 Publications

An article describing the proposed solution was published and presented in INForum 2013 [SFL13]. A more in-depth paper is in preparation for future publication.

A publicly available repository with the implemented prototype can be found on `https://github.com/trxsys/gluon`. This repository also includes the evaluation tests used in Chapter 5, and explained in Appendix A, as well as instructions on how to use the tool and to run the evaluation tests.

1.6 Outline

Chapter 1 introduced the subject addressed by this dissertation. It discusses the problem, its motivation, and the idea of our solution based on the design by contract programming methodology.

In Chapter 2 we present the related work of this thesis. We will cover the conditions for concurrent programs correctness and the common types of concurrency-related anomalies. We will also discuss proposed analysis techniques to detect these anomalies. This chapter will also address the design by contract methodology, and its application in subjects related with ours. This chapter will end with a view of the fundamentals of program analysis that will be useful to understand the analysis we propose.

Chapter 3 defines the methodology of our analysis. We will present the definition of a contract specification and the algorithm to verify the contract. This will be presented in an programming language-agnostic way, making the definition of the analysis more general. We also propose two extensions of the analysis to further improve the expressivity of a contract.

Chapter 4 discusses the implementation of our prototype, that was designed for the *Java* programming language. This chapter will deal with the prerequisites of the analysis described in Chapter 3 as well as other implementation decisions that were left open

in that chapter, since they are specific to the programming language being analyzed. Chapter 5 evaluates our prototype, both validating the correctness of our analysis, and discussing performance results. The tests used in this evaluation are presented in Appendix A.

Finally, Chapter 6 discusses the final remarks of this dissertation and proposes future directions to further improve this work.

2

Related Work

In this chapter we describe previous work related or useful to this thesis. This will contextualize how our contributions compare to the current state of the art of atomicity violation detection.

2.1 Concurrent Programs Correctness

Atomicity is a fundamental correctness property of concurrent systems. Intuitively a trace of a concurrent execution is said to be atomic if there is no interference between concurrent threads that lead to undesirable results. The notion of undesirable behavior varies from program to program, and the correctness criteria of a program may be tolerant to weaker guarantees than others. To address different applications requirements several atomicity semantics were defined, which draw different trade-offs between the guarantees offered and performance. Sometimes a developer may choose a weaker atomicity semantics for performance reasons, sacrificing correctness to some extent.

In this section we present the most important atomicity semantics.

Linearizability Linearizability [HW90] states that an atomic operation should appear to happen in some point in time between the start and end of the operation execution. This provides a mapping from every atomic operation in a given trace to a point of apparent execution in the global time line of the trace execution, called linearization point, maintaining causality between non-concurrent operations. Figure 2.1 shows an example of a linearization of concurrent atomic operations. This is a strong guarantee which is very desirable but may be expensive to enforce.

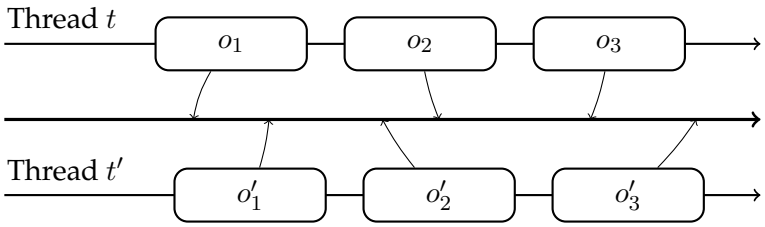


Figure 2.1: Example of linearizability of concurrent operations.

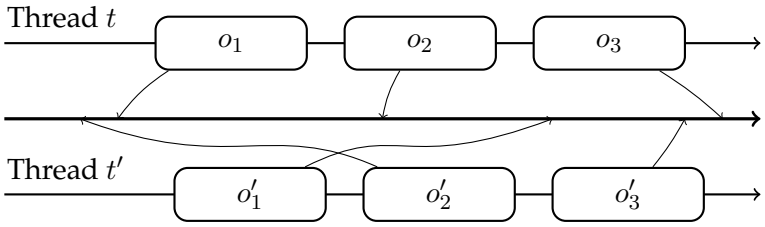


Figure 2.2: Example of serialization of concurrent operations.

Serializability The serializability semantics for atomic operations ensures that the state of the program after the concurrent execution of the atomic operations is the same as *some* sequential execution of those operations. This definition is more abstract than linearizability since the state of the program is taken into account and therefore the semantics of the atomic operations are relevant. It is easy to see that every trace that is linearizable is also serializable.

It follows from the definition that the atomic operations can run in an arbitrary order, as exemplified in Figure 2.2.

Strict Serializability In normal serializability we can freely choose a sequential scenario that is equivalent to our trace. Strict Serializability is strictly stronger than serializability, enforcing the additional condition that if an atomic operation A that finishes before the beginning of the execution of operation B in the concurrent trace, then A must occur before operation B in the equivalent sequential execution. Figure 2.3 gives an example of a strict serialization of concurrent atomic operations. Dashed lines denote invalid serialization points.

This atomicity model falls between serializability and linearizability in the correctness criteria hierarchy.

Snapshot Isolation Snapshot Isolation is the weakest atomicity semantic presented here. Conceptually, the atomic operations take a snapshot of the program state at their beginning and operate on this snapshot. When the atomic operation finishes, the changes made in the snapshot are written back to the program state. Additionally, snapshot isolation states that no two concurrent atomic operations write on overlapping state regions, i.e.,

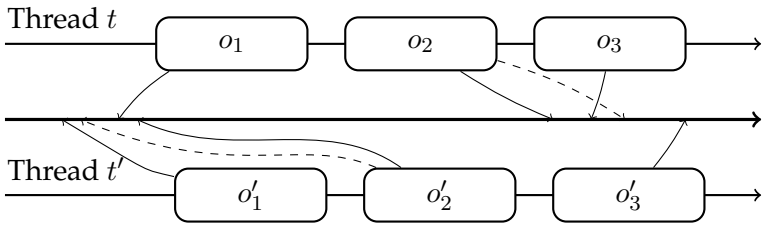


Figure 2.3: Example of strict serialization of concurrent operations.

in the same memory cells. This allows *write skew* anomalies to happen [BBGMOO95]. A write skew anomaly may occur when two transactions read overlapping memory regions that are then disjointly updated.

Even though this is a weak atomicity model, that allow well-known anomalies to happen, it is the most used model for database transactions since these anomalies are not common, and validating the transactions is made relatively cheap.

2.1.1 Race Conditions

Race conditions are anomalies that are caused by nondeterminism in a system, and lead to undesirable results. These anomalies were first identified in electronics, where the propagation time of a signal plus the reaction time of a logic gate introduce delays. Therefore, electric signals that theoretically arrive simultaneously drift by a small amount of time causing invalid outputs or an invalid state to be stored. One solution is to *synchronize* the events of a electronic circuit with a clock that gives periodic ticks. These ticks are points of synchronization where the whole state of the circuit is stable because enough time was given from the last tick to allow electric signals to fully propagate.

A similar definition applies to computer systems where different threads of execution run concurrently. The time of the execution of instructions of the multiple threads are not only arbitrary, but nondeterministic. This may lead to abnormalities if the threads share internal state. A solution to this is to impose synchronization between threads, reducing the universe of the execution traces that are allowed to run. Many synchronization mechanisms were introduced, some more low-level, such as locks/mutexes, barriers and semaphores; and others more high-level such as monitors and transactional memory.

Many races conditions are caused by data races and atomicity violations [LPSZ08] which will be discussed next.

Data Races A data race happens when two threads concurrently access the same memory region and at least one access is a write [NM92]. This may lead to invalid results, e.g. reading values while they are being updated. Data races can be avoided by enforcing *mutual exclusion* in some code regions, that is, using a synchronization mechanism to ensure that at most one thread is executing code inside that region. These code regions that require mutual exclusion are called *critical regions*.

Listing 2 An example of a program containing a benign data race.

```
1  unsigned int new_views;
2  unsigned int total_views;
3
4  ...
5
6  void new_hit()
7  {
8      new_hits++;
9      atomic { total_hits++; }
10
11     if (new_hits > 1000)
12     {
13         atomic { log_hits(total_hits); }
14         new_hits=0;
15     }
16 }
```

It is worth noticing that a data race may not be a race condition. By definition a race condition implies some violation of the program correctness. While in general a data race leads to an incorrect behavior, this may not always be the case. This is exemplified in Listing 2, where a web server maintains the number of hits on a certain HTTP resource. When 1000 new hits occur it logs the total of hits to keep a traffic history. The accesses to the shared variable `new_hits` are not protected and a data race can happen; however no real harm can happen, even if this variable became corrupted, it only causes the log to be immediately written or postponed. This data races are called *benign* since they occurrence does not compromise the intended correctness of the program.

Atomicity Violation Atomicity violations are race conditions where the lack of atomicity of the operations cause incorrect behavior of the program. This may be due to total lack of synchronization or to an incorrect scope of the synchronized blocks of code. The notion of atomicity violation is directly tied to the expected semantics of the program. Therefore an atomic violation is, by definition, a flaw in the program that causes incorrect results or behavior.

A program can contain atomicity violations while being free of data races. This can happen when a program executes two operations in two atomic phases, and the intended behavior is only assured if these two operations execute as a single atomic operation.

2.1.2 Detecting Atomicity Violations

Atomicity violations are among the most common cause of errors in concurrent programs [LPSZ08], and much research was done to try to identify this type of anomaly.

In this section we present some of the more relevant work in this area.

High-Level Data Races Artho et al defined the notion of view consistency in [AHB03]. View consistency violations are defined as high-level data races and represent sequences

Listing 3 An example of a program containing a stale value error.

```
1 void withdraw(int v)
2 {
3     int current;
4
5     atomic { current=account.getBalance(); }
6     atomic { account.setBalance(current-v); }
7 }
```

of atomic operations in the code that should be atomic as a whole. A *view of an atomic operation* is the set of variables that are accessed in that atomic operation. The set of views of a thread t is denoted as $V(t)$, and a thread τ is said to be compatible with a view v if, and only if, $\{v \cap v' \mid v' \in V(\tau)\}$ forms a chain, i.e., is totally ordered under $\subseteq$. The program is view consistent if every view from every thread is compatible with every other thread. The idea behind the definition of compatibility is that the view v implies a correlation between the variables it contains, and other threads should access these variables in a disciplined manner.

The notion of high-level data races (HLDR) does not capture every anomaly regarding the execution of atomic operations, and a HLDR does not imply a real atomicity violation. However this concept is precise enough to capture many real world anomalies.

This definition was subsequently extended by Praun and Gross [VPG04] to introduce *methods view consistency*. A method view is the union of the views inside a method of a class. The definition of method views consistency is analogous to the definition of view consistency. Furthermore this work distinguishes read and write accesses in the method views.

A further refinement of high-level data race was introduced by Dias et al in [DPL12]. This approach also takes into account the type of accesses in the views. They also refine the chain property for read accesses to reduce false positives: if there is no future dependency between the read variables of the two views the anomaly is not considered.

Stale Value Errors Stale value errors [BL04] are another type of anomalies that are also related to multiple atomic operations that should be treated as a single atomic operation. These anomalies are characterized by reading a value in an atomic operation and reusing that value on subsequent atomic operations. This may represents an atomicity violation because the value may be outdated, since it could have been updated by a concurrent thread. The freshness of the values may or may not be a problem depending on the application.

Listing 3 show an example of a program with a stale value error. Here the `withdraw()` contains two atomic regions, and the value `current`, that is obtained in the first atomic block, is used in the second atomic block. The second block updates the balance with a possibly outdated balance, thus overriding a concurrent modification of the account balance. This type of anomaly is common in concurrent programs, specially when the

atomic blocks are hidden behind a function or method call.

Many analysis address, directly or indirectly, stale value errors [AHB04; BL04; DPL12; LSTD11; FQ03; FF04; FFY08; VPG04; WS03]. We will briefly describe some of the analysis that directly target stale value errors. In [BL04] Burrows et al describe a dynamic analysis to detect stale value errors by instrumenting the program's code to keeps track of stale variables. The program checks at run-time that every variable read is not stale, and aborts the execution otherwise. In [AHB04] Artho et al proposes a static analysis to detect stale values. This analysis statically infers the flow of data escaping synchronization regions and reports usages of those values in other synchronization blocks.

Access Patterns In [VTD06] Vaziri et al define eleven access patterns that potentially represent an atomicity violation. These access patterns are sequences of read and write accesses denoted by $R_t(L)$ and $W_t(L)$ and represent, respectively, read and write accesses to memory locations L performed by thread t. The sequence order represents the execution order of the atomic operations. An example of an access pattern defined in Vaziri's paper is $R_t(x)\ W_{t'}(x)\ W_t(x)$, that represents a stale value error, since thread t is updating variable x based on an old value. The patterns make explicit use of the atomic set of variables, i.e., sets of variables that are correlated and must be accessed atomically. These correlated variables are assumed to be known. These eleven patterns are proved to be complete with respect to serializability.

A related approach by Teixeira et al [LSTD11] identifies three access patterns that capture a large number of anomalies. These anomalies are refereed to RwR, where two related reads are interleaved by a write in those variables; WrW where two related writes are interleaved by a read in those variables; and RwW that represents a stale value error.

Invariant Based Another approach to detect atomicity violations is by directly knowing the intended semantics of the program. This was the approach followed by Demeyer and Vanhoof in [DV12]. The authors defined a pure functional concurrent programming language that is a subset of *Haskell*, and includes the *IO Monad*, hence modeling sequential execution and providing shared variables that can be accessed inside atomic transactions. A specification of the invariants of the program's functions are provided by the programmer in logic. A shared variable is said to be consistent if all invariants related to it hold before and after every atomic transaction. The static analysis acquires the facts about the program and feeds them to a theorem prover to test if every shared variable is consistent.

This approach is very accurate provided that the programmer can express the notion of program correctness by using invariants on the global state, but is also expensive because a theorem prover is required to verify that the invariants hold.

Other Approaches Flanagan et al also proposed several methods for detecting atomicity violations [FF04; FFY08; FF10]. In [FF04] Flanagan presents a dynamic analysis for serializability violations. The central notion of this work are Lipton's reductions [Lip75].

If a reduction exists from one trace to another then the execution of both traces yields the same state (although different states may be obtained intermediately). Reductions can be found by commuting right- and left-mover operations. Their analysis specifies which operations are movers and uses a result from Lipton's Theory of Reductions, that specified the conditions for a reduction to exists. If these conditions hold he shows that there is a reduction from the atomic operations in the dynamically obtained concurrent trace, to a serialization trace. If these conditions are not met, then an anomaly is reported. This can, however, lead to false positives.

Another publication from Flanagan et al provides a sound and complete *dynamic* analysis for verifying serialization [FFY08]. This work uses a well-known result from database theory that states that a trace is serializable if and only if no cycle exists in the happens-before graph of the atomic operations [BHG87]. The dynamic analysis defined maintains this happens-before graph and report anomalies if a cycle is found.

A different approach is presented by Shacham et al in [SBASVY11]. In this work the atomic operations are extracted from the program to be analyzed to create an adversary that will run them concurrently to the original program. If two different runs yield different results then an anomaly is reported. Some heuristics are used to explore the search space of possible interleavings from the adversary, avoiding a prohibitively expensive exhaustive search.

In [WZJ11] Weeratunge et al introduce an analysis to detect and fix *Heisenbugs*. Heisenbugs are race conditions that, due to non-determinism, are hard to reproduce. Their technique collects traces from the application and analyze consecutive pairs of accesses to shared memory. If some trace has a pair of accesses interleaved with some concurrent access to the variables involved, then that pair of accesses is considered erroneous and are locks are placed to protect them. A generic concurrency bug fixing strategy is presented in [JZDLL12]. The detection of anomalies is given as a front-end to the fixer which then heuristically employ a fixing strategy.

In [LDZ13] Liu et al develop a method to identify incorrect atomic compositions when using a software module. Although this work and ours share the same goals, the approaches are very distinct. Their approach does not use design by contract, and instead they automatically infer what constitutes a module and where an atomic composition may be wrongly implemented. Two types of possible wrong compositions are identified: one value returned by a call to the module is used by the client in a subsequent call to that modules; and two methods of the module are always invoked together by the client program. This information is obtained statically. To verify that a composition identified as wrong truly leads to undesirable results, their tool dynamically checks those atomic compositions. This is achieved by executing different scheduling of the atomic operations and comparing the results with an observed normal execution. While this approach is interesting it has its disadvantages. Namely, the criteria used to identify wrong atomic compositions covers a limited number of scenarios that represent atomic violations, leading to false negatives; a correct trace of the program must be obtained in

the simulated execution of different scheduling; and the number of different scheduling of the atomic operations are exponential and may be unfeasible with a large number of atomic operations, even with pruning.

2.2 Design By Contract

Design by contract (DbC) is a software design technique to promote code reuse and software reliability, introduced by Meyer [Mey92]. The central notion of design by contract is that an object's method should be treated as a service. In real world a service provider requires certain conditions in order for a service to be applicable. For example, a delivery company requires the client to correctly provide the destination address, respect the maximum weight of packages, and, of course, pay. If these conditions are met by the client then the delivery company ensures the delivery of the package. Meyer suggests that these principles should also apply to object-oriented programming: a contract exists between the client of an object and that object. This contract specifies the preconditions the client code must meet in order for a method to be called and the postconditions that the method ensures after its execution. If the client calls the method without meeting the preconditions, no guarantee is offered about the correctness of the method's result.

Design by contract is an alternative to *defensive programming*, where the programmer assumes that a method may be called in any situation, with arbitrary arguments, and has to address all misuses that can happen. Listing 4 shows the function `sum(n)`, which computes the sum of the first n positive natural numbers, in both defensive programming (left) and design by contract methodology (right). The DbC implementation provides simpler code and documents the admissible usage scenarios and function semantics in the contract. In an object the pre- and postconditions can also refer to the internal object's state.

Listing 4 Function programmed in defensive programming (left) and design by contract (right).

```
1 int sum(int n)
2 {
3     if (n < 0)
4         // handle exception
5
6     return n*(n+1)/2;
7 }
```

```
1 @requires  n ≥ 0
2 @ensures   returns ∑_{i=0}^{n} i
3 int sum(int n)
4 {
5     return n*(n+1)/2;
6 }
```

The main advantage of design by contract is the increase in the reliability and readability of the code. This methodology clearly describes the conditions and semantics for methods to be called. This provides documentation for developers, a way of identifying bugs, clearly assigns blame if some condition is not met, and allows code correctness verification.

Verifying contracts can be done dynamically by testing the pre- and postconditions

assertions in run-time. This is the approach adopted by the Eiffel programming language [Mey87] where pre- and postconditions are syntactically part of the function definition. Other approaches use Hoare Logic [Hoa69] and theorem provers to statically verify that the code respects the contracts [FLLNSS02; BLS05].

The concept of design by contract methodology can be used to specify other types of contracts besides logic propositions over the program state. For instance, contracts can be used to describe access policies were certain methods are only allowed to be called by authenticated objects.

Concurrent Design by Contract Concurrency imposes a challenge to design by contract methodology, since a client cannot always ensure that the preconditions are met before a method call: a concurrent execution might change the global state and make that assertion false. In [Mey97] Meyer presents a solution to this problem based on monitors. A second type of preconditions are introduced with a different semantics: when a method is called and the precondition does not hold, the caller blocks until the precondition is satisfied. (We always assume mutual exclusion inside the object's methods.) The postconditions are only guaranteed to hold immediately after the method execution.

Meyer's solution may lead to deadlocks if the wait conditions are never met, thus failing to identify a bug. Nienaltowski et al propose a new contract semantics that applies both to sequential and concurrent programming [NMO09]: if the precondition depends only on the client, i.e., cannot be invalidated concurrently, then it must hold when the call is performed; otherwise the call is suspended until the precondition holds.

Contracts for Object Protocol Verification Some classes require that the sequence of method calls must obey some restrictions. The set of legal sequences of calls to methods is called the *class protocol*. As an example consider an object that represents a file: the first method to be called must be `open()`, then we can read or write to the file and we finish by closing the file. A natural way of specifying this is by using an finite state automaton where each state allows a set of methods to be called, as exemplified in Figure 2.4. Equivalently this may be represented by the regular expression $(\texttt{open}(\texttt{read}|\texttt{write})^*\texttt{close})^*$.

Beckman et al use typestate, a type system approach, to statically verify protocol compliance [BBA08]. The programmers define the object abstract states that correspond to the nodes in the finite state automaton. Each method requires the object to be in a defined abstract state and to make a transition to another state. In order for an object to be used it must first be *unpacked* to a specified abstract state, where only a subset of methods that corresponds with that state are available. The type system statically detect if an unpack operation leads to an invalid abstract states.

A dynamic analysis for method protocol verification was proposed by Cheon et al in [CP07] that uses regular expression-like annotations to specify the protocol. This notation extends JML (Java Modeling Language). Two different semantics for protocol compliance were suggested: in the weaker semantics a program respects the specification if

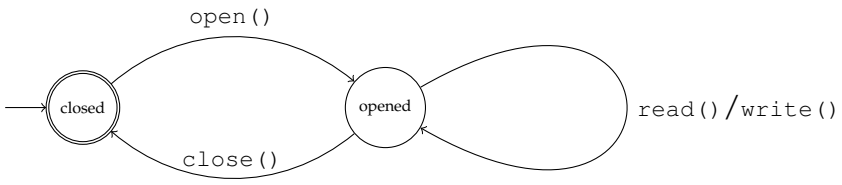


Figure 2.4: Example of a protocol specification represented by an automaton.

its calls are a supersequence of the specification, whether the strong semantics requires an exact matching. The tool they implemented offers dynamic checking of the weaker semantics by instrumenting the method calls with assertions that check and maintain an finite state automaton associated with the called object.

Hurlin builds upon the previous work of Cheon, extending it to specify concurrent protocols [Hur09]. This extends the expressiveness of the protocol specification language by adding a conditional construct where the method that can be called depending on a boolean expression over the objects state; and a construct to specify that two methods can run concurrently. Analyzing the protocol compliance is done by adding JML-like assertions to the original program. The analysis can be performed statically by using a theorem prover to check the satisfiability of the assertions.

2.3 Program Analysis

Program analysis consists of detecting properties about a piece of software. The information obtained about a program is usually used to automatically detect and report possible flaws about that program, to obtain debug information that can help the programmer understand the run-time behavior of the program, and to optimize the code of the program to achieve better run time or memory consumption.

This section will discuss the principal types and methodologies for program analysis, as well as the most common supporting data structures used to define an analysis.

2.3.1 Static and Dynamic Analysis

The two main approaches to program analysis are static and dynamic analysis. They differ in the way they gather information about the program and offer very distinct tradeoffs, both in efficiency and precision.

Dynamic analysis analyzes the program by monitoring its actions during the program's execution. This is usually done by instrumenting the code of the program to register the types of events that are relevant to the analysis. The registered information can then be processed during the program's execution or *post-mortem*. An alternative to

instrumenting the program's code is to run the program in a virtual machine that will keep track of the execution behavior of the program. Valgrind [NS03] is an example of a dynamic analysis tool that can be used to detect memory corruption and memory leak problems.

On the other hand, static analysis tries to infer the run-time behavior of the program by only analyzing its code. A static analysis takes as its input the source code or the compiled code of the program under analysis, and creates a model to represent the program. The representation will then be processed by the analysis algorithm to extract the relevant information about the program. These program representations will be further discussed in Section 2.3.2. The type system of a statically typed language is an example of a static analysis.

When the goal of the analysis is to detect possible flaws in the program, the static analysis approach tends to be preferred. This is because it is possible for the static analysis to detect *all* the flaws of the program[1], but this often leads to an over-approximation of the real anomalies of the program, reporting all the real flaws of the program and also *false positives*, i.e., occurrences that the analysis mark as a flaw, but are in fact innocent. Static analysis can be expensive to perform, depending on the complexity of the analysis.

Dynamic analysis can be advantageous if a static version of the analysis is too expensive in terms of time or space, or if a static version reports many false positives. Since dynamic analysis only analyzes specific runs of a program, it only explores a limited set of the program's states, which translate in a high number of *false negatives*, i.e., real flaws that are not reported. The trade-off is that the analysis has access to every run-time information that can lead to a high precision detection, which translates in a low or non-existent number of *false positives*[2].

Hybrid solutions exists, that perform static and dynamic analysis to programs. This types of analysis usually try to infer as much information as possible statically and complement this with a dynamic analysis to try to achieve the best of both worlds.

2.3.2 Program Representation

Static analysis verification tools usually offer different types of representation of the program under analysis, with different levels of abstraction. These representations of programs are used to extract the required information from simpler models of the program and avoid dealing with unnecessary information.

This section will cover some of the most commonly used representations of programs used in static analysis.

[1] An analysis that is guaranteed to detect all the flaws of a program is said to be *sound*.

[2] An analysis that is guaranteed never to yield false positives is said to be *complete*. Rice's theorem implies that no non-trivial static analysis, for Turing complete programming languages, can be both sound and complete.

```
1 while (n >= 0)
2 {
3     s+=n*n;
4     n--;
5 }
```

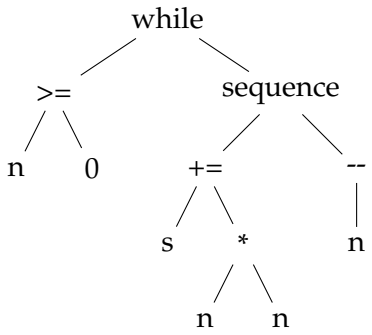


Figure 2.5: Example of an abstract syntax tree.

Abstract Syntax Tree The abstract syntax tree (AST) of a program represents the syntactic structure of that program's source code. This tree is generated by the source language parser, and gives a complete representation of the program. The nodes of the AST represent syntactic constructions of the language, such as *if*, *while*, *assignment*, *integer*, etc...

This kind of structure is created directly from the source code, explicitly encoding the grammatical structure of the source programming language, getting rid of unnecessary syntactic details (hence *abstract* syntax tree). Since it so closely represents the source code it is usually used to generate other data structures that are more suitable for program analysis.

Figure 2.5 exemplifies a abstract syntax tree (right) of a simple block of code (left). As shown the abstract syntax tree simply encodes the grammatical structure of the program.

Intermediate Representation An intermediate instruction representation is a language that represents the semantics of the program with simple instructions. This is beneficial for program optimization and analysis, because it breaks down a complex high-level language to a reduced set of instructions, which can more easily be processed by a static analysis algorithm. Another advantage is that an intermediate representation can be generated by different front-ends that process different high-level languages, making the analysis more generic.

A common type of intermediate languages is the *three address code*, where each instruction has, at most, three operands. This instructions can accept operands with constant values and memory addresses. These memory addresses do not necessarily correlate with concrete memory addresses, and can seen as registers in a virtual processor, that are latter translated in concrete register or memory addresses by the back-end of the compiler that generates the target machine's code. Another variation of intermediate representation is the *static single assignment form* (SSA) [RWZ88] where every variable is assigned only once (from a static point of view). This property can be advantageous for some types of analysis and code transformation. Listing 5 shows a three address code representation (right) of a for loop (left).

Listing 5 Example of three address code.

```
for (i=0; i < 10; i++)
    s=s+i*i;
```

```
        i = 0
L:      b = i >= 10
        if b goto E
        t = i*i
        s = s+t
        i = i+1
        goto L
E:
```

```
while (n != 1)
{
    if (n%2 == 0)
        n=n/2;
    else
        n=3*n+1;
    c++;
}
```

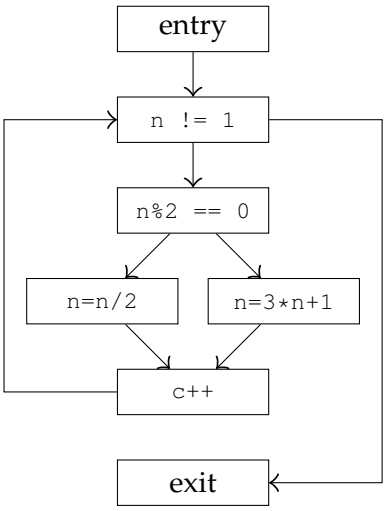


Figure 2.6: Example of a control flow graph.

Control Flow Graph A control flow graph [All70] of a method or function captures the control flow paths the method can take. In this graph the vertices represent single instructions of the program. An edge $u \to v$ represents that instruction u can be immediatly followed by the execution of instruction v. Therefore the successors of an instruction i in the control flow graph represent the different alternatives the program may take in run-time after executing i (unless the execution of i breaks the regular control flow, for instance, by raising an exception or terminating the program).

Figure 2.6 exemplifies a control flow graph (right) of a simple block of code (left). As can be seen, the control flow graph encompasses all the control flow logic of the program.

Call Graph A call graph [Ryd79] represents the call relations between methods or functions of a program. This is represented in a directed graph where the vertices are methods/functions, and an edge $u \to v$ exists if, and only if, the method u can directly calls method v. The graph contains no information about the order by which the methods are called nor the number of times the method is invoked.

This representation is fairly abstract, providing only a high-level view of the relations between methods, and is typically used to obtain information for more complex analysis. This graph can be used to easily detect dead code, i.e., code that is guaranteed not to be executed, since unused methods will form components that are not reachable from the

```
1  boolean even(int n)
2  {
3      if (n == 0) return true;
4      else return odd(n-1);
5  }
6
7  boolean odd(int n)
8  {
9      if (n == 0) return false;
10     else return even(n-1);
11 }
12
13 void g(int n) { return n*n; }
14
15 void f(int n)
16 {
17     if (odd(n)) return n+1;
18     else return g(n);
19 }
```

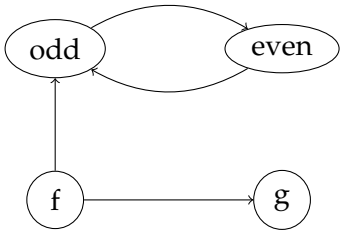


Figure 2.7: Example of a call graph.

main method. It is also easy to detect direct and indirect recursion since they form loops in the graph. Figure 2.7 shows an example of a call graph (right) and the corresponding methods (left).

2.3.3 Types of Static Code Analysis

Many static analysis are follow well-studied general methodologies for analysis design. The most common of these frameworks for analysis will be presented in this section.

Data Flow Analysis Data flow analysis [Coc70; All70; KU77] works directly on the control flow graph of a method. Each node of the control flow graph will have a value associated. The concrete analysis defines which types of values are associated with the nodes of the control flow graph, and how to propagate these values to the neighbor nodes. Each type of node can have specific rules on how to manipulate the values received from the adjacent nodes (forming a set of constraints relating the values of the nodes). A fixed-point algorithm is then used to propagate all information across the control flow graph until the values stabilize. (To ensure that a fixed-point is always reached the values associated with the nodes of the control flow graph must form a finite-height lattice over the constraints defined.) The final result of the analysis is the values associated with the nodes after the fixed-point is reached. If the analysis is defined in a conservative manner the results will always be sound.

Constraint Based Analysis Analysis based on Constraints [Aik94] are composed by two phases. First the analysis inspects the target code to generate a set of constraints that define the nature of the desired information about the program. For instance, these

constrains can be equation that express relations between variables. After the generation of set of constrains they must be resolved to obtain the desired end result of the analysis.

Abstract Interpretation Abstract interpretation [CC77] is a technique used to acquire information about the semantics of a program. The analysis defines an *abstract semantics* of the target programming language. This abstract semantics over-approximates the concrete semantics of the program, in order to avoid false positives, i.e., it considers all possible concrete execution of the program. The abstract semantic is defined in such a way that it always terminates, and computes the information required about the program.

As an example, say we want to know all the variables that are modified in a block of code. The abstract semantics for that analysis will record the variables written in every assignment, and simply propagate that set of written variables along the program. It will explore all alternatives of the conditional statements and will "iterate" every loop exactly once. The end result would be a superset of the variables accessed in every possible concrete execution of that block of code.

Type and Effect Systems Type systems [Car96] are the type of static analysis a programmer most often deals with. It assigns types (which can be seen as set of values) to syntactic expressions based on well-defined inference rules. If no rule applies to a program's expression the program is not well-typed and is rejected by the type system. The most common use of a type system is to avoid run-time errors by making sure that the values of expressions are acceptable in the context being used.

More sophisticated type systems may annotate the types of expressions with more information as to detect more complex errors.

Effect systems extends type systems with annotations about the effects that expression make produce. Examples of effects that can be described by effect systems are changing global variables, performing input/output, allocating memory, etc... An example of an effect system is the *Java* checked exceptions, which make sure that a method handles the exceptions that may be raised by the methods it calls.

3

Methodology

This chapter presents the static analysis for the verification of the module's contract. This analysis is the main contribution of this dissertation. We will define what constitutes a contract of the module and its semantics. The analysis has two fundamental phases: the extraction of the behavior of the program, and the verification of the module's contract based on the extracted behavior.

We also propose two extensions to the analysis that augment the expressivity of the contract, allowing a developer to specify in more detail what may lead to a atomicity violation.

3.1 Analysis Overview

The analysis we propose verifies statically if a client program complies with the contract of a given module. This is achieved by verifying that the threads launched by the program always execute atomically the sequence of calls defined by the contract.

This analysis has the following phases:

i) Determine the entry methods of the threads the program may launch.

ii) Determine which of the program's methods are atomically executed. A method is *atomically executed* if it is atomic[1] or if the method is always called by atomically executed methods.

[1] An atomic method is a method that explicitly applies a concurrency control mechanism to enforce atomicity.

iii) Extract the behavior of each of the program's threads with respect to the usage of the module under analysis.

iv) For each thread, check that its usage of the module respects the contract as defined in Section 3.2.

For the analysis to be applicable we require the following conditions: a) it must be possible to identify when the module is used in the target programming language; b) the accesses to the module are always done through a well defined API; c) it must be possible to identify the regions of code that run atomically; d) it must be possible to identify the starting point of the threads the program may launch.

In Section 3.2 we define the contract of the module. Section 3.3 we introduce the algorithm that extracts the program's behavior with respect to the module's usage. Section 3.4 defines the methodology that verifies whether the extracted behavior complies to the contract.

3.2 Contract Specification

The contract of a module specifies which sequences of calls of its methods must be executed atomically, as to avoid atomicity violations in the module's client program. In the spirit of the *design by contract* methodology, we assume that the definition of the contract, including the identification of which sequences of calls should be executed atomically, is a responsibility of the module's developer.

Definition 1 (Contract)**.** The contract of a module with public methods $m_1, \cdots, m_n$ is of the form,

$$1.\ e_1$$
$$2.\ e_2$$
$$\vdots$$
$$k.\ e_k$$

where each clause $i = 1, \cdots, k$ is described by e_i, a star-free regular expression over the alphabet $\{m_1, \cdots, m_n\}$. Star-free regular expressions are regular expressions without the Kleene star, using only the alternative (|) and the concatenation (implicit) operators.

Each sequence defined in e_i must be executed atomically by the program using the module, otherwise there is a violation of the contract. The contract specifies a finite number of sequences of calls, since it is the union of star-free languages. Therefore, it is possible to have the same expressivity by explicitly enumerating all sequences of calls, i.e., without the use of the alternative operator. We chose to offer the alternative operator so that the programmer can group similar scenarios under the same clause.

Since the verification analysis assumes that the contract defines a finite number of call sequences, this excludes the Kleene star operator. However, in Section 3.5.1, we describe

a method to simulate this operator, maintaining a finite number of call sequences, at the expense of precision loss.

Example Consider the class `java.util.ArrayList` that represents arrays, offered by the *Java* standard library. For simplicity we will only consider a few methods: `add(obj)`, `get(idx)`, `set(idx, obj)`, `contains(obj)`, `indexOf(obj)`, `remove(idx)`, and `size()`.

The following contract defines some of the clauses for this class.

1. `contains indexOf`
2. `indexOf` (`remove` | `set` | `get`)
3. `size` (`remove` | `set` | `get`)
4. `add indexOf`.

Clause 1 of `ArrayList`'s contract denotes that the execution of `contains()` followed by `indexOf()` should be atomic, otherwise the client program may confirm the existence of an object in the array, but fail to obtain its index due to a concurrent modification. Clause 2 represents a similar scenario where, the position of the obtained object is modified. In clause 3 we deal with the common situation where the program verifies if a given index is valid before accessing the array. To make sure that the size obtained by `size()` is valid when accessing the array we should execute these calls atomically. Clause 4 represents scenarios where an object is added to the array and then the program tries to obtain information about that object by querying the array.

Another relevant clause is `contains indexOf` (`remove` | `set` | `get`), but the contract's semantic already enforces the atomicity of this clause as a consequence of the composition of clauses 1 and 2, as they overlap in the `indexOf()` method.

3.3 Extracting the Behavior of a Program

The behavior of the program with respect to the module usage can be modeled as the individual behavior of all the threads the program may launch. The usage of a module by a thread t of a program can be described by a language L over the alphabet $m_1, \cdots, m_n$, the public methods of the module. A word $m_1 \cdots m_n \in L$ if some execution of t may run the sequence of calls $m_1, \cdots, m_n$ to the module.

To extract the usage of a module by a client program, our analysis generates a context-free grammar that represents the language L of a thread t of the client program, which is represented by its control flow graph (CFG). The CFG of the thread t represents every possible path that the control flow may take during its execution. In other words, the analysis generates a grammar G_t such that, if there is an execution path of t that runs the sequence of calls $m_1, \cdots, m_n$, then $m_1 \cdots m_n \in \mathcal{L}(G_t)$. (The language represented by a grammar G is denoted by $\mathcal{L}(G)$.)

Definition 2 (Thread Behavior Grammar)**.** The grammar $G_t = (N, \Sigma, P, S)$, is built from the CFG of the client's program thread t.

We define,

- N, the set of non-terminals, as the set of nodes of the CFG. Additionally we add non-terminals that represent each method of the client's program (represented in calligraphic font);
- Σ, the set of terminals, as the set of identifiers of the public methods of the module under analysis (represented in bold);
- P, the set of productions, as described bellow, by rules 3.1–3.5;
- S, the grammar initial symbol, as the non-terminal that represents the entry method of the thread t.

For each method `f()` that thread t may run we add to P the productions that respect the rules 3.1–3.5. Method `f()` is represented by $\mathcal{F}$. A CFG node is denoted by $\alpha : [\![v]\!]$, where α is the non-terminal that represents the node and v its type. We distinguish the following types of nodes: *entry*, the entry node of method $\mathcal{F}$; *mod.h()*, a call to method `h()` of the module *mod* under analysis; *g()*, a call to another method `g()` of the client program; and *return*, the return point of method $\mathcal{F}$. The $succ : N \to \mathcal{P}(N)$ function is used to obtain the successors of a given CFG node.

$$\text{if } \alpha : [\![\text{entry}]\!], \quad \{\mathcal{F} \to \alpha\} \cup \{\alpha \to \beta \mid \beta \in succ(\alpha)\} \subset P \tag{3.1}$$

$$\text{if } \alpha : [\![\text{mod.h()}]\!], \quad \{\alpha \to \mathbf{h}\ \beta \mid \beta \in succ(\alpha)\} \subset P \tag{3.2}$$

$$\text{if } \alpha : [\![\text{g()}]\!], \quad \{\alpha \to \mathcal{G}\ \beta \mid \beta \in succ(\alpha)\} \subset P \quad \text{where } \mathcal{G} \text{ represents } \texttt{g()} \tag{3.3}$$

$$\text{if } \alpha : [\![\text{return}]\!], \quad \{\alpha \to \epsilon\} \subset P \tag{3.4}$$

$$\text{if } \alpha : [\![\text{otherwise}]\!], \quad \{\alpha \to \beta \mid \beta \in succ(\alpha)\} \subset P \tag{3.5}$$

No more productions belong to P.

The rules 3.1–3.5 capture the structure of the CFG in the form of a context-free grammar. Intuitively this grammar represents the flow control of the thread t of the program, ignoring everything that is not related with the module's usage. The grammar is built in such a way that if $\mathbf{f}\ \mathbf{g} \in \mathcal{L}(G_t)$ then the thread t may invoke method `mod.f()`, followed by `mod.g()`.

Rules 3.1–3.5 preserve the structure of the control flow graph, so that every path in the graph corresponds to a derivation in the grammar. Rule 3.1 adds a production that relates the non-terminal $\mathcal{F}$, that represents method `f()`, to the entry node of the CFG of `f()`. This will allow other productions that reference method `f()` to use the non-terminal $\mathcal{F}$. In Rule 3.2 we treat calls to the module under analysis by recording them in the grammar.

```
1  void f()
2  {
3      m.a();
4      if (cond)
5          g();
6      m.b();
7  }
8
9  void g()
10 {
11     m.c();
12     if (cond)
13     {
14         g();
15         m.d();
16         f();
17     }
18 }
```

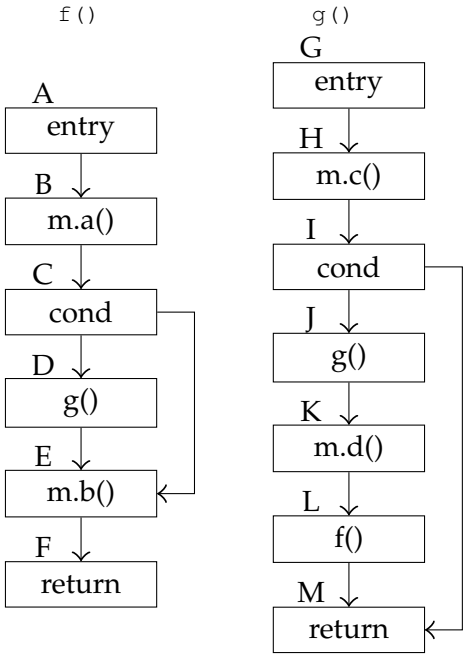


Figure 3.1: Program with recursive calls using the module `m` (left) and respective CFG (right).

If a CFG node A calls `mod.h()` and has a successor B then the production $A \rightarrow \mathbf{h}\, B$ will be in the grammar, and can be read as "non-terminal A generates all words of B prefixed with $\mathbf{h}$". Rule 3.3 handles calls to another method `g()` of the client program (method `g()` will have its non-terminal $\mathcal{G}$ added by Rule 3.1). The return point of a method simply adds an ϵ production to the grammar (Rule 3.4). All others types of CFG nodes are handled uniformly, preserving the CFG structure by making them reducible to the successor non-terminals (Rule 3.5). It is important to notice that only the client program code is analyzed.

The G_t grammar may be ambiguous, i.e., offer several different derivations to the same word. Each ambiguity in the parsing of a sequence of calls $m_1 \cdots m_n \in \mathcal{L}(G_t)$ represents different contexts where these calls can be executed by thread t. It is, therefore, necessary to allow such ambiguities so that the verification of the contract can cover all the occurrences of the sequences of calls in the client program.

The language $\mathcal{L}(G_t)$ contains every sequence of calls the program may execute, i.e., it produces no false negatives. However $\mathcal{L}(G_t)$ may contain sequences of calls that the program does not execute (for instance calls performed inside a block of code that is never executed), which may lead to false positives.

Examples Figure 3.1 (left) shows a program that consists of two methods that call each other mutually. We assume that method `f()` is the entry point of the thread. The module under analysis is represented by object `m`. The control flow graphs of these methods are shown in Figure 3.1 (right). According to Definition 2, we construct the grammar

$G_1 = (N_1, \Sigma_1, P_1, S_1)$, where

$$N_1 = \{\mathcal{F}, \mathcal{G}, A, B, C, D, E, F, G, H, I, J, K, L, M\},$$
$$\Sigma_1 = \{\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}\},$$
$$S_1 = \mathcal{F},$$

and P_1 has the following productions:

$$\begin{array}{ll} \mathcal{F} \to A & \mathcal{G} \to G \\ A \to B & H \to \mathbf{c}\, I \\ B \to \mathbf{a}\, C & I \to J \mid M \\ C \to D \mid E & J \to \mathcal{G}\, K \\ D \to \mathcal{G}\, E & K \to \mathbf{d}\, L \\ E \to \mathbf{b}\, F & L \to \mathcal{F}\, M \\ F \to \epsilon & M \to \epsilon \end{array}$$

In this example we can see how the grammar mimics the control flow graph structure. For each edge $u \to v$ of the CFG the grammar includes a production of the form $u \to \cdots v$, which captures the control flow of the program as a grammar.

Context-free grammars can easily represent loops and recursion of the code. This example show how a recursive call is encoded in the grammar: since methods are represented by non-terminals (which we show in a calligraphic font) we can use them in the body of any production. The example above shows this, for instance, in the production $J \to \mathcal{G}\, K$ that represents a direct recursive call to `g()`.

A second example, shown in Figure 3.2, exemplifies how the Definition 2 handles a flow control with loops. In this example we have a single function `f()`, which is assumed to be the entry point of the thread. For this example we have $G_2 = (N_2, \Sigma_2, P_2, S_2)$, with

$$N_2 = \{\mathcal{F}, A, B, C, D, E, F, G, H\},$$
$$\Sigma_2 = \{\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}\},$$
$$S_2 = \mathcal{F}.$$

The set of productions P_2 is,

$$\begin{array}{ll} \mathcal{F} \to A & E \to \mathbf{c}\, F \\ A \to B & F \to B \\ B \to \mathbf{a}\, C \mid \mathbf{a}\, G & G \to \mathbf{d}\, H \\ C \to D \mid E & H \to \epsilon \\ D \to \mathbf{b}\, F. & \end{array}$$

```
1  void f()
2  {
3      while (m.a())
4      {
5          if (cond)
6              m.b();
7          else
8              m.c();
9
10         count++;
11     }
12
13     m.d();
14 }
```

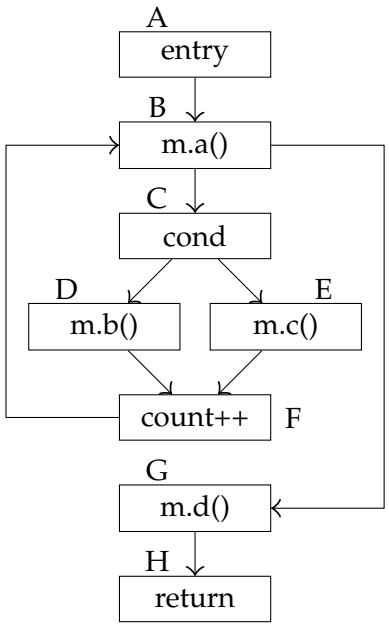


Figure 3.2: Program using the module `m` (left) and respective CFG (right).

To better understand how the grammar deals with loops we will show a derivation of the word **a b a c a d**. The only way for the program to perform these calls to module `m` is to iterate the loop exactly twice, entering the "then" branch of the if in the first iteration and the "else" branch in the second.

The derivation of **a b a c a d** is unambiguously done as follows:

$$\mathcal{F} \Rightarrow A \Rightarrow B \Rightarrow \mathbf{a}\, C \Rightarrow \mathbf{a}\, D \Rightarrow \mathbf{a\ b}\, F \Rightarrow \mathbf{a\ b}\, B \Rightarrow \mathbf{a\ b\ a}\, C \Rightarrow \mathbf{a\ b\ a}\, E$$
$$\Rightarrow \mathbf{a\ b\ a\ c}\, F \Rightarrow \mathbf{a\ b\ a\ c}\, B \Rightarrow \mathbf{a\ b\ a\ c\ a}\, G \Rightarrow \mathbf{a\ b\ a\ c\ a\ d}\, H \Rightarrow \mathbf{a\ b\ a\ c\ a\ d}.$$

3.4 Contract Verification

The verification of a contract must ensure that all sequences of calls specified by the contract are executed atomically by all threads the client program may launch. Since there is a finite number of call sequences defined by the contract we can verify each of these sequences to check if the contract is respected.

Algorithm 1 presents the pseudo-code of the algorithm that verifies a contract against a client's program. For each thread t of a program P, it is necessary to determine if (and where) any of the sequences of calls defined by the contract $w = m_1, \cdots, m_n$ occur in P (line 4). To do so, each of these sequences are parsed by the grammar G'_t (line 5), the grammar includes all words and sub-words of G_t. Sub-words must be included since we want to take into account partial traces of the execution of thread t.

Notice that G'_t may be ambiguous. Each different parsing tree represents different locations where the sequence of calls $m_1, \cdots, m_n$ may occur in thread t. Function `parse()` returns the set of these parsing trees. Each parsing tree contains information about the

Algorithm 1 Contract verification algorithm.

Require: P, client's program;
C, module contract (set of allowed sequences).

```
1:  for t ∈ threads(P) do
2:      G_t ← build_grammar(t)
3:      G'_t ← subword_grammar(G_t)
4:      for w ∈ C do
5:          T ← parse(G'_t, w)
6:          for τ ∈ T do
7:              N ← lowest_common_ancestor(τ, w)
8:              if ¬run_atomically(N) then
9:                  return ERROR
10: return OK
```

location of each method call of $m_1, \cdots, m_n$ in program P (since non-terminals represent CFG nodes). Additionally, by going upwards in the parsing tree, we can find the node that represents the method under which all calls to $m_1, \cdots, m_n$ are performed. This node is the lowest common ancestor of terminals $m_1, \cdots, m_n$ in the parsing tree (line 7). Therefore we have to check that the obtained lowest common ancestor is always executed atomically (line 8) to make sure that the whole sequence of calls is executed under the same atomic context. Since it is the *lowest* common ancestor we are sure to require minimal synchronization from the program. A parsing tree contains information about the location in the program where a contract violation may occur, therefore we can offer detailed instructions to the programmer on where this violation occurs and how to fix it.

Grammar G_t can use all the expressiveness offered by context-free languages. For this reason it is not sufficient to use the $LR(\cdot)$ parsing algorithm [Knu65], since it does not handle ambiguous grammars. To deal with the full class of context-free languages a GLR parser (Generalized LR parser) must be used. GLR parsers explores all the ambiguities that can generate different derivation trees for a word. We distinguish three parsing algorithms that offer good worst-case time complexity and are therefore suitable to be used: Tomita [Tom87], CYK [You67], and Earley [Ear70].

Another important point is that the number of parsing trees may be infinite. This is due to loops in the grammar, i.e., derivations from a non-terminal to itself ($A \Rightarrow \cdots \Rightarrow A$). An infinite number of parsing trees may occur in a loop in the grammar that is not productive to parse the grammar, and the parsing algorithm can choose to iterate that loop an arbitrary number of times, creating one parsing branch for each possibility. This often occur in G_t (every loop in the control flow graph will yield a corresponding loop in the grammar). For this reason the `parse()` function must detect and prune parsing branches that will lead to redundant loops, ensuring that a finite number of finite-height parsing trees is returned. To achieve this the parsing algorithm must detect a loop in the list of reduction it has applied in the current parsing branch, and abort it if the loop did not contribute to parse a new terminal.

```
1  void atomic run()
2  {
3      f();
4      m.c();
5  }
6
7  void f()
8  {
9      m.a();
10     g();
11 }
12
13 void g()
14 {
15     while (cond)
16         m.b();
17 }
```

$$\mathcal{R} \to \mathcal{F}\ \mathbf{c}$$
$$\mathcal{F} \to \mathbf{a}\ \mathcal{G}$$
$$\mathcal{G} \to A$$
$$A \to B \mid \epsilon$$
$$B \to \mathbf{b}\ A$$

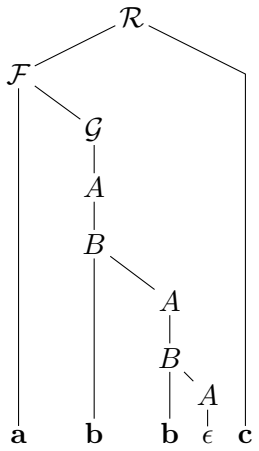


Figure 3.3: Program (left), simplified grammar (center) and parsing tree of **a b b c** (right).

Examples Figure 3.3 shows a program (left), that uses the module `m`. The method `run()` is the entry point of the thread t and is atomic. In the center of the figure we show a simplified version of the G_t grammar. (The G'_t grammar is not shown for the sake of brevity.) The methods `run()`, `f()`, and `g()` are represented in the grammar by the non-terminals $\mathcal{R}$, $\mathcal{F}$, and $\mathcal{G}$ respectively. If we apply Algorithm 1 to this program with the contract $C = \{\mathbf{a}\ \mathbf{b}\ \mathbf{b}\ \mathbf{c}\}$ the resulting parsing tree, denoted by τ (line 6 of Algorithm 1), is represented in Figure 3.3 (right). To verify that all calls represented in this tree are executed atomically, the algorithm determines the lowest common ancestor of **a b b c** in the parsing tree (line 7), in this example $\mathcal{R}$. Since $\mathcal{R}$ is always executed atomically (**`atomic`** keyword), it is ensured that the contract of the module is respected by the client program.

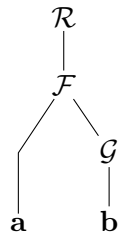
Figure 3.4 exemplifies a situation where the generated grammar is ambiguous. In this case the contract is $C = \{\mathbf{a}\ \mathbf{b}\}$. The figure shows the two distinct ways to parse the word **a b** (right). Both these trees will be obtained by our verification algorithm (line 5 of Algorithm 1). The first tree (top) has $\mathcal{F}$ as the lowest common ancestor of **a b**. Since $\mathcal{F}$ corresponds to the method `f()`, which is executed atomically, this tree respects the contract. On the other hand, the second tree (bottom) has $\mathcal{R}$ as the *lowest common ancestor* of **a b**, corresponding to the execution of the **`else`** branch of method `run()`. This non-terminal ($\mathcal{R}$) does not correspond to an atomically executed method, therefore the contract is not met and a contract violation is detected.

Sub-word Grammar To create a grammar with every sub-word of G we can employ the following conversion. Let $G = (N, \Sigma, P, S)$ be a context-free grammar. Assume, without loss of generality, that G is in Chomsky normal form[2] and does not generate the empty language. This means that every production of the grammar is of the form $A \to B\,C$ or $A \to \alpha$, where A, B, C are non-terminals and α is a terminal. We want to define a

[2]Every context-free grammar can be written in Chomsky normal form.

```
1  void run()
2  {
3      if (...)
4          f();
5      else
6      {
7          m.a();
8          g();
9      }
10 }
11
12 void atomic f()
13 {
14     m.a();
15     g();
16 }
17
18 void atomic g()
19 {
20     m.b();
21 }
```

$$\mathcal{R} \to \mathbf{a}\,\mathcal{G} \mid \mathcal{F}$$
$$\mathcal{F} \to \mathbf{a}\,\mathcal{G}$$
$$\mathcal{G} \to \mathbf{b}$$



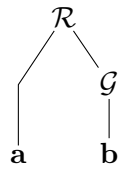


Figure 3.4: Program (left), simplified grammar (center) and parsing trees of $\mathbf{a}\,\mathbf{b}$ (right).

grammar $G' = (N', \Sigma, P', S')$ such that G' generates all sub-word of G and no more:

$$\mathcal{L}(G') = \{w \mid \exists \omega, \omega' \;\; \omega w \omega' \in \mathcal{L}(G)\}.$$

We define $G' = (N', \Sigma, P', S')$ as,

$$\begin{aligned}
A \in N \Leftrightarrow A, A^{<}, A^{>}, A^{<>} &\in N' \\
S' &= S^{<>} \\
A \to B\,C \in P \Leftrightarrow A \to B\,C &\in P' \\
&\wedge A^{<} \to B^{<} \mid B\,C^{<} \in P' \\
&\wedge A^{>} \to C^{>} \mid B^{>}\,C \in P' \\
&\wedge A^{<>} \to B^{>}\,C^{<} \mid B^{<>} \mid C^{<>} \in P' \\
A \to \alpha \in P \Leftrightarrow A \to \alpha &\in P' \\
&\wedge A^{<} \to \epsilon \mid \alpha \in P' \\
&\wedge A^{>} \to \epsilon \mid \alpha \in P' \\
&\wedge A^{<>} \to \epsilon \mid \alpha \in P'.
\end{aligned}$$

For each non-terminal we add three new non-terminals: $A^{<}$, $A^{>}$, and $A^{<>}$. Intuitively these non-terminals represents, respectively, all prefixes, suffixes and sub-words of A. For example the production $A \to B\,C$ in G generates the productions $A^{<>} \to B^{>}\,C^{<} \mid B^{<>} \mid C^{<>}$ in G', which can be read as "the sub-words of A are the suffixes of B concatenated with the prefixes of C; the sub-words of B; and the sub-words of C".

Listing 6 Example of programs that violate the atomicity of $a\ (b\ c)^*\ d$.

```
1  atomic
2  {
3      m.a();
4      m.b();
5      m.c();
6  }
7  atomic
8  {
9    m.d();
10 }
```

```
1  atomic
2  {
3      m.a();
4  }
5  atomic
6  {
7      m.b();
8      m.c();
9      m.d();
10 }
```

As can be seen the number of productions grows only by a constant factor. This does not hold for context-free grammars that have productions with bodies of arbitrary length, but in the Chomsky normal form the bodies of the production are never greater than two. This is also the case of the grammar generated from the control flow graph in Definition 2.

3.5 Extending the Analysis

This section will show how to extended the presented analysis in order to increase the expressivity and precision of the contract. We will augment the contract with the Kleene star, allowing full-fledged regular expression. We will also show how to specify parameter correlation across method call, which greatly extend the expressivity of the contract.

3.5.1 Contracts with the Kleene Star

It would be useful to extend the contract as defined in Definition 1, to allow the Kleene star, making the clauses of the contract full-fledged regular expression. However, Algorithm 1 require that the number of words defined in a contract is finite, which clearly forbids the usage of the Kleene operator. In fact it can be shown that this is not computable, since it is equivalent of knowing if a regular language is a subset of a context-free language, which is undecidable (reduction from the *context-free grammar universality problem*).

However, it is possible to add the Kleene star if we sacrifice precision. It is undesirable to miss anomalies, so we will try to approximate a regular expression e with a set of star-free regular expressions $u = \{u_1, \cdots, u_n\}$ in such a way that every word of $\mathcal{L}(e)$ is sure to have a sub-word in $\mathcal{L}(u)$. Furthermore Algorithm 1 must guarantee that the atomicity of the whole e is ensured by the parts $u_1, \cdots, u_n$.

For the sake of simplicity lets consider only the case where $e = e_1\ e_2^*\ e_3$, where e_1, e_2, e_3 are star-free regular expressions. Even though this is only a special case of arbitrary regular expressions it captures the idea behind this method. Given $e = e_1\ e_2^*\ e_3$ we generate the contract $C = \{e_1\ e_3, e_1\ e_2, e_2\ e_2, e_2\ e_3\}$.

Listing 7 Examples of atomic violation with data dependencies.

```
1 void replace(int o, int n)
2 {
3     if (array.contains(o))
4     {
5         int idx=array.indexOf(o);
6         array.set(idx,n);
7     }
8 }
```

Example Consider $e = a\ (b\ c)^*\ d$. Then we will generate the contract

1. `a d`
2. `a b c`
3. `b c d`
4. `b c b c`

Note that any word of $\mathcal{L}(e)$ will have to be atomic by Algorithm 1: if the $(b\ c)^*$ does not consume any symbols then it is matched by clause 1; if $(b\ c)^*$ consumes $b\ c$ only once then it is matched by clauses 2 and 3; if $(b\ c)^*$ consumes $b\ c$ more than once it will be matched by clauses 2, 3 and 4.

Since there is an overlap in sequences defined in the clauses we force a program to perform every sequence of calls of $\mathcal{L}(e)$ in the same atomic scope. Listing 6 exemplifies this. Both programs do not execute $a\ b\ c\ d \in \mathcal{L}(e)$ atomically. The first one (left) will respect clause 2 but will violate clause 3; conversely the second listing (right) will satisfy clause 3 but will not respect clause 2. Since the clauses overlap the only way for the program to respect them is to execute the whole sequence of calls in the same atomic context.

3.5.2 Contracts with Parameters

Frequently contract clauses can be refined by considering the flow of data across calls to the module. For instance Listing 7 shows a procedure that replaces an item in an array by another. This listing contains two atomicity violations: the element might not exist when `indexOf()` is called; and the index obtained might be outdated when `set()` is executed. Naturally, we can define a clause that forces the atomicity of this sequence of calls as `contains indexOf set`, but this can be substantially refined by explicitly require that a correlation exists between the `indexOf()` and `set()` calls. To do so we extend the contract specification to capture the arguments and return values of the calls, which allows the user to establish the relation of values across calls.

The contract can therefore be extended to accommodate this relations, in this case the clause might be

```
contains(X)  Y=indexOf(X)  set(Y,_).
```

This clause contains variables (`X`, `Y`) that must satisfy unification for the clause to be applicable. The underscore symbol (_) represents a variable that will not be used (and therefore requires no binding). Algorithm 1 can easily be modified to filter out the parsing trees that correspond to calls that do not satisfy the unification required by the clause in question (this can be done in the beginning of the loop of line 6).

The unification relation should not require an exact match between the terms of the program. For example, the calls

```
array.contains(o); idx=array.indexOf(o+1); array.set(idx,n);
```

also implie a data dependency between the first two calls. We should say that A unifies with B if, and only if, the value of A depends on the value of B, which can occur due to value manipulation (data dependency) or control-flow dependency (control dependency). This can be obtained by an information flow analysis, such as presented in [BC85], which can statically infer the variables that influenced the value that a variable hold on a specific part of the program.

This extension of the analysis can be a great advantage for some types of modules. As an example we rewrite the contract for the *Java* standard library class, `java.util.ArrayList`, presented in Section 3.2:

1. `contains(X) indexOf(X)`
2. `X=indexOf(_)` `(remove(X)` | `set(X,_)` | `get(X))`
3. `X=size()` `(remove(X)` | `set(X,_)` | `get(X))`
4. `add(X) indexOf(X).`

This contract captures in detail the relations between calls that may be problematic, and excludes from the contract sequences of calls that does not constitute atomicity violations.

4

Prototype

A prototype was implemented to evaluate our static analysis, described in Chapter 3. This tool verifies *Java* programs using Soot [VRCGHLS99], a Java static analysis framework. This framework directly analyses *Java bytecode*, thus allowing compiled programs to be verified, without requiring access to its source code, which often is not available, specially for third-party libraries. The prototype goes by the name of "gluon" and is publicly available in `https://github.com/trxsys/gluon`[1].

We now give an overview of the several phases gluon performs to verify a program. Gluon starts by searching the class that represents the module under analysis, and extracts its contract. The contract of the module is defined as an annotation of the class representing the module under analysis. The next phase identifies all the possible entry methods of threads that the program may launch. This will be described in Section 4.1. The analysis proceeds by determining which methods are atomically executed in the program, which will be addressed in Section 4.2. After this step is completed the contract can then be verified. For each thread, the grammar that represents its behavior (Definition 2, page 26) is constructed. The words, i.e., sequences of methods call, of the contract are parsed by the grammar, obtaining occurrences of that call sequence in the program. Section 4.3 will discuss the parsing phase of the analysis.

This chapter will describe our prototype and the choices that are implementation-specific and were left open in the analysis defined in Chapter 3.

[1] The `README` file contains instructions on how to use our tool.

4.1 Thread Detection

It is simple to identify all possible thread entry points in a *Java* program. In *Java*, threads can only start running in the `main()` method, and the `run()` method of classes that implement, directly or indirectly, the `java.lang.Runnable` interface [DPL12]. In most cases threads are implemented by extending the `java.lang.Thread` class and implementing the `run()` method. The `java.lang.Thread` class itself extends the `java.lang.Runnable` interface.

Gluon scans every class of the target program and collects all the information about the methods matching the description above, identifying all the entry points of the threads launched by the program.

4.2 Atomically Executed Methods

In our implementation a method can be marked atomic with a *Java* `@Atomic` method annotation. This annotation is in fact used to enforce the atomic execution of methods in the *Deuce Software Transactional Memory Library* [KSF10].

As defined in Section 3.1, we say that a method is *atomically executed* if it is atomic or if the method is always called by atomically executed methods. This way methods that do not employ any synchronization mechanism but are always called by methods that ensure atomic execution are considered atomically executed as well (by directly applying a synchronization mechanism or by transitive application of this definition).

To gather this information our tool traverses the call graph for each thread, keeping track of the context of the methods (executed in an atomic scope or not), and marking the methods as atomically executed if so.

For example, consider the program shown in Listing 4.1 (left) and the corresponding call graph (right). The double circle nodes corresponds to atomic methods (`foo()`) and rectangular nodes denote atomically executed methods (`foo()`, `f()`, and `h()`). Since the `f()` and `h()` methods are not called by an non-atomically executed method they inherit the atomic executed status from `foo()`. If a method is also called in at least one non-atomic context, e.g. `g()`, it will not be marked as atomically executed.

4.3 Parser

Our GLR parser is based on Tomita's parser [Tom87]. Tomita presents a non-deterministic version of the $LR(0)$ parsing algorithm with some optimizations in the representation of the parsing stack that improve the temporal and spacial complexity of the parsing phase.

Our implementation works mostly in the same way as Tomita's parser, and implements one of its proposed optimization: parsing stack sharing. This allows parsing stacks to be shared between different parsing branches that share an initial common history, which greatly improves memory usage. A full Tomita's parser would also implement

```
1  void g() { }
2  void h() { g(); }
3  void f() { h(); }
4  void bar() { g(); }
5
6  void atomic foo()
7  {
8      f();
9      bar();
10 }
11
12 void run()
13 {
14     foo();
15     bar();
16 }
```

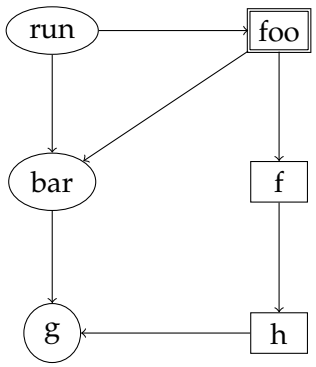


Figure 4.1: Example of atomically executed methods.

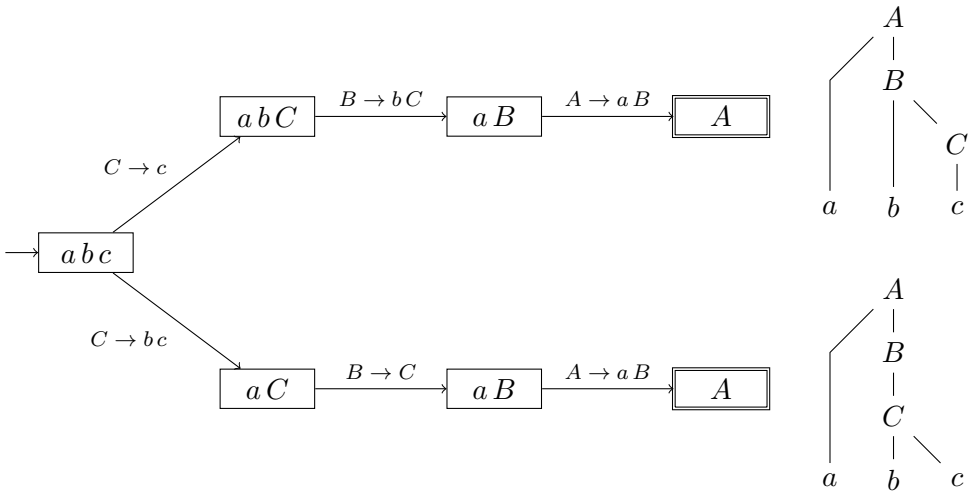


Figure 4.2: Example of parsing a word with two derivations.

parsing stack merging, i.e., when two parsing stacks of different branches reach a similar state they are merged, reducing the number of branches to explore while preserving the divergence in the history of the reductions applied.

To better understand how the parser work lets consider the following grammar.

$$A \to a\,B$$
$$B \to b\,C \mid C$$
$$C \to c \mid b\,c$$

This grammar is ambiguous, since the there are two ways to parse the word $a\,b\,c$. Figure 4.2 exemplifies how our parser handle ambiguities. This figure shows the parsing

of the word $a\,b\,c$ in the grammar above. Initially we start with $a\,b\,c$, and each transition applies a reduction to the word. (We omit shift actions for simplicity.) Accept nodes are represented by a double rectangle. The first action of the parser is ambiguous, since two reductions are applicable ($C \rightarrow c$ and $C \rightarrow b\,c$). This creates two branches to explore both alternative derivations. The figure shows both these derivations and the resulting parsing trees for each branch.

4.4 Optimizations

To achieve a good time performance we implemented a few optimizations. The performance of this prototype will be presented in Chapter 5, Section 5.2.

The time performance optimizations discussed here had a great impact on the analysis efficiency, and some of them are crucial to achieve an acceptable run time. Each of these optimizations reduced the analysis run time, in some cases, by a few orders of magnitude, without sacrificing precision.

Grammar Simplification The size of the grammar can get large, even with few calls to the module. This impacts the performance of the parser, specially because the sub-word grammar introduces many ambiguities in the grammar. The grammar can easily be reduced by applying simple rewrite rules.

When constructing the grammar, most control flow graph nodes will have a single successor. Rule 3.5 of Definition 2 (page 26) will always be applied to nodes of this type, since they represent an instruction that does not call any function. This creates an unnecessarily large number of productions, with many redundant ambiguities in the grammar due to the way the sub-words are added in G'_t, described in Section 3.4.

To avoid exploring redundant parsing branches, we rewrite the grammar to transform productions of the form $A \rightarrow B, B \rightarrow C$ to $A \rightarrow C$. This optimization reduced the analysis time by one order of magnitude, considering the majority of the tests we performed. Even such a simple optimization can have a big impact in the performance of the parser. One of the tests performed (Elevator test) could not be analyzed in a acceptable time prior to this optimization. The cost of this grammar reduction is negligible: it was performed in less than $8\ \mathrm{ms}$ for all tests that will be presented in Chapter 5.

Partial Parsing Parsing an ambiguous grammar can lead to the exploration of redundant parsing trees. This optimization prunes parsing trees, whenever we detect that we are exploring a redundant derivation of a word.

Since the GLR parser builds the derivation tree bottom-up we can be sure to find the lowest common ancestor of the terminals as early as possible. The lowest common ancestors will be the first non-terminal in the tree that covers all the terminals of the parse tree. The lowest common ancestor can be efficiently determined during the parsing phase if we propagate bottom-up the number of terminals that each node of the tree covers.

Figure 4.3: A parsing tree with no redundant loops (left) and with redundant loops (right).

Whenever a lowest common ancestor is identified, we check if we already successfully parsed a tree with that lowest common ancestor, and if so, we prune the current parsing branch.

The correctness of this optimization lies in the fact that a lowest common ancestor should only be reported once per contract word, since it represents the same occurrence of that word in the code. This optimization was able to reduce the run time of the analysis by two orders of magnitude for some of the tests.

Loop Detection As described in Section 3.4, we must handle loops during the parsing. This is not an optimization, it is required to guarantee the termination of the parsing algorithm (loops in the grammar can create parsing trees of infinite height).

To achieve a good performance we should prune parsing branches that generate unproductive loops as soon as possible. Our implementation guarantees that the same non-terminal never appears twice in a parsing tree without contributing to the recognition of a new terminal. To do so we keep track of the number of terminals covered by each of the nodes of the parsing tree. When a reduction creates a non-terminal that is already present in the tree, we verify that the new non-terminal contributed to parse a new terminal. If no new non-terminal was added to the tree this represents an unproductive loop and the current parsing branch is abandoned.

Figure 4.3 exemplifies two parsing trees. The first tree (left) contains a repetition of the non-terminal A, but is not redundant since it contributed to a new terminal (**b**). However the second tree (right) adds a redundant reduction ($A \to A\,\epsilon$) and therefore this parsing branch will be pruned. If this parsing branch was not abandoned it would create an arbitrary height trees by successively application of the reduction $A \to A\,\epsilon$.

In-depth Branch Exploration Tomita's parsing algorithm explores the parsing branches in-breadth. This means that the parser must maintain in memory many parsing branches which are explored in pseudo-parallelism. By exploring the parsing branches in-breadth Tomita's parser guarantees that every finite derivation will eventually be parsed, even if the grammar contains can generate infinite derivations.

We do not need to worry about infinite derivations since our parser detects and prune parsing trees with unproductive loops. This enables us to explores the parsing branches in-depth, improving the memory efficiency of our parser: only one parsing branch is

Listing 8 Example of a program prepared to be analyzed by gluon.

```
@Contract(clauses="a␣b␣c;"
                  +"c␣c;")
class Module
{
    public Module() { }
    public void a() { }
    public void b() { }
    public void c() { }
}

public class Main
{
    private static Module m;

    private static void f() { m.c(); }

    @Atomic
    private static void g() { m.a(); m.b(); f(); }

    public static void main(String[] args)
    {
        m=new Module();

        for (int i=0; i < 10; i++)
            if (i%2 == 0)
                m.a();
            else
                m.b();

        f();
        g();
    }
}
```

maintained in memory, and only after a branch is completely explored (either accepted, rejected, or pruned) does the parser continues to explore another branch.

The worst case memory complexity is the same for both in-depth and in-breadth, but in practice the memory usage is greatly improved by exploring the branches in-depth.

4.5 Using Gluon

To use gluon the class that represents the module to be analyzed must be annotated with its contract. The name of module must be passed to gluon as an argument, since the tool verifies only one module per run.

Listing 8 exemplifies a program prepared to be analyzed by gluon. The class `Module` contains three dummy methods, `a()`, `b()`, and `c()`, and the `@Contract` annotation defines the contract $C = \{\texttt{a b c}, \texttt{c c}\}$.

The analysis output given by gluon is:

```
Checking thread Main.main():

  Verifying word a b c:

      Method: Main.g()
      Calls Location: Main.java:18 Main.java:18 Main.java:15
      Atomic: YES

      Method: Main.main()
      Calls Location: Main.java:26 Main.java:28 Main.java:15
      Atomic: NO

  Verifying word c c:

    No occurrences
```

We can see that gluon detected one entry method of a thread, the starting point of the program `Main.main()`. The words of the contract were then verified to be atomic in that thread.

The sequence `a b c` can be executed in two locations, one beginning in method `g()`, and another in method `main()`, by iterating the **`for`** loop at least twice. The occurrence of the sequence of calls executed in `g()` is identified as atomically executed, and do not constitute a violation of the contract. The second occurrence, called from `main()` is detected as not atomically executed, which is a violation of the contract. Gluon also outputs the exact lines where the calls are performed, making it easy for the programmer to understand and fix the violation of the contract.

The second sequence of calls, `c c`, is never executed, since the method `c()` is only called once by the program. Gluon correctly detects that no occurrences of that word occurs in the program.

5

Evaluation

In this chapter we will evaluate our implementation of the analysis, using the prototype described in Chapter 4. Our evaluation will cover both the correctness and the efficiency of our tool. The results that will be shown not only evaluate the analysis, but also the implementation specific decisions and optimizations of the prototype.

To evaluate the proposed analysis we use a set of 15 tests adapted from the literature [VPG04; AHB03; AHB04; VPG04; LSTD11; Ibm; Pes11; BBA08]. These tests are small *Java* programs that simulate real scenarios where an atomicity violation exist and lead to undesirable behaviors of the programs. We modified the programs to employ a modular design so that the code that cause the atomicity violation were encapsulated in a class. Contracts for those classes were wrote in order to enforce the correct scope of calls from the rest of the program. In some tests the contract contains extra clauses, that did not lead to atomicity violations, but that should nonetheless belong to the contract since they represent potential anomalies.

Appendix A describes each of the tests used in this chapter in detail. All the tests are avaliable in the prototype repository in `https://github.com/trxsys/gluon`[1].

Example To exemplify the types of tests executed, consider Listing 9. This listing shows the original Account test, further covered in Appendix A, Section A.1. We adapted the test so that the `update()` method became part of the client program. This method contains a stale value error, since the current balance is obtained and modified in two separate atomic steps. The balance obtained can be concurrently modified by another thread which causing an atomicity violation.

[1] The `README` file contains instructions on how to run the tests presented in this section.

Listing 9 Original Account test.

```
1  class Account
2  {
3      private int balance;
4
5      ...
6
7      public synchronized int getBalance()
8      {
9          return balance;
10     }
11
12     private synchronized void setBalance(int value)
13     {
14         balance=value;
15     }
16
17     void update(int a)
18     {
19         int tmp=getBalance();
20         tmp=tmp+a;
21         setBalance(tmp);
22     }
23 }
```

We wrote the contract $C = \{$`getBalance setBalance`$\}$, that specifies that a call to `getBalance` followed by a call to `setBalance` should always be executed atomically by the client program of the class `Account`.

The result of our tool was the following:

```
Checking thread Main.main():

  Verifying word getBalance setBalance:

    No occurrences

Checking thread Update.run():

  Verifying word getBalance setBalance:
      Method: Update.update()
      Atomic: NO
```

Gluon correctly identified the two thread entry points of the program, `Main.main()` and `Update.run()`. The first thread (`Main.main()`) does not contain the call sequence `getBalance setBalance`. Our tool detects an instance of that sequence of calls in the second thread (`Update.run()`), in the method `update()` of class `Update`, and warns the user that that sequence is not executed atomically. To fix this atomicity violation the program should make the reported method (`update()`) atomic, thus respecting the contract.

Table 5.1: Validation results.

Test	Threads	Contract Words	Violations	Violations Detected (%)	False Positives
UnderReporting [VPG04]	2	1	1	100%	0
NASA [AHB03]	2	1	1	100%	0
Local [AHB03]	2	2	1	100%	0
StringBuffer [FF04]	1	1	1	100%	0
Jigsaw [VPG04]	2	1	1	100%	0
Coord04 [AHB04]	2	2	1	100%	0
Knight [LSTD11]	2	1	1	100%	0
Coord03 [AHB03]	5	4	1	100%	0
Account [VPG04]	2	4	2	100%	0
Allocate Vector [Ibm]	2	1	1	100%	0
VectorFail [Pes11]	2	2	1	100%	0
Store [Pes11]	3	1	1	100%	0
Connection [BBA08]	2	2	2	100%	0
Arithmetic DB [LSTD11]	2	2	2	100%	0
Elevator [VPG04]	2	2	2	100%	0

5.1 Validation

This section will present the validation results, that evaluate the analysis correctness and precision. The validation results will show that our analysis can indeed identify the violations of the contract.

Table 5.1 summarizes the results of the validation tests. The columns represent the number of threads' starting methods (Threads); the number of words in the contract, i.e., $|C|$ as defined in Algorithm 1 (Contract Words); the number of contract violations known to be present in the client program (Violations); the percentage of violations present in the client program that were detected (Violations Detected (%)); and the number of false positives reported (False Positives).

Our tool was able to *detect all violation of the contract* performed by the client program, so no false negatives occurred, which supports the soundness of the analysis. This is the expected result since the analysis was designed to be conservative, i.e., take into account all possible executions of the program under analysis.

Some tests include additional contract clauses with call sequences that are not contained in the test programs. This shows that, in general, the analysis does not report spurious violations, i.e., false positives. However, it is possible to create situations where false positives occur. For instance, the analysis assumes that a loop may iterate an arbitrary number of times, which makes it consider execution traces that may not be possible.

It was also created a version of each tests where the atomicity violation was fixed, by enclosing the calls in a new atomic region. This corrected version of the tests was also verified and the prototype correctly detected that all contract's call sequences in the client program were now atomically executed, and the program no longer violates the contract. Correcting a program is trivial since the prototype pinpoints the methods that

Table 5.2: Performance results.

Test	Client SLOC	Client CFG Nodes	Grammar Productions	Parsing Branches	Analysis Run Time (ms)
UnderReporting	20	23	56	5	17
NASA	89	67	101	6	21
Local	24	18	46	7	17
StringBuffer	27	24	47	8	19
Jigsaw	100	45	104	9	25
Coord04	35	33	80	12	20
Knight	135	209	343	16	59
Coord03	151	112	187	23	27
Account	42	36	62	34	20
Allocate Vector	183	197	391	64	73
VectorFail	70	51	127	78	31
Store	621	559	892	197	136
Connection	74	69	194	298	51
Arithmetic DB	243	343	732	1048	161
Elevator	268	456	1030	188177	699

must be made atomic, and ensures that the synchronization required has the smallest possible scope (since it is the method that corresponds to the *lowest* common ancestor of the terminals in the parse tree).

5.2 Performance

This section discusses the performance that our tool achieved when the test benchmarks were analyzed. The efficiency of a static analysis tool is an important subject since their performance are often a limiting factor in real-world applications. This section will discuss the time and space efficiency of the analysis, which will show that our tool offers a good performance.

The performance evaluation is presented in Table 5.2. The columns represent the number of lines of the client program (Client SLOC); the number of nodes of the control flow graph of the client program (Client CFG Nodes); the number of grammar productions in G'_t for all threads t (Grammar Productions); the number of branches the parser explored, including failed and pruned branches (Parsing Branches); and the analysis run time (Analysis Run Time). The run time of the analysis includes detecting the threads, identifying of the atomic methods, creating the grammar, creating the parsing table, and the parsing time. This time excludes the Soot framework initialization time, that creates the data structures used to represent the program that are then used by the analysis (see Section 2.3.2). This initialization time was always 35 ± 5 s in the tests presented.

The run time of our analysis always took less than one second for all the tests. The results show that our tool can run efficiently. The Elevator program is the slowest test,

since the parser explores a reasonably large number of branches, but even so, the test took little more that half a second to run. This high number of parse trees is due to the complexity of the control flow of the program, which offer a high amount of distinct control flow paths. In general the parsing phase will dominate the time complexity of the analysis, so the analysis run time will be roughly proportional to the number of parsing branches explored.

The analysis of large programs with a complex control flow might be problematic. To analyze these programs where the number of control flow paths makes the analysis prohibitively expensive we can limit the scope of the analysis to each class of the program. This way we will have smaller grammars to be parsed, but we will fail to detect contract violations that happens across different classes. This can easily be implemented by ignoring method calls to external classes in the grammar generation (Rule 3.3 of Definition 2). In this case a grammar must be generated for each possible entry point of the class, i.e., each public method that is called from outside the class (this can be easily obtained by traversing the call graphs of the threads of the program).

Memory usage is not a problem for our analysis. The asymptotic space complexity is determined by the size of the parsing table and the largest parsing tree. The memory usage will not be affected by the number of parsing trees because our GLR parser explores the parsing branches in-depth instead of in-breadth. (In-depth exploration is possible because we never have infinite height parsing trees due to our detection of unproductive loops.)

6

Conclusion

6.1 Concluding Remarks

Concurrent programming brings various challenges to modern programming languages, that usually offer some kind of abstraction for modular design. Atomicity violations are one of the most common causes of errors in concurrent programs and can be easy to overlook when composing call to services of a modular component.

In this dissertation we present the problem of atomicity violations when using a module, even when their methods are correctly synchronized by some concurrency control mechanism. We propose a solution based on the design by contract methodology. In our approach the developer of the module defines a contract that specifies which sequences of calls to that module should be executed in an atomic manner, therefore avoiding potential atomicity violations.

We introduced an interprocedural static analysis to verify these contracts. The proposed analysis extracts the behavior of the client's program with respect to the module usage, and verifies whether the contract is respected. The extraction of the program's behavior is flexible and can easily be adapted to other analysis that are based on the control flow that the program might take.

We also propose two extensions that can greatly improve the expressiveness of the contract, and how the verification algorithm can be adapted to verify these augmented contracts.

A functional prototype was implemented and is publicly available. We evaluated our approach with a set of known tests and the results suggest that our analysis is sound and precise. Furthermore, the analysis show good performance results, making it suitable for the verification of real-world applications.

6.2 Future Work

The presented work suggests a few future projects that might significantly improve our contributions:

- Evaluate different parsing algorithms in the verification phase. In particular study the CYK and Earley algorithms, and the trade-offs between them and Tomita's parsing algorithm.
- The grammar obtained by Definition 2 can still be further simplified. It would be interesting to evaluate the performance improvements that may arise from further simplification, both in the parsing table creation and in the parsing procedure.
- As it stands the verification algorithm explicitly creates the sub-word grammar from the original grammar. Modifying the parsing algorithm to accept sub-words, from the original grammar, may be a interesting optimization, with significant performance improvements, specially because redundant ambiguities are introduced in the sub-word grammar by this step.
- Try to automatically infer an initial version of the contract from the module's code. This is especially interesting if we consider the expressiveness of contracts with parameters as described in Section 3.5.2.
- The analysis can be improved with points-to analysis. This allows the analysis to take into account different instances of a class as distinct modules, which is helpful in an object-oriented programming language like *Java*.
- Another possible direction that may be worth exploring is using the analysis to automatically enforce the atomicity of the sequences of calls specified by the contract, requiring no explicit synchronization in the source code.
- Implement and evaluate the extensions to the analysis proposed in Section 3.5. In particular adding parameters to the contract greatly enhance the description of potential atomicity violations.
- Further evaluation with real-world applications.

Bibliography

[Aik94] A. Aiken. “Set constraints: Results, applications and future directions”. In: *Principles and Practice of Constraint Programming*. Springer. 1994, pp. 326–335.

[All70] F. E. Allen. “Control flow analysis”. In: *SIGPLAN Not.* 5.7 (July 1970), pp. 1–19. ISSN: 0362-1340. DOI: 10.1145/390013.808479. URL: http://doi.acm.org/10.1145/390013.808479.

[AHB03] C. Artho, K. Havelund, and A. Biere. “High-level data races”. In: *Software Testing, Verification and Reliability* 13.4 (Dec. 2003), pp. 207–227. ISSN: 0960-0833. DOI: 10.1002/stvr.281.

[AHB04] C. Artho, K. Havelund, and A. Biere. “Using block-local atomicity to detect stale-value concurrency errors”. In: *Automated Technology for Verification and Analysis* (2004), pp. 150–164.

[BLS05] M. Barnett, K. R. M. Leino, and W. Schulte. “The Spec# programming system: an overview”. In: *Proceedings of the 2004 international conference on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*. CASSIS’04. Marseille, France: Springer-Verlag, 2005, pp. 49–69. ISBN: 3-540-24287-2, 978-3-540-24287-1. DOI: 10.1007/978-3-540-30569-9_3.

[BBA08] N. E. Beckman, K. Bierhoff, and J. Aldrich. “Verifying correct usage of atomic blocks and typestate”. In: *SIGPLAN Not.* 43.10 (Oct. 2008), pp. 227–244. ISSN: 0362-1340. DOI: 10.1145/1449955.1449783.

[BBGMOO95] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O’Neil, and P. O’Neil. “A critique of ANSI SQL isolation levels”. In: *SIGMOD Rec.* 24.2 (May 1995), pp. 1–10. ISSN: 0163-5808. DOI: 10.1145/568271.223785.

[BC85] J.-F. Bergeretti and B. A. Carré. “Information-flow and data-flow analysis of while-programs”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 7.1 (1985), pp. 37–61.

[BHG87] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1987. ISBN: 0-201-10715-5.

[BL04] M. Burrows and K. Leino. "Finding stale-value errors in concurrent programs". In: *Concurrency and Computation: Practice and Experience* 16.12 (2004), pp. 1161–1172.

[Car96] L. Cardelli. "Type systems". In: *ACM Computing Surveys* 28.1 (1996), pp. 263–264.

[CP07] Y. Cheon and A. Perumandla. "Specifying and checking method call sequences of Java programs". In: *Software Quality Control* 15.1 (Mar. 2007), pp. 7–25. ISSN: 0963-9314. DOI: 10.1007/s11219-006-9001-4.

[Coc70] J. Cocke. "Global common subexpression elimination". In: *ACM Sigplan Notices*. Vol. 5. 7. ACM. 1970, pp. 20–24.

[CC77] P. Cousot and R. Cousot. "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints". In: *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM. 1977, pp. 238–252.

[DV12] R. Demeyer and W. Vanhoof. "A Framework for Verifying the Application-Level Race-Freeness of Concurrent Programs". In: *22nd Workshop on Logic-based Programming Environments (WLPE 2012)*. 2012, p. 10.

[DPL12] R. J. Dias, V. Pessanha, and J. M. Lourenço. "Precise Detection of Atomicity Violations". In: *Hardware and Software: Verification and Testing*. Lecture Notes in Computer Science. HVC 2012 Best Paper Award. Springer Berlin / Heidelberg, Nov. 2012.

[Ear70] J. Earley. "An efficient context-free parsing algorithm". In: *Communications of the ACM* 13.2 (1970), pp. 94–102.

[FF04] C. Flanagan and S. N. Freund. "Atomizer: a dynamic atomicity checker for multithreaded programs". In: *SIGPLAN Not.* 39.1 (Jan. 2004), pp. 256–267. ISSN: 0362-1340. DOI: 10.1145/982962.964023.

[FF10] C. Flanagan and S. N. Freund. "FastTrack: efficient and precise dynamic race detection". In: *Commun. ACM* 53.11 (Nov. 2010), pp. 93–101. ISSN: 0001-0782. DOI: 10.1145/1839676.1839699.

[FFY08] C. Flanagan, S. N. Freund, and J. Yi. "Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs". In: *SIGPLAN Not.* 43.6 (June 2008), pp. 293–303. ISSN: 0362-1340. DOI: 10.1145/1379022.1375618.

[FLLNSS02] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. "Extended static checking for Java". In: *SIGPLAN Not.* 37.5 (May 2002), pp. 234–245. ISSN: 0362-1340. DOI: 10.1145/543552.512558.

[FQ03] C. Flanagan and S. Qadeer. "Types for atomicity". In: *TLDI '03: Proceedings of the 2003 ACM SIGPLAN international workshop on Types in languages design and implementation*. New Orleans, Louisiana, USA: ACM, 2003, pp. 1–12. ISBN: 1-58113-649-8. DOI: http://doi.acm.org/10.1145/604174.604176.

[HW90] M. P. Herlihy and J. M. Wing. "Linearizability: a correctness condition for concurrent objects". In: *ACM Trans. Program. Lang. Syst.* 12.3 (July 1990), pp. 463–492. ISSN: 0164-0925. DOI: 10.1145/78969.78972.

[Hoa69] C. A. R. Hoare. "An axiomatic basis for computer programming". In: *Commun. ACM* 12.10 (Oct. 1969), pp. 576–580. ISSN: 0001-0782. DOI: 10.1145/363235.363259.

[Hur09] C. Hurlin. "Specifying and checking protocols of multithreaded classes". In: *Proceedings of the 2009 ACM symposium on Applied Computing*. SAC '09. Honolulu, Hawaii: ACM, 2009, pp. 587–592. ISBN: 978-1-60558-166-8. DOI: 10.1145/1529282.1529407.

[Ibm] *IBM's Concurrency Testing Repository*.

[JZDLL12] G. Jin, W. Zhang, D. Deng, B. Liblit, and S. Lu. "Automated concurrency-bug fixing". In: *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*. OSDI'12. Hollywood, CA, USA: USENIX Association, 2012, pp. 221–236. ISBN: 978-931971-96-6.

[KU77] J. B. Kam and J. D. Ullman. "Monotone data flow analysis frameworks". In: *Acta Informatica* 7.3 (1977), pp. 305–317.

[Knu65] D. E. Knuth. "On the translation of languages from left to right". In: *Information and control* 8.6 (1965), pp. 607–639.

[KSF10] G. Korland, N. Shavit, and P. Felber. "Deuce: Noninvasive software transactional memory in Java". In: *Transactions on HiPEAC* 5.2 (2010), p. 43.

[Lip75] R. J. Lipton. "Reduction: a method of proving properties of parallel programs". In: *Commun. ACM* 18.12 (Dec. 1975), pp. 717–721. ISSN: 0001-0782. DOI: 10.1145/361227.361234.

[LDZ13] P. Liu, J. Dolby, and C. Zhang. "Finding incorrect compositions of atomicity". In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM. 2013, pp. 158–168.

[LSTD11] J. Lourenço, D. Sousa, B. Teixeira, and R. Dias. "Detecting concurrency anomalies in transactional memory programs". In: *Computer Science and Information Systems/ComSIS* 8.2 (2011), pp. 533–548.

[LPSZ08] S. Lu, S. Park, E. Seo, and Y. Zhou. "Learning from mistakes: a comprehensive study on real world concurrency bug characteristics". In: *SIGPLAN Not.* 43.3 (Mar. 2008), pp. 329–339. ISSN: 0362-1340. DOI: 10.1145/1353536.1346323.

[Mey87] B Meyer. "Eiffel: programming for reusability and extendibility". In: *SIGPLAN Not.* 22.2 (Feb. 1987), pp. 85–94. ISSN: 0362-1340. DOI: 10.1145/24686.24694.

[Mey92] B. Meyer. "Applying "Design by Contract"". In: *Computer* 25.10 (Oct. 1992), pp. 40–51. ISSN: 0018-9162. DOI: 10.1109/2.161279.

[Mey97] B. Meyer. "Object-oriented software construction". In: 2nd. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1997. Chap. 30.7, pp. 990–998. ISBN: 0-13-629155-4.

[NS03] N. Nethercote and J. Seward. "Valgrind: A program supervision framework". In: *Electronic notes in theoretical computer science* 89.2 (2003), pp. 44–66.

[NM92] R. H. B. Netzer and B. P. Miller. "What are race conditions?: Some issues and formalizations". In: *ACM Lett. Program. Lang. Syst.* 1.1 (Mar. 1992), pp. 74–88. ISSN: 1057-4514. DOI: 10.1145/130616.130623.

[NMO09] P. Nienaltowski, B. Meyer, and J. S. Ostroff. "Contracts for concurrency". In: *Form. Asp. Comput.* 21.4 (July 2009), pp. 305–318. ISSN: 0934-5043. DOI: 10.1007/s00165-007-0063-2.

[Pes11] V. Pessanha. "Verificação Prática de Anomalias em Programas de Memória Transaccional". MA thesis. Universidade Nova de Lisboa, 2011.

[RWZ88] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. "Global value numbers and redundant computations". In: *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM. 1988, pp. 12–27.

[Ryd79] B. G. Ryder. "Constructing the call graph of a program". In: *Software Engineering, IEEE Transactions on* 3 (1979), pp. 216–226.

[SBASVY11] O. Shacham, N. Bronson, A. Aiken, M. Sagiv, M. Vechev, and E. Yahav. "Testing atomicity of composed concurrent operations". In: *SIGPLAN Not.* 46.10 (Oct. 2011), pp. 51–64. ISSN: 0362-1340. DOI: 10.1145/2076021.2048073.

[SFL13] D. Sousa, C. Ferreira, and J. Lourenço. "Prevenção de Violações de Atomicidade usando Contratos". In: *INForum 2013*. Universidade de Évora, 2013.

[Tom87] M. Tomita. "An efficient augmented-context-free parsing algorithm". In: *Comput. Linguist.* 13.1-2 (Jan. 1987), pp. 31–46. ISSN: 0891-2017. URL: http://dl.acm.org/citation.cfm?id=26386.26390.

[VRCGHLS99] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. "Soot - a Java bytecode optimization framework". In: *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*. CASCON '99. Mississauga, Ontario, Canada: IBM Press, 1999, pp. 13–.

[VTD06] M. Vaziri, F. Tip, and J. Dolby. "Associating synchronization constraints with data in an object-oriented language". In: *ACM SIGPLAN Notices*. Vol. 41. 1. ACM. 2006, pp. 334–345.

[VPG04] C. Von Praun and T. Gross. "Static detection of atomicity violations in object-oriented programs". In: *Journal of Object Technology* 3.6 (2004), pp. 103–122.

[WS03] L. Wang and S. Stoller. "Run-Time Analysis for Atomicity". In: *Electronic Notes in Theoretical Computer Science* 89.2 (Oct. 2003), pp. 191–209. ISSN: 15710661. DOI: 10.1016/S1571-0661(04)81049-1.

[WZJ11] D. Weeratunge, X. Zhang, and S. Jaganathan. "Accentuating the positive: atomicity inference and enforcement using correct executions". In: *SIGPLAN Not.* 46.10 (Oct. 2011), pp. 19–34. ISSN: 0362-1340. DOI: 10.1145/2076021.2048071.

[You67] D. H. Younger. "Recognition and parsing of context-free languages in time n^3". In: *Information and control* 10.2 (1967), pp. 189–208.

Appendix

This Appendix describes each of the tests use to validate our approach, whose results were presented in Chapter 5.

All the tests are avaliable in the prototype repository, in `https://github.com/trxsys/gluon`.

A.1 Account

This test consists on a simple program that simulates concurrent deposits on a bank account. This test was adapted from [VPG04]. Listing 10 illustrates the `Account` class (right) and the client of that class (left). Both methods offered by the class `Account` are atomic. The client code increments the account balance by performing two calls, one to obtain the current balance and the other to update it.

Atomicity Violation There is a stale value error in the `update()` method. This method obtains the current balance of the account and then updates it, adding `v`. This can lead to an atomicity violation if concurrent threads are performing the same operation, since the balance obtain in line 2 may be outdated when the balance is updated in line 4.

Contract The contract defined for the class `Account` is

1. `getBalance setBalance`
2. `setBalance getBalance`

The first clause captures the atomicity violation present in `update()`. The second clause can also potentially represent a atomicity violation. In this case no sequence of

Listing 10 Account test.

```
void update(int v)
{
    int tmp=account.getBalance();
    tmp=tmp+v;
    account.setBalance(tmp);
}
```

```
@Contract(clauses=
           "getBalance␣setBalance;"
          +"setBalance␣getBalance;")
class Account
{
    int balance;

    @Atomic
    int getBalance()
    {
        return balance;
    }

    @Atomic
    void setBalance(int value)
    {
        balance = value;
    }
}
```

Listing 11 Allocate Vector test.

```
void alloc_block(int i)
{
    int blk=vector.getFreeBlock();
    if (blk != -1)
        vector.markAsAlloced(blk);
}
```

```
@Contract(clauses=
          "getFreeBlock␣markAsAlloced;")
class AllocationVector
{
    ...

    // search and return an index of
    // a free block
    @Atomic
    int getFreeBlock() { ... }

    // mark block i as alloced
    @Atomic
    void markAsAlloced(int i)
    { ... }
}
```

calls that match the second clause is made by the program.

A.2 Allocate Vector

This test contains a vector that is shared across threads. Threads use the vector as a memory pool to allocate blocks of memory. This test was adapted from [Ibm]. Listing 11 shows the `AllocationVector` class (right) and the client of that class (left). Both methods offered by the class `AllocationVector` are atomic. The client code obtains an index of a free block and, unless none exists, mark the block as used.

Atomicity Violation There is a stale value error in the `alloc_block()` method. This method obtains a free block and marks it as used in two atomic operations. This can lead

Listing 12 Arithmetic Database test.

```
@Atomic
int get_key_by_result(int result)
{
    for (Pair<Int,Int> t: res_table)
        if (t.v == result)
            return t.k;
    return -1;
}

void insert_expression(Expression exp)
{
    int expv=exp.eval();
    int fkey=get_key_by_result(expv);

    if (fkey < 0)
    {
        fkey=res_table.get_max_key();
        fkey=(fkey == null) ? 0 : fkey+1;
        res_table.insert(fkey,expv);
    }
    exp_table.insert(exp,foreign_key);
}
```

```
@Contract(clauses
          ="get_max_key␣insert;"
          +"iterator␣insert;")
class Table<K,V>
{
    ...

    @Atomic
    K get_max_key() { ... }

    @Atomic
    void insert(K k, V v)
    { ... }

    @Atomic
    Iterator<Pair<Int,Int>>
        iterator()
    { ... }
}
```

to an atomicity violation: a block being allocated by multiple threads, since the block may not be free when it is marked as used.

Contract The contract defined for the class `AllocateVector` is

1. `getFreeBlock` `markAsAlloced`

A.3 Arithmetic Database

This test simulates a simple database with two tables. The database maintains a set of arithmetic expressions and the corresponding values in two table. The expression table, `exp_table`, associates expressions with an external key that references the result table, `res_table`. The result table keeps the values of those expressions. This test was adapted from [LSTD11]. Listing 12 shows the `Table` class (right) and the client of that class (left). All methods offered by the class `Table` are atomic. The program adds new expressions to the database. If there is already the value of the expression in the result table it only creates a new entry in the expression table, referencing that value. Otherwise it also creates a new entry in the result table with the value of the expression.

Atomicity Violation There are two atomicity violations in the client program:

1. Before adding the expression the client verifies if its value already exists in the result table, if it does not exist it creates a new entry in that table in another atomic

operation. Meanwhile a concurrent thread could also insert that value in the result table leading to a double insertion of the same value.

2. The foreign key is obtained by adding one to the key of highest value in the result table. A concurrent thread might obtain the same foreign key and overwrite this insertion.

Contract The contract defined for the class `Table` is

1. `iterator get_max_key`
2. `get_max_key insert`

A.4 Connection

In this test multiple threads use a connection represented by the class `Connection`. This class also keeps a counter of the number of messages sent through the current socket. This test was adapted from [BBA08]. Listing 13 shows the `Connection` class (right) and the client of that class (left). All methods offered by the class `Connection` are atomic.

Atomicity Violation There are two atomicity violations in the client program:

1. The `disconnect()` method closes the connection and resets the message counter in two atomic phases. A concurrent thread may see an inconsistent state where the socket is closed but the counter is non-zero.

2. The `trySendMessage()` method checks if the socket is open before sending a message. Meanwhile a concurrent thread may close the connection before the message is sent.

Contract The contract defined for the class `Connection` is

1. `isConnected send`
2. `resetSocket resetCounter`

A.5 Coordinates'03

In this test multiple threads access a pair of integers concurrently. This test was adapted from [AHB03]. Listing 14 shows the `Var` class (right) and the client of that class (left). All methods offered by the class `Var` are atomic.

Atomicity Violation The client program obtain the `x` and `y` variables in two atomic operations. A concurrent thread may modify the variables causing an inconsistent values to be read.

Listing 13 Connection test.

```
void disconnect()
{
    connection.resetSocket();
    connection.resetCounter();
}

boolean trySendMsg(String msg)
{
    if (connection.isConnected())
    {
        connection.send(msg);
        return true;
    }

    return false;
}
```

```
@Contract(clauses
  ="isConnected␣send;"
  +"resetSocket␣resetCounter;")
class Connection
{
    int counter;
    Socket socket;

    ...

    @Atomic
    void resetCounter()
    {
        counter=0;
    }

    @Atomic
    void resetSocket()
    {
      socket.close();
    }

    @Atomic
    boolean isConnected()
    {
        return !socket.isClosed();
    }

    @Atomic
    void send(String msg)
    {
        socket.send(msg);
        counter++;
    }
}
```

Contract The contract defined for the class `Vars` is

1. `getX` `getY`
2. `getY` `getX`
3. `getX` `getY`
4. `getY` `getX`

A.6 Coordinates'04

In this test multiple threads access a pair of integers concurrently. This test was adapted from [AHB04]. Listing 15 shows the `Coord` class (right) and the client of that class (left). All methods offered by the class `Coord` are atomic.

Listing 14 Coordinates'03 test.

```
1  void run()
2  {
3      int x=v.getX();
4      int y=v.getY();
5      use(x,y);
6  }
```

```
1  @Contract(clauses="getX␣getY;"
2                    +"getY␣getX;"
3                    +"setX␣setY;"
4                    +"setY␣setX;")
5  class Vars
6  {
7      int x = 0;
8      int y = 0;
9
10     ...
11
12     @Atomic
13     int getX() { return x; }
14
15     @Atomic
16     void setX(int x) { this.x=x; }
17
18     @Atomic
19     int getY() { return y; }
20
21     @Atomic
22     void setY(int y) { this.y=y; }
23 }
```

Atomicity Violation The client program resets `x` and `y` variables in two atomic operations. A concurrent thread may read the variables in the middle of the reset, reading an inconsistent state.

Contract The contract defined for the class `Coord` is

1. `resetX` `resetY`
2. `resetY` `resetX`

A.7 Elevator

In this test multiple threads control a set of elevators concurrently. This test was adapted from [VPG04]. Listing 16 shows the `Controls` class (right) and the client of that class (left). All methods offered by the class `Controls` are atomic. The threads periodically verify if the elevator needs to go up or down with the methods `checkUp()`/`checkDown()` and move the elevator up or down accordingly with `claimUp()`/`claimDown()`.

Atomicity Violation The client program may verify that the elevator need to go up, but a concurrent thread may order the elevator to go up. This will order the elevator to go up twice.

Contract The contract defined for the class `Controls` is

Listing 15 Coordinates'04 test.

```
void reset()
{
    coord.resetX();
    coord.resetY();
}
```

```
@Contract(clauses="resetX␣resetY;"
                  +"resetY␣resetX;")
class Coord
{
    int x=0;
    int y=0;

    ...

    @Atomic
    void swap()
    {
        int oldX=x;
        x=y;
        y=oldX;
    }

    @Atomic
    void resetX()
    {
        x=0;
    }

    @Atomic
    void resetY()
    {
        y=0;
    }
}
```

1. `checkUp claimUp`
2. `checkDown claimDown`

A.8 Jigsaw

In this test concurrent threads concurrently access a pool of resources. This test was adapted from [VPG04]. Listing 17 shows the `ResourceStoreManager` class (right) and the client of that class (left). All methods offered by the class `ResourceStoreManager` are atomic. The method `loadResourceStore()` returns a resource store given a resource store manager an a resource.

Atomicity Violation The method `loadResourceStore()` verifies that the resource store manager is not closed before executing a lookup. A concurrent thread can close the resource store manager thus and an invalid resource store is added to a closed store manager and is returned.

Contract The contract defined for the class `ResourceStoreManager` is

Listing 16 Elevator test.

```
boolean up(Controls c,
           String lift,
           int floor)
{
    if (c.checkUp(floor))
    {
        c.claimUp(floor);
        return true;
    }

    return false;
}
```

```
@Contract(clauses="checkUp␣claimUp;"
                  +"checkDown␣claimDown;")
class Controls
{
    Floor[] floors;

    ...

    @Atomic
    void claimUp(int floor)
    {
        ...
    }

    @Atomic
    boolean checkUp(int floor)
    {
        ...
    }
}
```

1. checkClosed lookupEntry

A.9 Knight

In this test concurrent threads concurrently try to find a best solution to shortest path in a chess board, restricted to the legal moves a knight can do. This test was adapted from [LSTD11]. Listing 18 shows the KnightMoves class (right) and the client of that class (left). All methods offered by the class KnightMoves are atomic. The threads concurrently perform a depth first search in the board and the class KnightMoves keeps the minimum number of moves used to reach a place in the chess board.

Atomicity Violation The method check__solution() verifies that it has found a better solution than the current one. If this is the case the new solution is updated but since the operations are executed in two atomic operations the solution verified may be outdated and the current solution is overwritten.

Contract The contract defined for the class KnightMoves is

1. get_solution get_solution

A.10 Local

This is a very simple test where the module represents an integer. Multiple threads increment that integer concurrently. This test was adapted from [AHB03]. Listing 19 shows

Listing 17 Jigsaw test.

```
ResourceStore
loadResourceStore(
  ResourceStoreManager r,
  Object k)
{
    Entry e;

    if (r.checkClosed())
        return null;

    e=r.lookupEntry(k);
    return e.getStore();
}
```

```
@Contract(clauses
          ="checkClosed lookupEntry;")
class ResourceStoreManager
{
    boolean closed=false;
    Map entries=new HashMap();

    ...

    @Atomic
    boolean checkClosed()
    {
        return closed;
    }

    @Atomic
    Entry lookupEntry(Object key)
    {
        Entry e=entries.get(key);
        if (e == null)
        {
            e=new Entry(key);
            entries.put(key, e);
            entries=null;
        }
        return e;
    }

    @Atomic
    void shutdown()
    {
        entries.clear();
        closed=true;
    }
}
```

the `Cell` class (right) and the client of that class (left). All methods offered by the class `Cell` are atomic.

Atomicity Violation The method `inc()` obtains the value of the cell, increments it and stores the new value. This value can be outdated when it is stored thus annulling a concurrent modification.

Contract The contract defined for the class `Cell` is

1. `getValue setValue`
2. `setValue getValue`

Listing 18 Knight test.

```
void check_solution(
  KnightMoves km,
  Point p,
  int moves)
{
    if (km.get_solution(p) > moves)
        km.set_solution(p,moves);
}
```

```
@Contract(clauses
          ="get_solution␣set_solution;")
public class KnightMoves
{
    // solution[x][y] is the current
    // best way to reach (x,y)
    private int solution[][];

    ...

    @Atomic
    int get_solution(Point p)
    {
        return solution[p.x][p.y];
    }

    @Atomic
    void set_solution(Point p, int m)
    {
        solution[p.x][p.y]=m;
    }
}
```

A.11 NASA

In this test multiple threads concurrently execute tasks. Each tasks requires some properties to be acquired and must hold during its execution. To acquire a property the task sets its value and marks it as achieved. Concurrent events may change or remove properties. This test was adapted from [AHB03]. Listing 20 shows the `TaskManager` class (right) and the client of that class (left). All methods offered by the class `TaskManager` are atomic.

Atomicity Violation The method `run_task()` sets the value of a property and acquire that property, marking it as achieved. Another event may trigger a concurrent task that removes that property, but the current task will still mark the property as achieved.

Contract The contract defined for the class `TaskManager` is

1. `setValue setArchieved`

A.12 Store

This test simulates a store were threads dispatch orders from a queue. Concurrent threads check the queue for new orders, dispatch them, and add the sale to a log. This test was adapted from [Pes11]. Listing 21 shows the `Store` class (right) and the client of that class (left). All methods offered by the class `Store` are atomic.

Listing 19 Local test.

```
void inc()
{
    int tmp;
    tmp=x.getValue();
    tmp++;
    x.setValue(tmp);
}
```

```
@Contract(clauses
          ="getValue setValue;"
          +"setValue getValue;")
class Cell
{
    int n=0;

    @Atomic
    int getValue()
    {
        return n;
    }

    @Atomic
    void setValue(int x)
    {
        n=x;
    }
}
```

Atomicity Violation Threads check the order queue for pending orders. If so, that order is treated and added to the log of sales. A concurrent thread may treat that order leaving no remaining orders pending. In this case a `null` will be returned by `treatOrder()` and added to the sales log.

Contract The contract defined for the class `Store` is

1. `hasOrders` `treatOrder`

A.13 String Buffer

This test is based on `java.lang.StringBuffer`. This test was adapted from [FF04]. Listing 22 shows the `StringBuffer` class (right) and the client of that class (left). All methods offered by the class `StringBuffer` are atomic.

Atomicity Violation The method `append()` copies the `other` buffer to an auxiliary `char` array. This copies exactly `len` characters from the buffer, but this value may be outdated if a concurrent thread modifies `other`.

Contract The contract defined for the class `StringBuffer` is

1. `length` `getChars`

Listing 20 NASA test.

```
void run_task()
{
    tm.setValue(v,n);
    tm.setAchieved(v,n);

    ...
}
```

```
@Contract(clauses="setValue setAchieved;")
class TaskManager
{
    Cell[] table;

    ...

    @Atomic
    void setValue(Object v, int n)
    {
        table[n].value=v;
    }

    @Atomic
    void setAchieved(Object v, int n)
    {
        table[n].achieved=true;
    }
}
```

Listing 21 Store test.

```
while (true)
{
    waitForClients();

    if (store.hasOrders())
    {
        Order o=store.treatOrder();
        addLog(o);
    }
}
```

```
@Contract(clauses
          ="hasOrders treatOrder;")
class Store
{
    @Atomic
    boolean hasOrders()
    {
        ...
    }

    @Atomic
    Order treatOrder()
    {
      ...
    }
}
```

A.14 Under-Reporting

This is a very simple test where the module represents a counter. Multiple threads increment that counter concurrently. This test was adapted from [FF04]. Listing 23 shows the `Counter` class (right) and the client of that class (left). All methods offered by the class `Counter` are atomic.

Atomicity Violation The method **`double`**`()` obtains the current value of the counter and adds it to itself in two atomic operations. The value read from the counter may be outdated if a concurrent thread modifies it, breaking the semantics of the method.

Contract The contract defined for the class `Counter` is

Listing 22 String Buffer test.

```
void append(StringBuffer t,
            StringBuffer other)
{
    int len=other.length();
    char[] value=new char[len];

    other.getChars(0,len,value,0);

    ...
}
```

```
@Contract(clauses="length getChars;")
class StringBuffer
{
    ...

    @Atomic
    int length()
    {
        ...
    }

    @Atomic
    void getChars(int srcBegin,
                  int srcEnd,
                  char[] dst,
                  int dstBegin)
    {
        ...
    }
}
```

Listing 23 Under-Reporting test.

```
void double()
{
    int i=c.inc(0);
    c.inc(i);
}
```

```
@Contract(clauses="inc inc;")
class Counter
{
    int i;

    ...

    @Atomic
    int inc(int a)
    {
        i+=a;
        return i;
    }
}
```

1. `inc` `inc`

A.15 Vector Fail

In this test the module maintains a pair of integers. Concurrent threads update the pair such that one of the elements is twice of the other. This test was adapted from [FF04]. Listing 24 shows the `Vector` class (right) and the client of that class (left). All methods offered by the class `Vector` are atomic.

Atomicity Violation The method `setVector()` sets the pair to $(v, 2v)$, updating each component in an atomic operation. A concurrent thread may modify the pair in the

Listing 24 Under-Reporting test.

```
void setVector(Vector vector, int v)
{
      vector.setElements(v, 2*v);

      int max=vector.getMax();
      int min=vector.getMin();

      assert max == 2*min;
}
```

```
@Contract(clauses = "getMax␣getMin;"
                    +"getMin␣getMax;")
class Vector
{
    int first;
    int second;

    ...

    @Atomic
    int getMax()
    {
        ...
    }

    @Atomic
    int getMin()
    {
        ...
    }
}
```

middle of the update, breaking the invariant.

Contract The contract defined for the class `Vector` is

1. `getMax getMin`
2. `getMin getMax`