Preventing Atomicity Violations with Contracts

Universidade NOVA de Lisboa — Technical Report

Diogo G. Sousa João M. Lourenço Carla Ferreira Ricardo Dias

***Abstract*—Software developers are expected to synchronize concurrent accesses to shared regions of memory with some mutual exclusion primitive that ensures atomicity properties to a sequence of program statements. This approach prevents data races but may fail to provide all necessary correctness properties, potentially leaving atomicity violations unaddressed. The composition of atomic operations may cause these atomicity violations if there is a correlation between them. To avoid this the operations should be grouped in a larger atomic scope to ensure their correct execution. This problem is especially common when using services provided by third party packages or modules, since the programmer may fail to identify which services are correlated. Developers of a module can specifying a contract that identifies which methods are correlated and must be executed in the same atomic scope, reducing the program errors due to atomicity violations. If a contract is complete and the program respects it then the program is safe from atomicity violations with respect to the contract's module. This work presents a static analysis based methodology to verify such contracts.**

I. Introduction

The encapsulation of a set of functionalities as services of a software module offers strong advantages in software development, since it promotes the reuse of code and ease maintenance efforts. If the programmer is unacquainted with the implementation details of a particular services, she may fail to identify correlations that exist across the services, such as data and code dependencies, leading to a inappropriate usage. This is particularly relevant in a concurrent setting, where it is hard to account for all the possible interleavings between threads and the effects of these interleaved calls to the module.

One of the requirements for the correct behavior of a module is to respect its *protocol*, which defines the legal sequences of invocations to its methods. For instance, a module that offers an abstraction to deal with files typically will demand that the programmer start by calling the method `open()`, followed by an arbitrary number of `read()` or `write()` operations, and concluding with a call to `close()`. A program that does not follow this protocol is incorrect and should be fixed. A way to force the programmer to conform to such a protocol is to use the design by contract methodology [1] to design contracts that specifies the module usage protocol. In this setting, the contract not only serves as useful documentation, but can also be automatically verified, ensuring the client's program obeys the module's protocol [2], [3].

The development of concurrent programs brings new challenges on how to define the protocol of a module. Not only it is important to respect the module's protocol, but is also necessary to guarantee the atomic execution of sequences

```
void schedule() {
    Task t=taskQueue.next();

    if (t.isReady())
        t.run();
}
```

Fig. 1. Program with an atomicity violation.

of calls that are susceptible of causing atomicity violations. These atomicity violations are possible, even when the module's methods are protected by some concurrency control mechanism. Figure 1 shows part of a program that schedules tasks. The `schedule()` method gets a task, verifies if it is ready to run, and execute it if so. This program contains a potential atomicity violation since the method may execute a task that is not marked as ready. This can happens when another thread concurrently schedules the same task, despite the fact the methods of `Task` are atomic. In this case the `isReady()` and `run()` methods should be executed in the same atomic context to avoid atomicity violations. Atomicity violations are one of the most common source of bugs in concurrent programming [4] and are particularly susceptible to occur when composing calls to a module, as the developer may not be aware of the implementation and internal state of the module.

In this paper we propose to extend module usage protocols with a specification of the sequences of calls that should be executed atomically. We will also present an efficient static analysis to verify these protocols.

The contributions of this paper can be summarized as:

1) A static analysis methodology to extract the behavior of a program with respect to the sequence of calls it may execute;
2) A static analysis to verify if a program conforms to a module's contract, hence that the module's correlated services are correctly invoked in the scope of an atomic region.

The remaining of this paper is organized as follows. In Section II we provide a specification and the semantics for the contract. Section III contains the general methodology of the analysis. Section IV presents the phase of the analysis that extracts the behavior of the client program while Section V shows how to verify a contract based on the extracted information. Section VIII follows with the presentation and discussion of the results of our experimental validation. The related work

is presented in Section IX, and we conclude with the final remarks in Section X.

II. Contract Specification

The contract of a module must specify which sequences of calls of its non-private methods must be executed atomically, as to avoid atomicity violations in the module's client program. In the spirit of the *programming by contract* methodology, we assume the definition of the contract, including the identification of the sequences of methods that 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 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 (implicit) concatenation 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 using the alternative operator. We chose to offer the alternative operator so the programmer can group similar scenarios under the same clause. Our verification analysis assumes the contract defines a finite number of call sequences.

Example Consider the array implementation offered by *Java* standard library, `java.util.ArrayList`. For simplicity we will only consider the methods `add(obj)`, `contains(obj)`, `indexOf(obj)`, `get(idx)`, `set(idx, 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 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, in addition, the position of the 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 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 (set | remove)`, 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.

III. Methodology

The proposed analysis verifies statically if a client program complies with the contract of a given module, as defined in Section II. 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:

1) Determine the entry methods of each thread launched by the program.
2) Determine which of the program's methods are atomically executed. We say that a method is *atomically executed* if it is atomic[1] or if the method is always called by atomically executed methods.
3) Extract the behavior of each of the program's threads with respect to the usage of the module under analysis.
4) For each thread, verify that its usage of the module respects the contract as defined in Section II.

In Section IV we introduce the algorithm that extracts the program's behavior with respect to the module's usage. Section V defines the methodology that verifies whether the extracted behavior complies to the contract.

IV. Extracting the Behavior of a Program

The behavior of the program with respect to the module usage can be seen as the individual behavior of any thread 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 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) [5]. The CFG of the thread t represents every possible path 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 $\mathcal{L}(G)$.)

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

A context-free grammar is especially suitable to capture the structure of the CFG since it easily captures the call relations between methods that cannot be captured by a weaker class of languages such as regular languages. The first example bellow will show how this is done. Another advantage of using context-free grammars (as opposed to another static analysis technique) is that we can use efficient algorithms for parsing to explore the language it represents.

Definition 2 (Program's Thread Behavior Grammar)**.** The grammar $G_t = (N, \Sigma, P, S)$ is build 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 1–5;
- 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 respecting the rules 1–5. 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.

$$\text{if } \alpha : [\![\text{entry}]\!], \quad \{\mathcal{F} \to \alpha\} \cup \{\alpha \to \beta \mid \beta \in succ(\alpha)\} \subset P \tag{1}$$

$$\text{if } \alpha : [\![\text{mod.h()}]\!], \quad \{\alpha \to \mathbf{h}\,\beta \mid \beta \in succ(\alpha)\} \subset P \tag{2}$$

$$\text{if } \alpha : [\![\text{g()}]\!], \quad \{\alpha \to \mathcal{G}\,\beta \mid \beta \in succ(\alpha)\} \subset P \quad \text{where } \mathcal{G} \text{ represents } \texttt{g()} \tag{3}$$

$$\text{if } \alpha : [\![\text{return}]\!], \quad \{\alpha \to \epsilon\} \subset P \tag{4}$$

$$\text{if } \alpha : [\![\text{otherwise}]\!], \quad \{\alpha \to \beta \mid \beta \in succ(\alpha)\} \subset P \tag{5}$$

No more productions belong to P.

Rules 1–5 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 not related with the module's usage. For example, if $\mathbf{f}\ \mathbf{g} \in \mathcal{L}(G_t)$ then the thread t may invoke, method `f()`, followed by `g()`.

Rule 1 adds a production that relates the non-terminal $\mathcal{F}$, representing method `f()`, to the entry node of the CFG of `f()`. Calls to the module under analysis are recorded in the grammar by the Rule 2. Rule 3 handles calls to another method `g()` of the client program (method `g()` will have its non-terminal $\mathcal{G}$ added by Rule 1). The return point of a method adds an ϵ production to the grammar (Rule 4). All others types of CFG nodes are handled uniformly, preserving the CFG structure by making them reducible to the successor non-terminals (Rule 5). 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. Therefore we need 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 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 2 (left) shows a program that consists of two methods that call each other mutually. Method `f()` is the entry point of the thread and the module under analysis is represented by object `m`. The control flow graphs of these methods are shown in Figure 2 (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:

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

A second example, shown in Figure 3, 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. We have $G_2 = (N_2, \Sigma_2, P_2, S_2)$, with

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

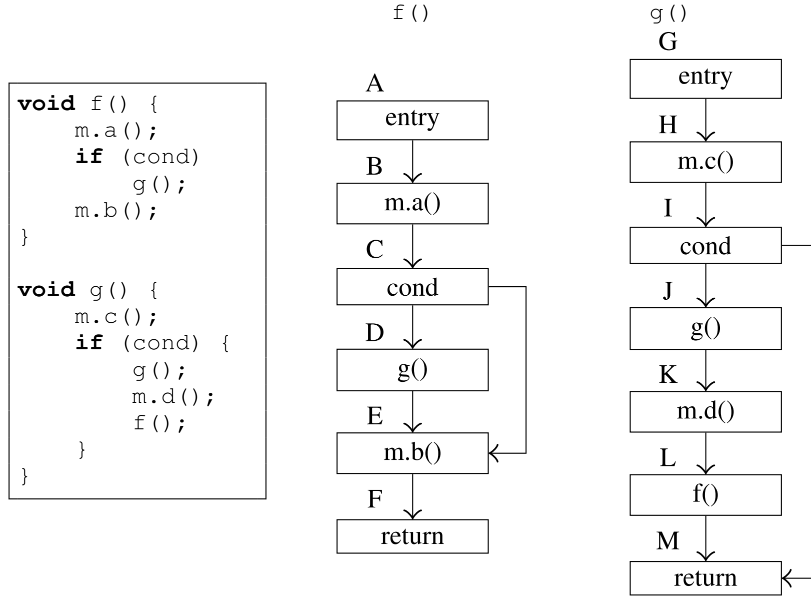

Fig. 2. Program with recursive calls using the module m (left) and respective CFG (right).

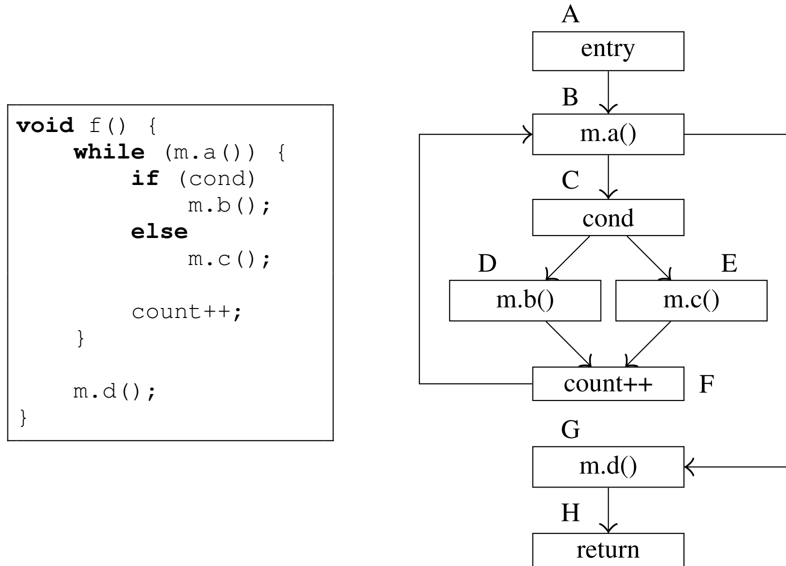

Fig. 3. Program using the module m (left) and respective CFG (right).

The set of productions P_2 is,

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

V. Contract Verification

The verification of a contract must ensure 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.

The idea of the algorithm is to generate a grammar the captures the behavior of each thread with respect to the module usage. Any sequence of the calls contained in the contract can then be found by parsing the word (i.e. the sequence of calls) in that grammar. This will create a parsing tree for each place where the thread can execute that sequence of calls. The parsing tree can then be inspected to determine the atomicity of the sequence of calls discovered.

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 the these sequences are parsed in the grammar G'_t

Algorithm 1 Contract verification algorithm.

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

1: **for** $t \in$ threads(P) **do**
2: $G_t \leftarrow$ make_grammar(t)
3: $G'_t \leftarrow$ subword_grammar(G_t)
4: **for** $w \in C$ **do**
5: $T \leftarrow$ parse(G'_t, w)
6: **for** $\tau \in T$ **do**
7: $N \leftarrow$ lowest_common_ancestor(τ, w)
8: **if** $\neg$run_atomically(N) **then**
9: **return** ERROR
10: **return** OK

(line 5) that 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, i.e., if we have a program `m.a(); m.b(); m.c(); m.d();` we are able to verify the word **b c** by parsing it in G'_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 location of each methods 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 the lowest common ancestor is always executed atomically (line 8) to make sure the whole sequence of calls is executed under the same atomic context. Since it is the *lowest* common ancestor we are sure to require the 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 expressivity offered by context-free languages. For this reason it is not sufficient to use the $LR(\cdot)$ parsing algorithm [6], 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 explore all the ambiguities that can generate different derivation trees for a word. A GLR parser was introduced by Tomita in [7]. Tomita presents a non-deterministic versions 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.

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 ($A \Rightarrow \cdots \Rightarrow A$), which often occur in G_t (every loop in the control flow graph will yield a corresponding loop in the grammar). For this

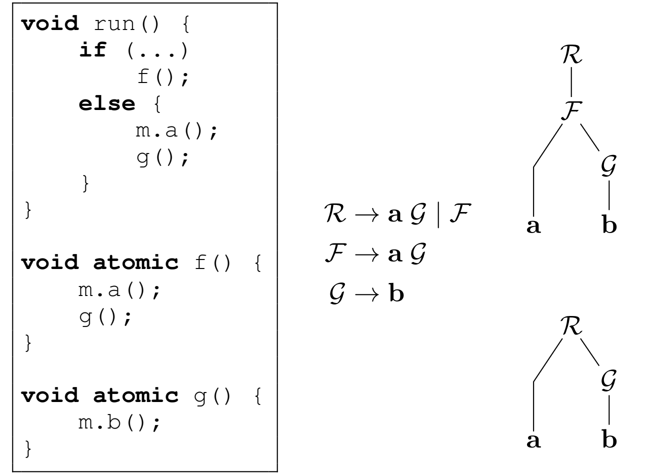

Fig. 5. Program (left), simplified grammar (center) and parsing tree of **a b** (right).

reason the parsing algorithm must detect and prune parsing branches that will lead to redundant loops, ensuring a finite number of 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 that loop did not contribute to parse a new terminal.

Examples Figure 4 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 shown a simplified version of the G_t grammar. (The G'_t grammar is not shown for the sake of brevity.) The `run()`, `f()`, and `g()` methods 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 $C = \{\mathbf{a\ b\ b\ c}\}$ the resulting parsing tree, denoted by τ (line 6 of Algorithm 1), is represented in Figure 4 (right). To verify 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 complies to the contract of the module.

Figure 5 exemplifies a situation where the generated grammar is ambiguous. In this case the contract is $C = \{\mathbf{a\ b}\}$. 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, so this tree respects the contract. 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.

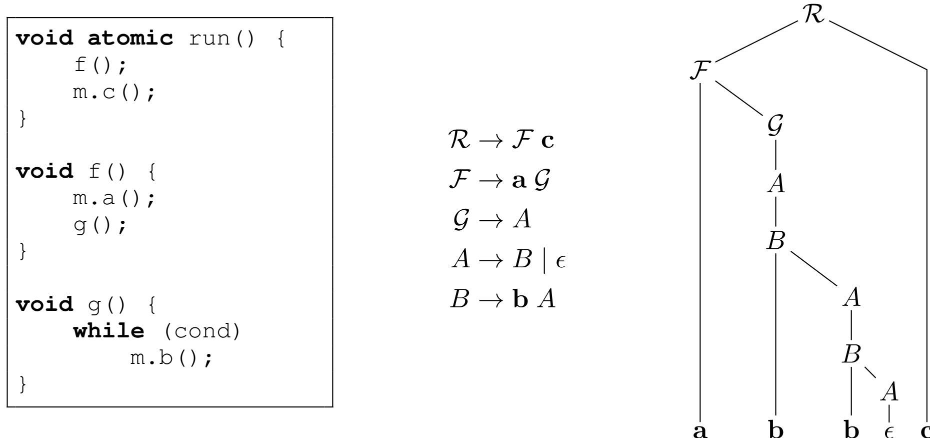

Fig. 4. Program (left), simplified grammar (center) and parsing tree of **a b b c** (right).

Algorithm 2 Contract verification algorithm with points-to information.

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

1: **for** $t \in$ threads(P) **do**
2: **for** $a \in$ mod_alloc_sites(t) **do**
3: $G_{t_a} \leftarrow$ make_grammar(t, a)
4: $G'_{t_a} \leftarrow$ subword_grammar(G_{t_a})
5: **for** $w \in C$ **do**
6: $T \leftarrow$ parse(G'_{t_a}, w)
7: **for** $\tau \in T$ **do**
8: $N \leftarrow$ lowest_common_ancestor(τ, w)
9: **if** $\neg$run_atomically(N) **then**
10: **return** ERROR
11: **return** OK

VI. Analysis with Points-to

In an object-oriented programming language the module is often represented as an object, in which case we should differentiate the instances of the class of the module. This section explains how the analysis is extended to handle multiple instances of the module by using *points-to* information.

To extend the analysis to points-to a different grammar is generated for each allocation site of the module. Each allocation site represents an instance of the module, and the verification algorithm verifies the contract words for each allocation site and thread (whereas the previous algorithm verified the contract words for each thread). The new algorithm is shown in Algorithm 2. It generated the grammar G_{t_a} for a thread t and module instance a. This grammar can be seen as the behavior of thread t with respect to the module instance a, ignoring every other instance of the module.

To generate the grammar G_{t_a} we adapt the Definition 2 to only take into account the instance a. The grammar generation is extended in the following way:

Definition 3 (Program's Thread Behavior Grammar with points-to)**.** The grammar $G_t = (N, \Sigma, P, S)$ is build from the CFG of the client's program thread t and an object allocation site a, which represents an instance of the module.

We define N, Σ, P and S in the same way as Definition 2.

The rules remain the same, except for rule 2, which becomes:

$$\text{if } \alpha : [\![\text{mod.h()}]\!] \text{ and mod can only point to } a \quad (6)$$
$$\{\alpha \to \mathbf{h}\,\beta \mid \beta \in succ(\alpha)\} \subset P$$
$$\text{if } \alpha : [\![\text{mod.h()}]\!] \text{ and mod can point to } a \quad (7)$$
$$\{\alpha \to \mathbf{h}\,\beta \mid \beta \in succ(\alpha)\} \subset P$$
$$\{\alpha \to \beta \mid \beta \in succ(\alpha)\} \subset P$$
$$\text{if } \alpha : [\![\text{mod.h()}]\!] \text{ and mod cannot point to } a \quad (8)$$
$$\{\alpha \to \beta \mid \beta \in succ(\alpha)\} \subset P$$

Here we use the points-to information to generate the grammar, and we should consider the places where a variable can point-to. If it may point-to our instance a or another instance we consider both possibilities in the Rule 7 of Definition 3.

VII. Prototype

A prototype was implemented to evaluate our methodology. This tool analyses *Java* programs using Soot [8], a Java static analysis framework. This framework directly analyses *Java bytecode*, allowing us to analyse a compiled program, without requiring access to its source code. In our implementation a method can be marked atomic with a *Java* annotation. The contract is also defined as an annotation of the class representing the module under analysis. The prototype is available in https://github.com/trxsys/gluon.

A. Optimizations

To achieve a reasonable time performance we implemented a few optimizations. Some of these optimizations reduced the analysis run time by a few orders of magnitude in some cases, without sacrificing precision.

A simple optimization was applied to the grammar to reduce its size. When constructing the grammar, most control flow

graph nodes will have a single successor. Rule 5 (Definition 2) will always be applied to these kind of nodes, since they represent an instruction that does not call any function. This creates redundant ambiguities in the grammar due to the multiple control flow paths that never use the module under analysis. To avoid exploration of redundant parsing branches we rewrite the grammar to transform productions of the form $A \rightarrow \beta B \delta$, $B \rightarrow \alpha$ to $A \rightarrow \beta \alpha \delta$, if no other rule with head B exists. For example, an **`if else`** that do not use the module will create the productions $A \rightarrow B$, $A \rightarrow C$, $B \rightarrow D$, $C \rightarrow D$. This transformation will reduce it to $A \rightarrow D$, leaving no ambiguity for the parser to explore here. This optimization reduced the analysis time by at least one order of magnitude considering the majority of the tests we performed. For instance, the Elevator test could not be analyzed in a reasonable time prior to this optimization.

Another optimization was applied during the parsing phase. 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 covering all the terminals of the parse tree. This is easily determined if we propagate, bottom-up, the number of terminals each node of the tree covers. Whenever a lowest common ancestor is determined we do not need further parsing and can immediately verify if the corresponding calls are in the same atomic context. This avoids completing the rest of the tree which can contain ambiguities, therefore a possibly large number of new branches is avoided.

Another key aspect of the parsing algorithm implementation is the loop detection. To achieve a good performance we should prune parsing branches that generated unproductive loop as soon as possible. Our implementation guarantees the same non-terminal never appears twice in a parsing tree without contributing to the recognition of a new terminal.

To achieve a better performance we also do not explicitly compute the subword grammar (G'_t). We have modified our GLR parser to parse subwords as described in [9]. This greatly improves the parser performance because constructing G'_t introduces many irrelevant ambiguities the parser had to explore.

The Table I show how much the of the optimizations improve the analysis performance. These results are build from an test made to stress the performance of gluon but is consistent with real applications. The *Improvement* column show how much of an improvement that particular optimization contributes to the analysis. The *Stop Parsing at LCA* cause an improvement that we were not able to measure since the test was unable to complete in reasonable time.

B. Class Scope Mode

Gluon normally analyzes the entire program, taking into account any sequence of calls that can spread across the whole program (as long as they are consecutive calls to a module). However this is infeasible for very large programs so, for these programs, we ran the analysis with for each class, ignoring calls to other classes. This will detect contract violations where the control flow does not escape the class, which is reasonable since code locality indicates a stronger correlations between calls.

```
void replace(int o, int n)
{
    if (array.contains(o))
    {
        int idx=array.indexOf(o);
        array.set(idx,n);
    }
}
```

Fig. 6. Examples of atomic violation with data dependencies.

This mode of operation can be useful to analyze large programs as they might have very complex control flow graphs and thus are infeasible to analyze with the scope of the whole program.

In this mode the grammar is build for each class instead of each thread. The methods of the class will create non-terminals $\mathcal{F}_1, \cdots, \mathcal{F}_n$, just as before. The only change in creating this grammar is that we create the productions $S \rightarrow \mathcal{F}_1 \mid \cdots \mid \mathcal{F}_n$ as the starting production of the grammar (S being the initial symbol). This means that we consider the execution of all methods of the class being analyzed.

C. Contracts with Parameters

Frequently contract clauses can be refined by considering the flow of data across calls to the module. For instance Listing 6 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.

In our implementation we require a exact match between the terms of the program to satisfy the unification, since it was sufficient for most scenarios. It can however be advantageous to generalize the unification relation. For example, the calls

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

TABLE I
OPTIMIZATION IMPROVEMENTS.

Optimization	Improvement
Grammar Simplification	428%
Stop Parsing at LCA	?
Subword Parser	3%

```
array.set(idx,n);
```

also imply 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 [10], 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 II:

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.

D. ICFinder

ICFinder tries to infer automatically what a module is, and incorrect compositions of pairs of calls to modules.

Two patterns are used to detect potencial atomicity violations in method calls compositions:

- USE: Detects stale value errors. This pattern detects data or control flow dependencies between two calls to the module.
- COMP: If a call to method `a()` dominates `b()` and `b()` post-dominates `a()` in some place, that is captured by this pattern. This means that, for each piece of code involving two calls to the module (`a()` and `b()`), if `a()` is always executed before `b()` and `b()` is always executed after `a()`, it is a COMP violation.

Both this patterns are extremely broad and contain many false positives. To deal with this the authors filter this results with a dynamic analysis that only consider violations as defined in [11]. This analysis assumes that the notion of *atomic set* was correctly inferred by ICFinder.

VIII. VALIDATION

To validate the proposed analysis we analyzed a few real-world programs (Tomcat, Lucene, Derby, OpenJMS and Cassandra) as well as small programs known to contain atomicity violations. These small programs were adapted from the literature [12]–[18] and are typically used to evaluate atomicity violation detection techniques. We modified these small programs to employ a modular design and we wrote contracts to enforce the correct atomic scope of calls to that module. Some additional clauses were added that may represent atomicity violations in the context of the module usage, even if the program do not violate those clauses.

For the large benchmarks analyzed we aimed to discover new, unknown, atomicity violations. To do so we had to create contracts in an automated manner, since the code base was too large. To automate the generation of contracts we employ a very simplistic approach that tries to infer the contract's clauses based on what is already synchronized in the code. This idea is that most sequences of calls that should be atomic was correctly used *somewhere*. Having this in mind we look for sequences of calls done to a module that are used atomically at least two points of the program. If a sequence of calls is done atomically in two places of the code that might indicate that these calls are correlated and should be atomic. We used these sequences as our contracts, after manually filtering a few irrelevant contracts. This is a very simple way to generate contracts, which should ideally be written by the module's developer to capture common cases of atomicity violations, so we can expect the contracts to be more fine-tuned to better target atomicity violations if the contracts are part of the regular project development.

Since these programs load classes dynamically it is impossible to obtain complete points-to information, so we are pessimistic and assumed every module instance could be referenced by any variable that are type-compatible. We also used the class scope mode described in Section VII-B because it would be impractical to analyze such large programs with the scope of the whole program. This restrictions did not apply to the small programs analyzed.

Table II summarizes the results that validate the correctness of our approach. The table contains both the macro benchmarks (above) and the micro benchmarks (bellow). The columns represent the number of clauses of the contract (Clauses); the number of violations of those clauses (Contract Violations); the number of false positives, i.e. sequences of calls that in fact the program will never execute (False Positives); the number of potential atomicity violations, i.e. atomicity violations that could happen *if* the object was concurrently accessed by multiple threads (Potential AV); the number of real atomicity violations that can in fact occur and compromise the correct execution of the program (Real AV); the number lines of code of the benchmark (SLOC); and the

TABLE II
VALIDATION RESULTS.

Benchmark	Clauses	Contract Violations	False Positives	Potential AV	Real AV	SLOC	Time (s)	ICFinder Static	ICFinder Final
Allocate Vector [16]	1	1	0	0	1	183	0.120	-	-
Coord03 [13]	4	1	0	0	1	151	0.093	-	-
Coord04 [14]	2	1	0	0	1	35	0.039	-	-
Jigsaw [12]	1	1	0	0	1	100	0.044	121	2
Local [13]	2	1	0	0	1	24	0.033	-	-
Knight [15]	1	1	0	0	1	135	0.219	-	-
NASA [13]	1	1	0	0	1	89	0.035	-	-
Store [17]	1	1	0	0	1	621	0.090	-	-
StringBuffer [14]	1	1	0	0	1	27	0.032	-	-
UnderReporting [12]	1	1	0	0	1	20	0.029	-	-
VectorFail [17]	2	1	0	0	1	70	0.048	-	-
Account [12]	4	2	0	0	2	42	0.041	-	-
Arithmetic DB [15]	2	2	0	0	2	243	0.272	-	-
Connection [18]	2	2	0	0	2	74	0.058	-	-
Elevator [12]	2	2	0	0	2	268	0.333	-	-
OpenJMS 0.7	6	54	10	28	4	163K	148	126	15
Tomcat 6.0	9	157	16	47	3	239K	3070	365	12
Cassandra 2.0	1	60	24	15	2	192K	246	-	-
Derby 10.10	1	19	5	7	1	793K	522	122	16
Lucene 4.6	3	136	21	76	0	478K	151	391	2

time it took for the analysis to complete (the analysis run time excludes the Soot initialization time, which were always less than 179s per run).

Our tool was able to detect all violation of the contract by the client program in the microbenchmarks, so no false negatives occurred, which supports the soundness of the analysis. Since some tests include additional contract clauses with call sequences not present in the test programs we also show that, in general, the analysis does not detect spurious violations, i.e., false positives.[2] A corrected version of each test was also verified and the prototype correctly detected that all contract's call sequences in the client program were now atomically executed. Correcting a program is trivial since the prototype pinpoints the methods that must be made atomic, and ensures the synchronization required has the finest possible scope, since it is the method that corresponds to the *lowest* common ancestor of the terminals in the parse tree.

The large benchmarks show that gluon can be applied to large scale programs with good results. Even with a simple automated contract generation we were able to detect 10 atomicity violations in real-world programs. Some of those bugs where reported (Tomcat, Derby, Cassandra). The false positives incorrectly reported by gluon were all due to conservative points-to information, since the program loads and calls classes and methods dynamically (leading to an incomplete points-to graph).

ICFinder [19] uses a static analysis to detect two types of common incorrect composition patterns. This is then filtered with a dynamic analysis. Of the atomicity violations detected by gluon none of them was captured by ICFinder, since they failed to match the definition of the patterns.

The performance results show our tool can run efficiently. For larger programs we have to use class scope mode, sacrificing precision for performance, but we still can capture interesting contract violations. The performance of the analysis depends greatly on the number of branches the parser explores. This high number of parsing branches is due to the complexity of the control flow of the program, offering a huge 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 proportional to the number of explored parsing branches. Memory usage is not a problem for the analysis, since the asymptotic space complexity is determined by the size of the parsing table and the largest parsing tree. Memory usage is not 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.

[2]In these tests no false positives were detected. However it is possible to create situations where false positives occur. For instance, the analysis assumes a loop may iterate an arbitrary number of times, which makes it consider execution traces that may not be possible.

IX. RELATED WORK

The methodology of design by contract was introduced by Meyer [1] as a technique to write robust code, based on contracts between programs and objects. In this context, a contract specifies the necessary conditions the program must met in order to call the object's methods, whose semantics is ensured if those pre-conditions are met.

Cheon et al. proposes the use of contracts to specify protocols for accessing objects [2]. These contracts use regular expressions to describe the sequences of calls that can be executed for a given *Java* object. The authors present a dynamic analysis for the verification of the contracts. This contrasts to our analysis which statically validates the contracts. Beckman et al. introduce a methodology based on *typestate* that statically verifies if a protocol of an object is respected [18]. This approach requires the programmer to explicitly *unpack* objects before it can be used. Hurlin [3] extends the work of Cheon to support protocols in concurrent scenarios. The protocol specification is extended with operators that allow methods to be executed concurrently, and pre-conditions that have to be satisfied before the execution of a method. This analysis is statically verified by a theorem prover. Theorem

proving, in general, is very limited since automated theorem proving tend to be inefficient.

Peng Liu et al. developed a way to detect atomicity violations caused by method composition [19], much like the ones we describe in this article. They define two patterns that are likely to cause atomicity violations, one capturing stale value errors and the other one by trying to infer a correlation between method calls by analyzing the control flow graph (if `a()` is executed before `b()` and `b()` is executed after `a()`). This patterns are captured statically and then filtered with a dynamic analysis.

Many works can be found about atomicity violations. Artho et al. in [13] define the notion of *high-level data races*, that characterize sequences of atomic operations that should be executed atomically to avoid atomicity violations. The definition of high-level data races do not totally capture the violations that may occur in a program. Praun and Gross [12] extend Artho's approach to detect potential anomalies in the execution of methods of an object and increase the precision of the analysis by distinguish between read and write accesses to variables shared between multiple threads. An additional refinement to the notion of high-level data races was introduced by Pessanha in [20], relaxing the properties defined by Artho, which results in a higher precision of the analysis. Farchi et al. [21] propose a methodology to detect atomicity violations in the usage of modules based on the definition of high-level data races. Another common type of atomicity violations that arise when sequencing several atomic operations are *stale value errors*. This type of anomaly is characterized by the usage of values obtained atomically across several atomic operations. These values can be outdated and compromise the correct execution of the program. Various analysis were developed to detect these types of anomalies [14], [20], [22]. Several other analysis to verify atomicity violations can be found in the literature, based on access patterns to shared variables [11], [15], type systems [23], semantic invariants [24], and other specific methodologies [25]–[27].

X. Concluding Remarks

In this paper we present the problem of atomicity violations when using a module, even when their methods are individually synchronized by some concurrency control mechanism. We propose a solution based on the design by contract methodology. Our contracts define which call sequences to a module should be executed in an atomic manner.

We introduce a 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.

A prototype was implemented and the experimental results shows the analysis is highly precise and can run efficiently on real-world programs.

XI. Acknowledgments

This work was partially supported by the Portuguese national research project PTDC/EIA-EIA/113613/2009 (Synergy-VM).

References

[1] B. Meyer, "Applying "design by contract"," *Computer*, vol. 25, no. 10, pp. 40–51, Oct. 1992.

[2] Y. Cheon and A. Perumandla, "Specifying and checking method call sequences of java programs," *Software Quality Control*, vol. 15, no. 1, pp. 7–25, Mar. 2007.

[3] C. Hurlin, "Specifying and checking protocols of multithreaded classes," in *Proceedings of the 2009 ACM symposium on Applied Computing*, ser. SAC '09. New York, NY, USA: ACM, 2009, pp. 587–592.

[4] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: a comprehensive study on real world concurrency bug characteristics," *SIGPLAN Not.*, vol. 43, no. 3, pp. 329–339, Mar. 2008.

[5] F. E. Allen, "Control flow analysis," *SIGPLAN Not.*, vol. 5, no. 7, pp. 1–19, Jul. 1970. [Online]. Available: http://doi.acm.org/10.1145/390013.808479

[6] D. E. Knuth, "On the translation of languages from left to right," *Information and control*, vol. 8, no. 6, pp. 607–639, 1965.

[7] M. Tomita, "An efficient augmented-context-free parsing algorithm," *Comput. Linguist.*, vol. 13, no. 1-2, pp. 31–46, Jan. 1987. [Online]. Available: http://dl.acm.org/citation.cfm?id=26386.26390

[8] 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*, ser. CASCON '99. IBM Press, 1999, pp. 13–.

[9] J. Rekers and W. Koorn, "Substring parsing for arbitrary context-free grammars," *ACM Sigplan Notices*, vol. 26, no. 5, pp. 59–66, 1991.

[10] J.-F. Bergeretti and B. A. Carré, "Information-flow and data-flow analysis of while-programs," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 7, no. 1, pp. 37–61, 1985.

[11] M. Vaziri, F. Tip, and J. Dolby, "Associating synchronization constraints with data in an object-oriented language," in *ACM SIGPLAN Notices*, vol. 41, no. 1. ACM, 2006, pp. 334–345.

[12] C. Von Praun and T. Gross, "Static detection of atomicity violations in object-oriented programs," *Journal of Object Technology*, vol. 3, no. 6, pp. 103–122, 2004.

[13] C. Artho, K. Havelund, and A. Biere, "High-level data races," *Software Testing, Verification and Reliability*, vol. 13, no. 4, pp. 207–227, Dec. 2003.

[14] ——, "Using block-local atomicity to detect stale-value concurrency errors," *Automated Technology for Verification and Analysis*, pp. 150–164, 2004.

[15] J. Lourenço, D. Sousa, B. Teixeira, and R. Dias, "Detecting concurrency anomalies in transactional memory programs," *Computer Science and Information Systems/ComSIS*, vol. 8, no. 2, pp. 533–548, 2011.

[16] "IBM's Concurrency Testing Repository."

[17] V. Pessanha, "Verificao prtica de anomalias em programas de memria transaccional (Practical verification of anomalies in transactional memory programs)," Master's thesis, Universidade Nova de Lisboa, 2011.

[18] N. E. Beckman, K. Bierhoff, and J. Aldrich, "Verifying correct usage of atomic blocks and typestate," *SIGPLAN Not.*, vol. 43, no. 10, pp. 227–244, Oct. 2008.

[19] 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.

[20] R. J. Dias, V. Pessanha, and J. M. Loureno, "Precise detection of atomicity violations," in *Hardware and Software: Verification and Testing*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, Nov. 2012, hVC 2012 Best Paper Award.

[21] E. Farchi, I. Segall, J. a. M. Lourenço, and D. Sousa, "Using program closures to make an application programming interface (api) implementation thread safe," in *Proceedings of the 2012 Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*, ser. PADTAD 2012. New York, NY, USA: ACM, 2012, pp. 18–24.

[22] M. Burrows and K. Leino, "Finding stale-value errors in concurrent programs," *Concurrency and Computation: Practice and Experience*, vol. 16, no. 12, pp. 1161–1172, 2004.

[23] L. Caires and J. a. C. Seco, "The type discipline of behavioral separation," *SIGPLAN Not.*, vol. 48, no. 1, pp. 275–286, Jan. 2013. [Online]. Available: http://doi.acm.org/10.1145/2480359.2429103

[24] 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.

[25] C. Flanagan and S. N. Freund, “Atomizer: a dynamic atomicity checker for multithreaded programs,” *SIGPLAN Not.*, vol. 39, no. 1, pp. 256–267, Jan. 2004.

[26] C. Flanagan, S. N. Freund, and J. Yi, “Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs,” *SIGPLAN Not.*, vol. 43, no. 6, pp. 293–303, Jun. 2008.

[27] C. Flanagan and S. N. Freund, “FastTrack: efficient and precise dynamic race detection,” *Commun. ACM*, vol. 53, no. 11, pp. 93–101, Nov. 2010.