10th IEEE International Conference on Software Testing, Verification and Validation

Verifying Concurrent Programs using Contracts

Ricardo J. Dias[†], Carla Ferreira[†], Jan Fiedor[*], João M. Lourenço[†], Aleš Smrčka[*], Diogo G. Sousa[†], Tomáš Vojnar[*]
[*] FIT, Brno University of Technology [†] FCT, Universidade Nova de Lisboa

***Abstract*—The central notion of this paper is that of *contracts for concurrency*, allowing one to capture the expected atomicity of sequences of method or service calls in a concurrent program. The contracts may be either extracted automatically from the source code, or provided by developers of libraries or software modules to reflect their expected usage in a concurrent setting. We start by extending the so-far considered notion of contracts for concurrency in several ways, improving their expressiveness and enhancing their applicability in practice. Then, we propose two complementary analyses—a static and a dynamic one—to verify programs against the extended contracts. We have implemented both approaches and present promising experimental results from their application on various programs, including real-world ones where our approach unveiled previously unknown errors.**

I. Introduction

The divide-and-conquer strategy is frequently applied to the development of large software products where the whole application is divided into interacting software modules, collaboratively developed by multiple teams. Objects in object-oriented programming languages are an example of such modules. Accessing the services provided by a software module requires one to follow a protocol that includes: (i) the syntax of the service, i.e., the name of the service and the type of its input and output parameters (including return values); (ii) the semantics of the service, i.e., the expected behavior of the service for a given set of input parameters; and (iii) the service access restrictions, e.g., the domain of the valid values for each parameter, dependency relations between services, atomicity requirements for execution in a concurrent setting, etc.

Violating the protocol of a service may cause all sorts of misbehaviors—from subtle, perhaps admissible, but wrong results to fault-stop fails, such as exceptions and segmentation faults. Compilers take good care of Aspect (i) of the protocol, i.e., syntax validation. Aspect (ii), service semantics, although not verified by compilers, is usually at least documented. Aspect (iii), service access restrictions, is usually not verified by compilers nor documented, which results in a deep dependency on programmers' clairvoyance on the usage of the services—in particular, when concurrency issues are involved.

Reports [9], [26], [35] emphasize that it often takes more than a month to fix a concurrency-related error and that nearly 70 % of the fixes are buggy when first released. In this paper, we aim at reducing this problem by addressing Aspect (iii) from the list above, i.e., service access restrictions, for the context of *concurrent (multi-threaded)* programs. In particular, we address restrictions of using services provided by software modules in a concurrent setting with the aim of avoiding atomicity violations and similar concurrency-related errors.

We build on the concept of *contracts for concurrency* [16], [37], a particular case of a software protocol, allowing one to enumerate sequences of public methods of a module that are required to be executed atomically. We extend the previously proposed notion of contracts for concurrency by allowing them to reflect both the *data flow* between the methods (in that a sequence of methods calls only needs to be atomic if they manipulate the same data) and the *contextual information* (in that a sequence of methods calls needs not be atomic wrt all other sequences of methods but only some of them).

Moreover, we propose novel methods for both *static* and *dynamic validation* of such protocols in client programs. While the static approach can analyse all possible executions of a program at once, it may not be feasible for larger programs and often reports false alarms. The dynamic analysis is more scalable and suffers much less from false alarms, but it is restricted to concrete program executions and to the errors that can be deduced from them. The static analysis is based on *grammars* and *parsing trees* while the dynamic uses the *happens-before relation* and *vector clocks* optimized for contract validation. We implemented both approaches in publicly available prototype tools and obtained promising experimental results with both of them, including discovery of previously unknown errors in large real-world programs.

The rest of the paper is organised as follows. In Section II, we present the notion of contracts for concurrency and extend them to consider the data flow and/or the contextual information of method calls. In Sections III and IV, we describe our static and dynamic contract validation methods, respectively, together with results of experiments with them. Section V summarises related works, and Section VI concludes the paper.

II. Contracts for Concurrency

A *contract for concurrency* [16], [37] (or simply *contract* herein) is a protocol for accessing public services of a module, i.e., the methods of its public API, expressing which of the methods are correlated and should be executed in the same atomic context (wrt its API usage) if applied on the same computational object. Therefore, a program that conforms to a contract is guaranteed to be safe from atomicity violations.

A. Basic Contracts

In [16], [37], a *contract* is formally defined as follows. Let $\Sigma_{\mathbb{M}}$ be a set of all public method names (the API) of a software module (or library). A *contract* is a set $\mathbb{R}$ of *clauses* where each clause $\varrho \in \mathbb{R}$ is a regular expression over $\Sigma_{\mathbb{M}}$. A contract violation occurs if any of the sequences represented by the contract clauses is interleaved with an execution of methods from $\Sigma_{\mathbb{M}}$ over the same object.

Example. Consider the `java.util.ArrayList` implementation of a resizable array of the Java standard library, and, for simplicity, take the following subset of the available methods: `add(obj)`, `contains(obj)`, `indexOf(obj)`, `get(idx)`, `set(idx, obj)`, `remove(idx)`, and `size()`. The below clauses belong to the contract for the `ArrayList` library:

(ϱ_1) `contains indexOf`
(ϱ_2) `indexOf ( set | remove | get )`
(ϱ_3) `size ( remove | set | get )`
(ϱ_4) `add ( get | indexOf )`

Clause ϱ_1 states that the execution of `contains()` followed by `indexOf()` should be atomic. Otherwise, the program may confirm the existence of an object in the array but fail to obtain its index as a concurrent thread can, e.g., remove the object. Clause ϱ_2 represents a similar scenario where the index of an object is obtained and then the index is used to modify the object. Without atomicity, a concurrent change of the array may shift the position of the object and cause malfunction. Clause ϱ_3 deals with programs that verify whether a given index is in a valid range (e.g., `index < size()`) and then access the array. To ensure `size()` is still valid when accessing the array, the calls must execute atomically. Clause ϱ_4 represents a scenario where an object is added to the array and then the program tries to obtain information about it by querying the array. Without atomicity, the object may no longer exist or its position in the array may have shifted.

Another relevant clause in the contract of `ArrayList` is:

(ϱ_5) `contains indexOf ( set | remove )`

However, the contract's semantic already enforces this clause since it results from the composition of clauses ϱ_1 and ϱ_2.

Still, it turns out that the above definition of contracts for concurrency is sometimes quite restrictive and can classify valid concurrent programs as unsafe. Hence, in Sections II-B and II-C, we propose two extensions that improve the expressiveness of contracts: one extends them with parameters, making it possible to consider the data flow between method calls; and the other adds contextual information that restricts the situations in which atomicity shall be enforced.

B. Extending Contracts with Parameters

Figure 1 illustrates a situation where basic contracts may be too restrictive. It shows a procedure that replaces item `a` in an array by item `b`. The procedure contains two atomicity violations: (i) item `a` does not need to exist anymore when `indexOf` is called; and (ii) the index obtained may be outdated when `set` is executed. A basic contract of Section II-A could cover this situation by a clause (ϱ_6) `contains indexOf set`. However, the given sequence needs to be executed atomically only if `contains` and `indexOf` have the same argument, and the result of `indexOf` is used as the first argument of `set`.

To express in a contract how the flow of data influences the dependencies between methods, we extend the contract specification by considering *method call parameters* and *return values*, expressed as *meta-variables*. Then, if a contract should be enforced only if the same object appears as an argument or as the return value of multiple calls in the given call sequence, we may express that by using the same meta-variable at the position of all the concerned parameters and/or return values.

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

Fig. 1: Example of atomicity violation with data dependencies.

Clause ϱ_6 may then be refined as follows—in particular, note the repeated use of meta-variables `X`/`Y`, requiring the same objects o_1/o_2 to appear at the positions of `X`/`Y`, resp.: (ϱ'_6) `contains(X)  Y = indexOf(X)  set(Y,_)`. Here, the underscore is a free meta-variable that imposes no restrictions.

Example. With the above extension, it is possible to refine the contract for `java.util.ArrayList` as follows:

(ϱ'_1) `contains(X) indexOf(X)`
(ϱ'_2) `X = indexOf(_) ( remove(X) | set(X,_) | get(X) )`
(ϱ'_3) `X = size() ( remove(X) | set(X,_) | get(X) )`
(ϱ'_4) `add(X) ( get(X) | indexOf(X) )`

This contract captures in detail the dependencies between method calls, expressing the relations that are problematic, excluding those that do not constitute atomicity violations.

C. Extending Contracts with Spoilers

Interleaving a sequence of calls listed in a contract clause with some methods of the given API may lead to an atomicity violation, while this is not the case for other methods. This is, however, not reflected in the basic contracts. For example, the clause `contains indexOf` states that this sequence of calls must always be executed atomically (wrt methods of the given module), regardless of which methods the other threads are executing. Interleaving a thread executing this sequence with another one is thus a contract violation regardless of whether the other thread executes `remove` or `get`, not distinguishing that the former is harmful while the latter not.

To cope with the above, we propose to augment contracts with *contextual information*, allowing one to express in which context the contract clauses shall be enforced. For that, each clause of the basic contract (now called a *target*) will be coupled with a set of *spoilers* that restrict its application. A spoiler represents a set of sequences of methods that may violate its target. Client programs must then ensure that each target is executed atomically wrt its spoilers, whenever executed on the same object. For the target clause `contains indexOf`, a possible spoiler is `remove`, and the extended clause would be: `contains indexOf` $\leftsquigarrow$ `remove`.

Formally, as before, let $\mathbb{R}$ be the set of *target* clauses where each target $\varrho \in \mathbb{R}$ is a regular expression over $\Sigma_{\mathbb{M}}$. Let $\mathbb{S}$ be the set of *spoilers* where each spoiler $\sigma \in \mathbb{S}$ is a regular expression over $\Sigma_{\mathbb{M}}$. We also define the alphabets $\Sigma_{\mathbb{R}} \subseteq \Sigma_{\mathbb{M}}$ and $\Sigma_{\mathbb{S}} \subseteq \Sigma_{\mathbb{M}}$ for the methods used in the targets or spoilers, respectively.

A *contract* is then a relation $\mathbb{C} \subseteq \mathbb{R} \times \mathbb{S}$ which defines for each target the spoilers that may cause atomicity violations. Note that one target may be violated by more than one spoiler and also one spoiler may violate more than one target. A contract is violated if any sequence represented by a target $\varrho \in \mathbb{R}$ executed on the same object o is fully interleaved with an execution of the sequence representing its spoiler $\sigma \in \mathbb{C}(\varrho)$ on the object o. A target sequence r is fully interleaved by a spoiler sequence s if the execution of r starts before the execution of s and the execution of s ends before that of r.[1]

Example. The basic contract for `java.util.ArrayList` with spoilers extending it with contextual information is below:

(ϱ_1'') `contains indexOf` $\leftrightsquigarrow$ `remove`
(ϱ_2'') `indexOf (remove | set | get)` $\leftrightsquigarrow$ `remove | add | set`
(ϱ_3'') `size (remove | set | get)` $\leftrightsquigarrow$ `remove`
(ϱ_4'') `add indexOf` $\leftrightsquigarrow$ `remove | set`

This contract explicitly captures which interferences are harmful and which interleavings shall be forbidden. All other interleavings, not captured by spoilers, are considered safe.

Finally, the extension of contracts with spoilers can be combined with the extension with parameters, allowing one to define fine-grained atomicity requirements for the methods of a module. This can be illustrated by the below clause:

`contains(X) indexOf(X)` $\leftrightsquigarrow$ `remove(_)`.

This clause requires sequences of `contains` and `indexOf` to be executed atomically but only when executed over the same object, when dealing with the same item `X`, and only wrt concurrent execution of `remove`. This captures the fact that any concurrent removal may lead to an atomicity violation, by either removing object `X` or by altering its position in the array. Note that `add` is not a spoiler since it does not interfere with the position of `X` as elements are added to the end of the array.

III. Static Contract Validation

We now propose a static approach for verifying whether a client program complies with the contract of a given module. We consider contracts in the form defined in Section II-B but restricted to star-free regular expressions.

Our approach is based on checking whether threads launched by the client program always execute atomically any sequence of calls expressed by contract clauses, and it has the following phases: (1) Extract the behaviour of each of the client program's threads wrt the usage of the module under analysis. (2) Determine which of the program's methods are *atomically executed*. We say that a method is *atomically executed* if it explicitly applies a concurrency control mechanism to enforce atomicity, or if the method is always called by other atomically executed methods. (3) For each thread, verify that its usage of the module respects the contract.

The next section covers Phase 1 by introducing an algorithm that extracts the program's behaviour wrt the module's methods. Section III-B covers Phases 2 and 3 by proposing an algorithm that verifies whether the extracted behaviour complies to the contract.

A. Extracting the Behaviour of a Program

The behaviour of a program can be seen as the join of the individual behaviours of all threads the program may launch. To extract the usage of a module by a thread, we start by extracting its control flow graph (CFG) [1] from the source code. From the CFG of a thread t, it is then simple to construct a context-free grammar G_t such that if there is an execution path of t that runs a sequence of method calls, then that sequence is a word of the language represented by G_t.

Context-free grammars were chosen to describe the structure of CFGs since they can capture the call relations between methods that cannot be captured by weaker classes of languages. Moreover, an advantage of using context-free grammars (compared with other static analysis techniques) is that we can use efficient parsing algorithms within the analysis.

Definition 1. The CFG of the client's program thread t is encoded by the grammar $G_t = (N, \Sigma_{\mathbb{M}}, P, I)$ where N is the set of nodes of the CFG (non-terminals), $\Sigma_{\mathbb{M}}$ is the set of the identifiers of the public methods $\mathbb{M}$ of the module under analysis (terminals), I is the initial non-terminal defined as the entry method, and P is the set of productions defined below.

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 a method, *mod.h()*—a call to method `h()` of the module *mod* under analysis, *g()*—a call to method `g()` of the client program, and *return*—the return point of a method. The function $succ : N \to \mathcal{P}(N)$ is used to obtain the successors of a given node N in the CFG. The entry method for a thread is determined by looking for extensions of the `Thread` class or implementations of the `Runnable interface`. The set P of productions is then defined by Rules 1–5 as follows (no other productions belong to P):

$$\text{for } \alpha : [\![\text{entry}]\!],\ \{\mathcal{F} \to \alpha\} \cup \{\alpha \to \beta \mid \beta \in succ(\alpha)\} \subset P \quad (1)$$
$$\text{for } \alpha : [\![\text{mod.h()}]\!],\ \{\alpha \to \mathbf{h}\beta \mid \beta \in succ(\alpha)\} \subset P \quad (2)$$
$$\text{for } \alpha : [\![\text{g()}]\!],\ \{\alpha \to \mathcal{G}\beta \mid \beta \in succ(\alpha)\} \subset P \quad (3)$$
$$\text{for } \alpha : [\![\text{return}]\!],\ \{\alpha \to \epsilon\} \subset P \quad (4)$$
$$\text{for } \alpha : [\![\text{otherwise}]\!],\ \{\alpha \to \beta \mid \beta \in succ(\alpha)\} \subset P \quad (5)$$

Intuitively, the grammar G_t represents the control flow of the thread t, ignoring everything not related with the module's usage. Rule 1 adds a production that relates the non-terminal $\mathcal{F}$, representing a method `f()`, to the entry node of the CFG of `f()`. Calls to the module under analysis are recorded in G_t by Rule 2. Rule 3 handles calls to other methods of the client program. The return point of a method adds an ϵ production to the grammar (Rule 4). All other types of CFG nodes are handled by Rule 5 while preserving the CFG structure.

Notice that only the client program code is analyzed, given the module contract clauses and its public methods.

The generated grammar G_t may be ambiguous, i.e., offer several different derivations of the same word. Each ambiguity

[1]Partial interleavings of targets and spoilers are not considered to cause an error. If they do, this can be handled by adding a new contract clause (target) whose spoiler is the appropriate fraction of the original spoiler.

in the parsing of a sequence of calls represents different contexts where these calls may be executed by the thread t. The ambiguity is thus expected and needed so that the verification of the contract can cover all possible occurrences of sequences of calls in the client program. Since we do not consider the values of data reachable at particular locations, the language may contain sequences of calls that the program can never execute, which may lead to false positives. However, the approach is conservative and never produces false negatives.

B. Contract Verification

The verification must ensure that all sequences of calls specified by a contract are executed atomically by the threads the client program may launch. Algorithm 1 presents the pseudo-code of our static approach for verifying this requirement.

The algorithm iterates over program threads (line 2). For each thread t, it first generates, as described above, a grammar G_t that captures the CFG of t (line 3). From G_t, a grammar G'_t describing all sub-words of the words generated by G_t is obtained (line 4). The sub-words correspond to parts of executions of the original program. The sub-words must be considered since a contract clause typically corresponds to a part of a run only. For example, if a thread executes a sequence `m.a(); m.b(); m.c();` a contract can correspond to **b c** only, which G'_t allows us to recognize.

The algorithm subsequently iterates over contract clauses $\varrho \in \mathbb{R}$ (line 5) and handles them one-by-one. To see whether a thread may generate a contract clause ϱ, representing a call sequence, it is enough to parse ϱ in G'_t (line 6). This will create a parsing tree for each location from which the thread can execute the given sequence of calls. Function `parse()` returns the set T of these parsing trees.

Each of the parsing trees in T is then inspected to determine the atomicity of the given call sequence (line 7). In particular, the parsing trees contain information about the location of each of the calls of contract ϱ in the program. Then, by moving upwards in the parsing tree, we can find the node that represents the method under which the call sequence defined by the contract is performed. This node is the lowest common ancestor of the call sequence of ϱ in the parsing tree (line 8).

The algorithm then checks whether the lowest common ancestor is always executed atomically (line 9) 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 the minimal synchronization from the program. A parsing tree contains information about the location in the program where a contract violation may occur, and so we can offer detailed instructions to the programmer on where this violation occurs and how to fix it.

Since the grammar G_t may be ambiguous, it is necessary to use a GLR (generalized *LR*) parsing algorithm to explore all different derivation trees of a word [25]. In particular, in our prototype implementation discussed later on, we use a GLR parser proposed by Tomita in [38], which defines a non-deterministic version of the LR(0) parsing algorithm.

```
1  Require: P: client's program, $\mathbb{R}$: module contract;
2  for $t \in threads(P)$ do
3      $G_t \leftarrow$ build_grammar($t$);
4      $G'_t \leftarrow$ subword_grammar($G_t$);
5      for $\varrho \in \mathbb{R}$ do
6          $T \leftarrow$ parse($G'_t, \varrho$);
7          for $\tau \in T$ do
8              $N \leftarrow$ lowest_common_ancestor($\tau, \varrho$);
9              if $\neg run\_atomically(N)$ then return ERROR;
10 return OK;
```

Algorithm 1: Static contract verification algorithm.

An important point is that the number of parsing trees may be infinite since loops in the CFG will yield corresponding loops in the grammar. The parsing algorithm must therefore 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 aborts whenever it detects a loop that did not contribute to parsing a new terminal.

Example. The left part of Figure 2 shows a program that uses a module `m`. The `run()` method is the entry point of a thread t. In the middle of the figure, we show the CFGs generated by the program code. On the top right, we show a simplified version of the G_t grammar. Methods `run()`, `f()`, and `g()` are represented by non-terminals $\mathcal{R}$, $\mathcal{F}$, and $\mathcal{G}$, respectively. The obtained grammar is ambiguous. Consider a contract clause ϱ = **a b**. The right part of the figure shows two distinct ways to parse ϱ. Both of the trees will be obtained by our algorithm (line 6). The first tree (middle right) has $\mathcal{F}$ as the lowest common ancestor of **a b**. As $\mathcal{F}$ corresponds to the method `f()`, which is executed atomically (note the `atomic` keyword), we conclude that this tree respects the contract. The second tree (bottom right) has $\mathcal{R}$ as the lowest common ancestor of **a b**, corresponding to the execution of the **`else`** branch of `run()`. This non-terminal ($\mathcal{R}$) does not correspond to an atomically executed method, the contract is thus not met, and a contract violation is detected. (Another example can be found in [11].)

C. Analysis with Points-to

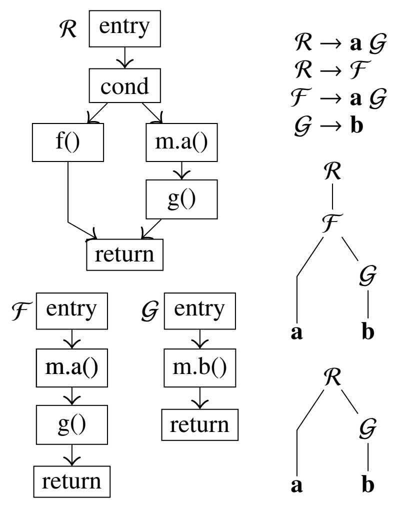
In object-oriented programming languages, a module is defined as a class, so we should differentiate between different instances of that class as they represent different objects. This section explains how our analysis can be extended to handle multiple instances of a module by using *points-to* information.

To include points-to information, we generate a different grammar for each allocation site of a module. Each allocation site represents an instance of the module, and the algorithm verifies the contract clauses for each allocation site and each thread. The revised algorithm (cf. [11]) is very similar to Algorithm 1. It iterates over threads and module instances generating a grammar G_{t_a} for a thread t and a module instance a. This grammar can be seen as the behavior of the thread t wrt the module instance a, ignoring every other instance of that module. To generate the G_{t_a} grammar, Definition 1 can be easily adapted to take into account the instance a only [11].

```
void run() {
    if (cond)
        f();
    else {
        m.a();
        g();
    }
}

void atomic f() {
    m.a();
    g();
}

void atomic g() {
    m.b();
}
```



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

Fig. 2: A program (left), its CFGs (middle), its ambiguous simplified grammar (top right), and parsing trees of **a b** (right).

D. Class Scope Mode

Our static analysis checks the entire program, taking into account any sequence of calls spreaded across the whole program (as long as they are consecutive calls to a module). However, this may become infeasible for very large programs. So, for these large programs, we propose a *class scope mode* of our analysis, an operation mode that checks each class individually, ignoring calls to other classes. This mode will detect contract violations where the control flow does not escape the class, which is reasonable since code locality indicates stronger correlations between calls.

In the class scope mode, the grammar describing the behaviours is built for each class instead of each thread. Methods of the class yield non-terminals $\mathcal{F}_1, \cdots, \mathcal{F}_n$ just as before. The only change when creating this grammar is that we create the productions $I \to \mathcal{F}_1 \mid \cdots \mid \mathcal{F}_n$ as the starting production of the grammar. This means that we consider the execution of all methods of the class under analysis.

E. Validation and Evaluation

To validate the above approach, we have implemented it in a tool called *Gluon* (https://github.com/trxsys/gluon). We used *Gluon* to analyze both some small benchmarking programs with atomicity violations, which can be seen as contract violations, as well as several real-world programs, including Tomcat, Lucene, Derby, OpenJMS, and Cassandra.

The small programs were adapted from the literature [2], [3], [4], [12], [23], [28], [40] where they are typically used to evaluate atomicity violation detection methods. We redesigned each of them as a main program using one or more modules, and we wrote the necessary contracts for each module.

For the larger, real-word programs analyzed, we aimed at discovering new, unknown, atomicity violations. For that, the contracts should ideally be written by the module developers alongside the code. However, this was not the case for the considered programs. Given that these programs had a rather large code base, we devised a way to create contracts in an automated manner by using a very simplistic approach that tries to infer the contract's clauses from the synchronized blocks present in the existing code base. The intuition behind this approach is that most sequences of calls that should be atomic are correctly used *somewhere* in the code. Having this in mind, we look for sequences of calls done to a module that are used atomically at least twice in the program as this situation may indicate that these calls are correlated and should be atomic everywhere. We used these sequences as clauses for our contracts after manually filtering a few irrelevant ones.

Since the considered real-world programs use dynamic class loading, it is impossible to obtain complete points-to information, and so we took a pessimistic approach and assumed every module instance could be referenced by any variable that is type-compatible. We also used the class scope mode described in Section III-D as it would be impractical to analyze such large programs with the scope of the whole program. These restrictions do not apply to the small programs analyzed.

Table I summarizes the results of our experiments. The table contains both the micro and macro benchmarks (top/bottom lines, resp.). 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 atomicity violations that can really occur and compromise the correct execution of the program (*Real AV*); the number lines of code of the benchmark (*SLOC*); and the time it took for the analysis to complete (*time*).

TABLE I: Validation results for static analysis.

Benchmark	**Clauses**	**Contract Violations**	**False Positives**	**Potential AV**	**Real AV**	**SLOC**	**Time (s)**
Allocate Vector [23]	1	1	0	0	1	183	0.120
Coord03 [2]	4	1	0	0	1	151	0.093
Coord04 [3]	2	1	0	0	1	35	0.039
Jigsaw [40]	1	1	0	0	1	100	0.044
Local [2]	2	1	0	0	1	24	0.033
Knight [28]	1	1	0	0	1	135	0.219
NASA [2]	1	1	0	0	1	89	0.035
Store [32]	1	1	0	0	1	621	0.090
StringBuffer [3]	1	1	0	0	1	27	0.032
UnderReporting [40]	1	1	0	0	1	20	0.029
VectorFail [32]	2	1	0	0	1	70	0.048
Account [40]	4	2	0	0	2	42	0.041
Arithmetic DB [28]	2	2	0	0	2	243	0.272
Connection [4]	2	2	0	0	2	74	0.058
Elevator [40]	2	2	0	0	2	268	0.333
OpenJMS 0.7	6	54	10	28	4	163K	148
Tomcat 6.0	9	157	16	47	3	239K	3070
Cassandra 2.0	1	60	24	15	2	192K	246
Derby 10.10	1	19	5	7	1	793K	522
Lucene 4.6	3	136	21	76	0	478K	151

For the microbenchmarks, *Gluon* was able to detect all violations of the contracts by the client programs. The absence of false negatives supports the soundness of the analysis. Since some of our tests included additional contract clauses not present in the original test programs, the results also indicate that the approach is not too inclined towards generating false positives. We created a corrected version of each microbenchmark which was also verified, and the prototype confirmed the compliance of the program with the module contract. Correcting the programs was easy since *Gluon* pinpoints the methods that must be made atomic and ensures the synchronization required has the finest possible scope (due to the use of the *lowest* common ancestor of the terminals in the parse tree).

Our tests with the macrobenchmarks have shown that *Gluon* can be applied to larger-scale programs with good results. Even with a simple automated contract generation, we were able to detect 10 atomicity violations in real-world programs. Six of these bugs were reported (Tomcat[2], Derby[3], Cassandra[4]). Two of them were immediately confirmed[2] as bugs by the Tomcat software development team and fixed in Tomcat 8.0.11, one was considered highly unlikely, and three have a pending confirmation. The false positives incorrectly reported by *Gluon* were all due to conservative points-to information in case of dynamic class loading.

The performance results show that *Gluon* is directly usable for small and medium-sized programs. For large programs, the *class scope mode* has to be used, sacrificing precision for performance, but still allowing one to capture interesting atomicity violations as shown by our results with Tomcat. The performance of *Gluon* strongly depends on the number of branches the parser explores. Larger programs tend to have more complex control flows and generate larger number of parsing branches. The parsing phase of *Gluon* dominates the execution time, which is proportional to the number of explored parsing branches. Memory usage is not a problem since the asymptotic space complexity is determined by the size of the parsing table and the largest parsing tree. It is not correlated with the number of parsing trees as our *GLR* parser explores the parsing branches in-depth instead of in-breadth. In-depth exploration is possible since we never have infinite height parsing trees due to our detection of unproductive loops.

IV. Dynamic Contract Validation

We now propose a *dynamic contract validation* method for contracts with contextual information (i.e., using both targets and spoilers) as defined in Section II-C. Though not discussed here, the method can be easily extended to support parameters by considering separate instances of target/spoiler pairs for different values of parameters (as done in our implementation).

Below, we first formalize a notion of multi-threaded program traces used as the input of our analysis. Then we define the happens-before relation that captures the ordering of events in program traces. Next, we describe our method for detecting contract violations. Finally, we provide results of experiments with a prototype implementation of the approach.

A. Preliminaries

For the below, we fix a set of threads $\mathbb{T}$, a set of targets $\mathbb{R}$, a set of spoilers $\mathbb{S}$, a set of contracts $\mathbb{C} \subseteq \mathbb{R} \times \mathbb{S}$, and a set of locks $\mathbb{L}$. We consider program traces in the form of sequences of events of the following types: a thread entering/exiting a method, a thread acquiring/releasing a lock, and a thread forking/joining another thread. Since each of the events can appear multiple times in a trace, we assume the events to be indexed by their position in the trace. However, we do not take the indices into account when looking for matches of the regular expressions of targets/spoilers in a trace. We denote the set of all events that can be generated by a thread $t \in \mathbb{T}$ as $\mathbb{E}_t$, and let $\mathbb{E} = \cup_{t \in \mathbb{T}} \mathbb{E}_t$. Then, a *trace* is a sequence $\tau = e_1 \ldots e_n \in \mathbb{E}^+$. We let $e_i \in \tau$ denote that the event e_i is present in the trace τ. By $start(t)/end(t)$, we denote the first/last event generated by a thread t.

Given a trace $\tau = e_1 \ldots e_n \in \mathbb{E}^+$, we call its sub-sequence $r = e_{i_1} e_{i_2} \ldots e_{i_k}$, $1 < k \leq n$, an *instance* of a target $\varrho \in \mathbb{R}$ iff (1) r consists of well-paired method enter/exit events executed by a thread $t \in \mathbb{T}$, (2) when restricted to the enter events only, r matches the regular expression of ϱ (if ϱ contains stars, the longest possible matches are considered only), and (3) apart from the events $e_{i_1}, ..., e_{i_k}$ there is no event from the alphabet of ϱ executed by t between the indices i_1 and i_k in τ. Intuitively, an instance of a target can interleave with events that are not its part, but only if they are outside of its alphabet. For instance, for a target $\varrho = abc$ and a trace $\tau = aabdc$, there is an instance of ϱ between indices 2 and 5 but not between 1 and 5. We denote by $e_i \in r$ that the event e_i is present in the target instance r. We let $start(r) = e_{i_1}$ and $end(r) = e_{i_k}$ denote the first/last event of r, respectively. We let $[\varrho]^\tau$ be the set of all instances of a target $\varrho \in \mathbb{R}$ in a trace τ and $[\mathbb{R}]^\tau = \cup_{\varrho \in \mathbb{R}} [\varrho]^\tau$ be the set of all instances of all targets from $\mathbb{R}$ in τ.

Likewise, we define the notion of an instance s of a spoiler $\sigma \in \mathbb{S}$ in a trace τ, its beginning/end events $start(s)/end(s)$, respectively, the set $[\sigma]^\tau$ of all instances of σ in τ, and the set $[\mathbb{S}]^\tau = \cup_{\sigma \in \mathbb{S}} [\sigma]^\tau$ of all instances of all spoilers from $\mathbb{S}$ in τ.

A *happens-before relation* $\prec_{hb}$ over a trace $\tau = e_1 \ldots e_n \in \mathbb{E}^+$ is the smallest transitively-closed relation on the set $\{e_1, ..., e_n\}$ of events in τ such that $e_j \prec_{hb} e_k$ holds whenever $j < k$ and one of the following holds: (i) Both events e_j and e_k are performed by the same thread (program order). (ii) Both events e_j and e_k acquire or release the same lock. (iii) One of the events e_j and e_k is a fork/join of a thread u in a thread t and the other is executed by u (fork-join synchronization). If two indices in a trace are not related by a happens-before relation, then the corresponding events are considered to be *concurrent*.

A contract $(\varrho, \sigma) \in \mathbb{C}$ is *violated* in a trace τ iff there is a target instance $r \in [\varrho]^\tau$ and a spoiler instance $s \in [\sigma]^\tau$ s.t. $start(s) \not\prec_{hb} start(r) \wedge end(r) \not\prec_{hb} end(s)$. Intuitively, the contract (ϱ, σ) is violated in τ if there are instances r/s of ϱ/σ, resp., where r may start before s and end after s, i.e., the target instance can be fully interleaved with the spoiler instance.

[2]https://issues.apache.org/bugzilla/show_bug.cgi?id=56784
[3]https://issues.apache.org/jira/browse/DERBY-6679
[4]https://issues.apache.org/jira/browse/CASSANDRA-7757

B. On-the-Fly Dynamic Contract Validation

If the entire trace is available, dynamic contract validation is easy. For all possibly conflicting instances of targets and spoilers, one simply checks whether a target is fully interleaved with a spoiler or not, i.e., $\forall(\varrho, \sigma) \in \mathbb{C}, \forall r \in [\varrho]^\tau, \forall s \in [\sigma]^\tau$ checks if $start(s) \not\prec_{hb} start(r) \wedge end(r) \not\prec_{hb} end(s)$ is satisfied. If it is, an error is reported.

However, this approach is not very practical. It scales poorly with the size of the trace, which can be huge. In some cases, e.g., for reactive programs, the trace can even be infinite. To address this problem, we propose an on-the-fly dynamic contract validation algorithm which does not require the whole trace to be available and yet guarantees that if a contract is violated in the trace, this will be detected.

1) Trace Windows: A crucial concept for our on-the-fly dynamic contract validation is the concept of a *trace window*, providing a gradually moving, partial view of the trace. Formally, a trace window υ is a subsequence of the trace τ. While, in the extreme case, the trace window may actually contain the entire trace, the goal is to keep it as small as possible. Later, we show that there is a maximum number of events that we need to keep in the window in order not to miss any error and that this number grows only with the number of targets and spoilers, not with the size of the trace.

We denote by $[\varrho]^\upsilon$ the set of all instances of a target $\varrho \in \mathbb{R}$ in a window υ and by $[\mathbb{R}]^\upsilon = \cup_{\varrho\in\mathbb{R}}[\varrho]^\upsilon$ the set of all instances of all targets from $\mathbb{R}$ in υ. In the same manner, we define the set $[\sigma]^\upsilon$ of all instances of spoiler $\sigma \in \mathbb{S}$ in υ and the set $[\mathbb{S}]^\upsilon = \cup_{\sigma\in\mathbb{S}}[\sigma]^\upsilon$ of all instances of all spoilers from $\mathbb{S}$ in υ.

We move events into the trace window υ as soon as they occur. However, in order for the window not to grow indefinitely, we also have to remove some events from it. We define the $\upsilon \to e$ operation which removes e from υ. We also generalize this operation for instances of targets/spoilers. The $\upsilon \to r$ operation removes all events from $r \in [\mathbb{R}]^\upsilon$ from υ provided they do not belong to another currently tracked instance of a target or spoiler, i.e., $\forall e_i \in r : \upsilon \to e_i \iff (\forall x \in [\mathbb{R}]^\upsilon \cup [\mathbb{S}]^\upsilon, x \neq r : e_i \notin x) \wedge (\forall x \in [\mathbb{R}]^\tau \cup [\mathbb{S}]^\tau, start(x) \in \upsilon \wedge end(x) \notin \upsilon : e_i \notin x)$. Likewise, we define the $\upsilon \to s$ operation that removes all events from $s \in [\mathbb{S}]^\tau$ from υ. As we show below, one can discard events corresponding to some of the older spoiler and target instances when newer ones appear in the window. The conditions allowing us to discard such instances are safe in that at least one instance of a violation of each target by each spoiler is always reported. However, if there are multiple occurrences of the conflict, just one is guaranteed to be preserved.

2) Discarding Spoilers: First, we aim at reducing the number of spoiler instances in a trace window. We say that discarding a spoiler instance s (i.e., removing this particular instance from the current trace window and not considering it in further contract violation detection) is *safe* iff whenever a contract violation can be detected using s, it can be detected without s too. The below lemma shows that, under some natural assumptions, reflected in our analysis, an instance s_1 of a spoiler σ can be safely discarded from the window provided the window contains a newer instance of the spoiler σ, i.e., an instance s_2 that started later than s_1.

In particular, we assume that events appear in the window υ as soon as they appear in the trace τ. Moreover, we assume that as soon as an instance r of a target ϱ appears in the window υ, i.e., $r \in [\varrho]^\upsilon$ becomes true, r is checked for contract violation against all instances s of all spoilers $\sigma \in \mathbb{C}(\varrho)$ conflicting with the given target ϱ that appear in the window υ, i.e., $s \in [\sigma]^\upsilon$. Then the following holds (for proofs, see [11]).

Lemma 1. *Let $s_1, s_2 \in [\sigma]^\upsilon$ be instances of a spoiler $\sigma \in \mathbb{S}$ present in a window υ of a trace τ. If s_1 started before s_2, i.e., $start(s_1) \prec_{hb} start(s_2)$, it is safe to discard s_1 from υ.*

Using Lemma 1 and the fact that spoiler instances in a single thread are ordered wrt $\prec_{hb}$, we can prove the below lemma that limits the number of spoiler instances to be preserved.

Lemma 2. *Let $T = \{ t \in \mathbb{T} \mid start(t) = e_l \Rightarrow l \leq j \}$ be the set of threads that started before the end of a window $\upsilon = e_i \dots e_j$. For each thread $t \in \mathbb{T}$ and for each spoiler $\sigma \in \mathbb{S}$, we need to preserve just the last instance of σ in υ running within t.*

3) Discarding Targets: We now aim at reducing the number of target instances, which turns out to be more challenging than for spoilers. We say that discarding a target instance r is *safe wrt a spoiler instance s* iff whenever a contract violation between r and s can be detected, then a conflict between s and some other target instance r' can be detected too. Note that, unlike in the case of spoilers, discarding a target instance is defined as safe wrt a given spoiler instance and not in general.

First, Lemma 3 shows that, given instances r_1 and r_2 of a target ϱ where r_1 ends before r_2 starts, r_1 can be safely discarded wrt any spoiler instance that (i) has not even started before the end of the window or that (ii) started even before r_1.

Lemma 3. *Let $\upsilon = e_i \dots e_j$ be a window of a trace τ with two instances $r_1, r_2 \in [\varrho]^\upsilon$ of a target $\varrho \in \mathbb{R}$ such that $end(r_1) \prec_{hb} start(r_2)$. It is safe to discard r_1 wrt any instance $s \in [\sigma]^\tau$ of a spoiler $\sigma \in \mathbb{S}$ forming a contract with ϱ, i.e., $(\varrho, \sigma) \in \mathbb{C}$, whenever either (i) s starts behind the window υ, meaning that if $start(s) = e_l$, then $j < l$, or (ii) s starts before r_1 starts, i.e., $start(s) \prec_{hb} start(r_1)$.*

Next, we consider the case when an instance s of a spoiler σ is running at the end of the window υ, there are two instances r_1 and r_2 of the same target ϱ conflicting with σ, r_1 ends before r_2 starts, but s does not start before r_1 and r_2. Lemma 4 shows that, in this case, discarding r_1 is safe wrt s.

Lemma 4. *Assume a window υ of a trace τ with two target instances $r_1, r_2 \in [\varrho]^\upsilon$ of a target $\varrho \in \mathbb{R}$ s.t. $end(r_1) \prec_{hb} start(r_2)$. Let $s \in [\sigma]^\tau$ be an instance of a spoiler $\sigma \in \mathbb{S}$ that forms a contract with ϱ, i.e., $(\varrho, \sigma) \in \mathbb{C}$, it is running at the end of υ, i.e., $start(s) \in \upsilon$ but $end(s) \notin \upsilon$, and it has not started before the given target instances, i.e., $start(s) \not\prec_{hb} start(r_2)$. Then discarding r_1 is safe wrt s.*

Since we check each spoiler instance against all target instances that are currently in the trace window as soon as the spoiler instance gets into the window, we can prove the below upper bound on the number of target instances to be preserved. Intuitively, by Lemma 3, one instance is kept wrt all not yet started and—on the other hand—old but still running spoiler instances. Further, by Lemma 4, one instance per thread in which a newer spoiler instance is running is to be preserved.

Lemma 5. *Let $T_1 = \{ t \in \mathbb{T} \mid start(t) = e_l \Rightarrow l \leq j \}$ be the threads that started before the end of a window $\upsilon = e_i \dots e_j$, and let $T_2 = \{ t \in T_1 \mid end(t) = e_l \Rightarrow l > j \}$ be the threads running at the end of υ. For each thread in T_1 and each target $\varrho \in \mathbb{R}$, we need to preserve at most $|T_2| + 1$ instances of ϱ.*

4) Vector Clocks and Further Optimizations: Next, as a further optimization, we will first introduce an application of *vector clocks* for efficiently tracking information about the happens-before relation between the spoiler/target instances that are (or were) in the current trace window. Essentially, instead of remembering the entire sequence of events forming a target/spoiler instance, we will remember the vector clocks of their start and end only. Keeping just these two vector clocks is sufficient as we need to know the happens-before relation only between the starts and ends of conflicting target/spoiler instances. Next, from Lemma 5, we know that we need to track—in the worst-case—for each thread and for each target, one instance of the target for each thread in which some potentially conflicting spoiler instance is running (a consequence of Lemma 4) plus one further instance for all other running or not yet started spoiler instances (a consequence of Lemma 3). We will propose an optimisation which will allow us to preserve, for each thread t and each target ϱ, the vector clocks of both the beginning and end just for the last instance of ϱ in t only. For the other instances required to be tracked by Lemma 4, we will remember the vector clock of their end only.

In general, a *vector clock* $VC : \mathbb{T} \to \mathbb{N}$ contains a clock value for each thread $t \in \mathbb{T}$ recorded at a certain point. In particular, we maintain, for each $t \in \mathbb{T}$, a vector clock $\mathbb{C}_t$ whose entries $\mathbb{C}_t(u)$ record, for each $u \in \mathbb{T}$, the clock value of the last operation of u that happens before the current operation of t. The t-component of this vector clock then represents the clock of the thread t. It is incremented at each lock release or fork operation. Next, we maintain a vector clock $\mathbb{L}_l$ for each lock $l \in \mathbb{L}$. These vector clocks are updated on synchronisation operations that impose a happens-before order of operations from different threads in a way described in [19].

Further, we assign to each event $e \in \tau$ executed by a thread $t \in \mathbb{T}$ a vector clock VC_e. This vector clock is set to the value of $\mathbb{C}_t$ when e is encountered in the execution of the program. It can then be determined whether an event e_t executed in the thread t happens before an event e_u executed in a thread u, i.e., $e_t \prec_{hb} e_u$, by checking whether $VC_{e_t}(t) \leq VC_{e_u}(t)$.

To allow for checking the conditions determining if a contract was violated or not, it now suffices to record the vector clocks of the start and end of the spoiler and target instances that are to be kept in the window wrt Lemmas 1, 3, and 4.

Moreover, for the target instances r to be remembered according to Lemma 4, i.e., those for which there is some running spoiler instance s that can collide with r, we can reduce the amount of stored information even further as follows. Instead of storing the vector clocks of the beginning and end of each target instance r of the above kind that appears in some thread t, we proceed as follows: (1) We remember in which threads u there are running spoiler instances s satisfying the first condition of contract violation wrt r, i.e., $start(s) \not\prec_{hb} start(r)$. (2) We remember the time when r ends its execution, i.e., $VC_{end(r)}(t)$, which is needed to check the second condition of contract violation, i.e., $end(r) \not\prec_{hb} end(s)$, once s ends. Both of these pieces of information can be remembered by maintaning a mapping $PV_t^{\varrho,\sigma} : \mathbb{T} \to \mathbb{N}$ for the thread $t \in \mathbb{T}$, the target $\varrho \in \mathbb{R}$ whose instance r is, and the spoiler $\sigma \in \mathbb{S}$ whose instance s is. Namely, for each thread u containing a spoiler instance s satisfying the first condition of contract violation, we may set $PV_t^{\varrho,\sigma}(u)$ to $VC_{end(r)}(t)$, while setting the other entries of $PV_t^{\varrho,\sigma}$ to 0.[5]

Using the above, when a spoiler instance s finishes its execution in a thread t, it suffices to check $PV_u^{\varrho,\sigma}(t)$ for each thread u other than t (as we do not consider conflicts within a single thread).[6] If the value is not 0, we know that the first condition of contract violation between s and the target instance r that ran in the thread u that we remebered through $PV_u^{\varrho,\sigma}$ only was satisfied. Then, by checking $PV_u^{\varrho,\sigma}(t) \leq VC_{end(s)}(u)$, we can determine if a violation occurred or not.

5) Method Description: We now summarise our optimized on-the-fly contract violation detection. Most of it is done by Algorithm 2 at method exit events. Algorithm 2 handles both conflicts between the latest, so far fully remembered spoiler and target instances (lines 3, 11) as well as between newly finished spoiler instances and older target instances partially remembered via $PV_t^{\varrho,\sigma}$ (lines 12–13). Algorithm 2 also discards older target/spoiler instances r'/s' (lines 7, 9) and maintains the $PV_t^{\varrho,\sigma}$ mapping (line 6). The latter is done by recording the above described data about an older target instance r' that can still collide with some running spoiler instance s according to Lemma 4, which is tested on lines 4–6, before r' is removed from the window.

Apart from the above, at an entry to a method, we perform recognition of target/spoiler instances. That is done using finite automata for recognising sequences of events matching the regular expressions representing the corresponding targets/spoilers, respectively. New runs through the automata may be initiated at each event, and, at the same time, an attempt to extend all so-far unfinished runs is done (if such a run cannot be extended via the current event and the event belongs to the alphabet of the concerned automaton, the run is discarded). When an exit from a method is encountered,

[5]By setting $PV_v^{\varrho,\sigma}(u)$ to $VC_{end(r)}(t)$, we remember both that the first condition of contract violation has been satisfied between r and s and the time when r ended. The time is remembered multiple times for possibly different threads u, but we tolerate this for the sake of obtaining uniform data structures. Since the space needed to store $PV_t^{\varrho,\sigma}$ corresponds to that of a vector clock, and we have a single $PV_t^{\varrho,\sigma}$ instead of two vector clocks for each target instance that needs to be remembered according to Lemma 4, we save up to $2 \cdot |T_2| - 1$ vector clocks where T_2 is the set of currently running threads.

[6]The meaning of the threads is swapped here wrt the previous paragraph in order to have the explanation in line with the code in Fig. 2.

```
    Data: window $\upsilon$, event $e \in \mathbb{E}$ generated by thread $t \in \mathbb{T}$
 1  if $\exists \varrho \in \mathbb{R}, r \in [\varrho]_t^\upsilon : e = end(r)$ then          // Target ended
 2    for $\sigma \in \mathbb{C}(\varrho), u \in \mathbb{T} : u \neq t$ do
 3      if $\exists s \in [\sigma]_u^\upsilon : start(s) \not<_{hb} start(r) \wedge end(r) \not<_{hb} end(s)$
          then $r$ is violated by $s$ ;
 4      if $\exists s \in [\sigma]_u^\tau : start(s) \in \upsilon \wedge end(s) \notin \upsilon$ then
 5        if $start(s) <_{hb} start(r)$ then
 6          if $\exists r' \in [\varrho]_t^\upsilon : r' \neq r \wedge start(s) \not<_{hb} start(r')$ then
              $PV_t^{\varrho,\sigma}(u) = VC_{end(r')}(t)$ ;
 7    if $\exists r' \in [\varrho]_t^\upsilon : r' \neq r$ then $\upsilon \rightarrow r'$ ;
 8  if $\sigma \in \mathbb{S}, s \in [\sigma]_t^\upsilon : end(s) = e$ then          // Spoiler ended
 9    if $\exists s' \in [\sigma]_t^\upsilon : s' \neq s$ then $\upsilon \rightarrow s'$ ;
10    for $\varrho \in \mathbb{C}(\sigma), u \in \mathbb{T} : u \neq t$ do
11      if $\exists r \in [\varrho]_u^\upsilon : start(s) \not<_{hb} start(r) \wedge end(r) \not<_{hb} end(s)$
          then $r$ is violated by $s$ ;
12      if $PV_u^{\varrho,\sigma}(t) \neq 0 \wedge PV_u^{\varrho,\sigma}(t) \leq VC_{end(s)}(u)$ then
13        an instance of $\varrho$ is violated by $s$;
```

Algorithm 2: Contract violation detection at method exit.

a check is performed to see whether some of the runs has reached an accepting state (this will then be recognised via the $end(r)/end(s)$ predicates on lines 1/8 of Algorithm 2).

C. Implementation and Experiments

We implemented the above approach extended to distinguish values of one parameter by tracking different target/spoiler instances for its different values. We used the ANaConDA framework [17] to monitor method calls and synchronization events in running C/C++ programs. ANaConDA also provides us with heuristic *noise injection* [15] that can disturb the common thread scheduling by inserting various delays into the threads. This can increase the number of witnessed interleavings and hence chances to see an interleaving from which our analysis can deduce that a contract violation is possible. We thus use two orthogonal methods to find rare concurrency-related bugs: noise injection and extrapolation based on the happens-before relation. In particular, we inject noise before the last method of each target instance which prolongs its execution and increases chances to encounter a spoiler instance capable of interleaving the target instance and causing a contract violation.

We tested our implementation on a set of small benchmarks with known atomicity violations as well as two real-world programs, Link Manager and Chromium-1. The small programs were taken from [2], [3], [40] and were also used in Section III-E to evaluate the static validation method (we used a C++ version as close as possible to the Java version).

Link Manager is a component of a cloud-connected thermostat used for managing parallel task processing (we were not allowed to identify the company developing it). A *manager* thread is issuing tasks to *executor* threads, which send results of the assigned tasks back to the *manager* through a shared queue. Our tool was used in the early stages of development of this program, and it uncovered an order violation error that happened when an *executor* sent the result of its task before the *manager* initialised the queue used to transfer the data. This caused the *manager* to wait forever for the task to be finished. One of the contracts we checked required that the queue cannot be used before it is initialised, i.e., no `send` or `receive` can occur between the start of the *manager* and the initialisation of the queue. The error occurred very rarely, so normal tests were unable to detect it. Our tool, however, was able to detect the error, and it was then promptly fixed.

TABLE II: Validation results for dynamic analysis.

Benchmark	T/S pairs	Contract Violations	False Positives	Potential AV	Real AV	SLOC	Time (s)
Coord03 [2]	8	380	0	0	380	116	1.01
Coord04 [3]	4	24	0	0	24	53	0.52
Local [2]	4	2	0	0	2	27	0.52
NASA [2]	1	100	0	0	100	96	0.60
Account [40]	1	176	0	0	176	54	0.53
Link Manager	2	1	0	0	1	1.5K	1.14
Chromium-1	2	2	0	0	2	7.5M	49.12

Chromium-1 is a program from the RADBench benchmark [24], an older version of the Chrome browser (version 6.0.472.35) containing a known atomicity violation leading to an assertion failure. As this error can be described using a contract, we tried our tool to find the error. The experiment was successful, showing that our tool can handle even large programs. Interestingly, to find the error without the on-the-fly approach, one would need to store a trace with more than 17 million method calls (about 1.6 GB of data) while the on-the-fly method needed about 10 MB of data only.

Table II provides results of experiments with our dynamic approach. The *T/S Pairs* column gives the number of target/spoiler pairs considered. The column *Contract Violations* gives the number of instances of such pairs found violated.[7] The column *False Positives*, included for compatibility with Table I, contains zeros only as, unlike the static approach, the dynamic one considers solely executable sequences of method calls (indeed, they were seen to execute). The column *Potential AV* contains numbers of detected contract violations that need not stay real if the values of more than one parameter per contract are taken into account (which is not yet supported in our tool). The column contains zeros only showing that we sufficed with tracking a sole parameter in all our experiments.[8] The column *Real AV* gives numbers of contract violations guaranteed to be real as they used at most one parameter, and our tool was thus able to distinguish the needed instances. Finally, the columns *SLOC* and *Time* give the numbers of lines of the considered programs and the analysis time.

The results show that our approach can be used to find real errors in real-world programs. Moreover, it can be used to detect not only atomicity violations, but also order violations which are hard to be found using exiting techniques.

[7]Compared with the static approach, we look for contract violations in the *execution* of a program, not its source code. As the code containing a contract violation may be executed repeatedly, we can detect (and report) the same contract violation many times. The static approach reports it only once.

[8]We tried an experiment in which we tracked no parameter values at all. Then, for Chromium-1, our tool reported 14 potential violations instead of the 2 real ones, showing that distinguishing target/spoiler instances is important.

V. Related Work

Design by contract was introduced by Meyer [31] as a way to write robust code, using contracts between programs and objects, checked at runtime. In this context, a contract consists of a pre- and post-condition of a method such that when the call of a method satisfies its pre-condition, the post-condition is guaranteed to be satisfied upon return from the method.

Cheon et al. [8] proposed a way of using contracts to specify protocols for accessing objects in a sequential setting. The contracts use regular expressions describing sequences of calls that can be executed for a given object. Hurlin [22] extended [8] with operators allowing one to specify which methods may be executed concurrently. The work, however, does not show how to validate such contracts, it only proposes a technique for automatically generating programs from contracts that are to be proven correct (e.g., by theorem proving) to show that the contracts adhere to the protocols they specify.

In [5], [34], *typestates* are used to specify protocols for accessing objects. A typestate can describe both the legal sequences of method calls and the data these methods may work with. In [5], the protocol must be defined by the user and then validated using three static analyses. If these analyses cannot establish correctness of the program, dynamic analysis is used to find protocol violations. In [34], a dynamic analysis is used to automatically infer protocols from program runs and then static analysis is used to check the protocols. All the protocols, however, do not consider concurrency-related issues. Beckman et al. [4] showed how to use *typestates* in concurrent scenarios. Their approach, however, requires the user not only to define the protocols to be checked, but also to annotate the code with additional information needed by the static checker to check if the protocols are respected. Typestate specifications are also much more complex compared with the specifications based on contracts we propose in this paper.

The work [30] deals with JavaMOP specifications of desired program properties that are validated dynamically at runtime. Using the approach, one can specify that some sequence of methods must be atomic, but the specific way of ensuring the atomicity (e.g., the fact that some lock must be held) has to be encoded by the user in the specification. On the other hand, when our contracts are used for checking atomicity, the user just specifies the sequence of method calls and does not have to care about the way the atomicity should be ensured.

Most works targeting errors in concurrent programs have concentrated on detecting data races and deadlocks. These errors are, however, of a different nature than those captured by contracts, and hence methods and tools developed for detecting them—including well-known ones, such as, Eraser [36], RaceTrack [42], GoldiLocks [13], FastTrack [19], or GoodLock [21]—cannot be used for contract violation detection.

Significantly less works targeted detection of various kinds of atomicity violation [18], [29], [40], including different forms of high-level data races [2], [12], [14] or stale value errors [3], [6], [12]. Detectors based on access patterns to shared variables [28], [39], type systems [7], semantic invariants [10], and dynamic analysis [18], [20], [41] have been proposed for detecting this kind of errors. Despite atomicity violation is closer to contract violation, contract violation is still more general. This is, atomicity violations can be detected as contract violations (possibly with a need to view accesses to variables as method calls) but not vice versa. An example of an error that can be captured via contract validation but not atomicity validation is that of order violation. Such an error happens in the Link Manager where a shared queue is used before it is initialised. As the queue (variable) is accessed only once in each of the threads and both accesses are guarded by the same lock, it is neither an atomicity violation nor a data race, and yet we were able to detect it.

ICFinder [27] is the closest tool to *Gluon*. It uses a static analysis to automatically infer which pairs of calls to a module are incorrect. This is achieved by identifying and applying two common incorrect composition patterns: one capturing stale value errors and the other one trying to infer correlations between method calls by analyzing the CFG of the client's program. These patterns are extremely broad and yield many false positives. The authors address this issue by filtering the results from the static analysis with a dynamic analysis that only considers violations defined in [39]. This analysis assumes that the notion of atomic set was correctly inferred by ICFinder. None of the atomicity violations detected by *Gluon* in our larger benchmarks was captured by ICFinder since their patterns failed to match the source of those violations.

In [16], a dynamic contract validation based on lock-sets was proposed. However, it supports basic contracts only, it can miss many violations, and it reports false positives. Our approach is based on the happens-before relation [19], [33], [41], encoded by vector clocks in a way specifically optimised for efficient tracking of target and spoiler instances. It supports contracts with spoilers, and it is able to detect more violations without producing false positives.

VI. Conclusion and Future Work

We have extended the previously established notion of contracts for concurrency with arguments and spoilers, each of the extensions allowing one to describe contracts more precisely. Then, we have proposed two methods to validate such contracts—namely, a static and a dynamic one, each of them offering complementary advantages. We have evaluated both methods on a set of simple as well as real-world programs, showing that both of them can be practically useful.

There are many possibilities for future work. For instance, while it is conceptually easy to support contracts with both arguments and spoilers in the dynamic approach, this can be rather costly in practice due to many target and spoiler instances to be tracked. Suitable optimisations are thus likely needed. Next, static validation of contracts with contextual information remains open. Further, it seems promising to combine the static and dynamic approach—e.g., by letting the static approach to drive the dynamic one to likely problematic code. More involved ways of automatically deriving contract candidates are also an interesting issue for further work.

VII. Acknowledgment

The Czech authors were supported by the Czech Science Foundation (project 17-12465S), the internal BUT FIT project FIT-S-17-4014, and the IT4IXS: IT4Innovations Excellence in Science (LQ1602). The Portuguese authors were supported by NOVA LINCS (ref. UID/CEC/04516/2013).

References

[1] F. E. Allen. Control flow analysis. *SIGPLAN Not.*, 5(7):1–19, July 1970.

[2] C. Artho, K. Havelund, and A. Biere. High-level data races. *Software Testing, Verification and Reliability*, 13(4):207–227, Dec. 2003.

[3] C. Artho, K. Havelund, and A. Biere. Using block-local atomicity to detect stale-value concurrency errors. *Automated Technology for Verification and Analysis*, pages 150–164, 2004.

[4] N. E. Beckman, K. Bierhoff, and J. Aldrich. Verifying correct usage of atomic blocks and typestate. *SIGPLAN Not.*, 43(10):227–244, Oct. 2008.

[5] E. Bodden and L. Hendren. The clara framework for hybrid typestate analysis. *International Journal on Software Tools for Technology Transfer*, 14(3):307–326, 2012.

[6] M. Burrows and K. Leino. Finding stale-value errors in concurrent programs. *Concurrency and Computation: Practice and Experience*, 16(12):1161–1172, 2004.

[7] L. Caires and J. a. C. Seco. The type discipline of behavioral separation. *SIGPLAN Not.*, 48(1):275–286, Jan. 2013.

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

[9] C. Cowan, H. Hinton, C. Pu, and J. Walpole. The cracker patch choice: An analysis of post hoc security techniques. In *Proceedings of the 23rd National Information Systems Security Conference*. USENIX Association, 2000.

[10] 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)*, page 10, 2012.

[11] R. F. Dias, C. Ferreira, J. Fiedor, J. M. Lourenço, A. Smrčka, D. G. Sousa, and T. Vojnar. Verifying concurrent programs using contracts. Technical report, 2016. http://www.fit.vutbr.cz/~vojnar/Publications/tr-contracts-16.pdf.

[12] 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. Springer Berlin / Heidelberg, Nov. 2012. HVC 2012 Best Paper Award.

[13] T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: A Race and Transaction-aware Java Runtime. In *Proc. of PLDI'07*, pages 245–255, New York, NY, USA, 2007. ACM.

[14] 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*, PADTAD 2012, pages 18–24, New York, NY, USA, 2012. ACM.

[15] J. Fiedor, V. Hrubá, B. Křena, Z. Letko, S. Ur, and T. Vojnar. Advances in noise-based testing. *STVR*, 24(7):1–38, 2014.

[16] J. Fiedor, Z. Letko, J. Lourenço, and T. Vojnar. Dynamic validation of contracts in concurrent code. In *Computer Aided Systems Theory–EUROCAST 2015*, number 9520, pages 555–564. Springer-Verlag, 2015.

[17] J. Fiedor and T. Vojnar. ANaConDA: A Framework for Analysing Multi-threaded C/C++ Programs on the Binary Level. In *Proc. of RV'13*, volume 7687 of LNCS, pages 35–41. Springer-Verlag, 2013.

[18] C. Flanagan and S. N. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. *SIGPLAN Not.*, 39(1):256–267, Jan. 2004.

[19] C. Flanagan and S. N. Freund. FastTrack: efficient and precise dynamic race detection. In *PLDI'09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 121–133, New York, NY, USA, 2009. ACM.

[20] C. Flanagan, S. N. Freund, and J. Yi. Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs. *SIGPLAN Not.*, 43(6):293–303, 2008.

[21] K. Havelund. Using runtime analysis to guide model checking of java programs. In *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, pages 245–264, London, UK, 2000. Springer-Verlag.

[22] C. Hurlin. Specifying and checking protocols of multithreaded classes. In *Proceedings of the 2009 ACM symposium on Applied Computing*, SAC '09, pages 587–592, New York, NY, USA, 2009. ACM.

[23] IBM's Concurrency Testing Repository.

[24] N. Jalbert, C. Pereira, G. Pokam, and K. Sen. Radbench: A concurrency bug benchmark suite. In *Proceedings of the 3rd USENIX Conference on Hot Topic in Parallelism*, HotPar'11, Berkeley, CA, USA, 2011. USENIX Association.

[25] D. E. Knuth. On the translation of languages from left to right. *Information and control*, 8(6):607–639, 1965.

[26] B. Krebs. A time to patch II: Mozilla, 2006. Last visited March 2016.

[27] 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*, pages 158–168. ACM, 2013.

[28] J. Lourenço, D. Sousa, B. Teixeira, and R. Dias. Detecting concurrency anomalies in transactional memory programs. *Computer Science and Information Systems/ComSIS*, 8(2):533–548, 2011.

[29] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: Detecting Atomicity Violations via Access Interleaving Invariants. In *Proc. of ASPLOS'06*, pages 37–48, New York, NY, USA, 2006. ACM.

[30] Q. Luo, Y. Zhang, C. Lee, D. Jin, P. O. Meredith, T. F. ŞerbănuŢă, and G. Roşu. *RV-Monitor: Efficient Parametric Runtime Verification with Simultaneous Properties*, pages 285–300. Springer International Publishing, Cham, 2014.

[31] B. Meyer. Applying "design by contract". *Computer*, 25(10):40–51, Oct. 1992.

[32] V. Pessanha. Verificação prática de anomalias em programas de memória transaccional (Practical verification of anomalies in transactional memory programs). Master's thesis, Universidade Nova de Lisboa, 2011.

[33] E. Pozniansky and A. Schuster. Efficient On-the-fly Data Race Detection in Multithreaded C++ Programs. In *Proc. of PPoPP'03*, pages 179–190, New York, NY, USA, 2003. ACM.

[34] M. Pradel, C. Jaspan, J. Aldrich, and T. R. Gross. Statically checking api protocol conformance with mined multi-object specifications. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 925–935, Piscataway, NJ, USA, 2012. IEEE Press.

[35] E. Rescorla. Security holes... who cares? In *Proceedings of the 12th Conference on USENIX Security Symposium*, pages 75–90, Berkeley, CA, USA, 2003. USENIX Association.

[36] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A Dynamic Data Race Detector for Multi-threaded Programs. In *Proc. of SOSP'97*, pages 27–37, New York, NY, USA, 1997. ACM.

[37] D. G. Sousa, R. J. Dias, C. Ferreira, and J. M. Lourenço. Preventing atomicity violations with contracts. *arXiv preprint arXiv:1505.02951*, May 2015.

[38] M. Tomita. An efficient augmented-context-free parsing algorithm. *Comput. Linguist.*, 13(1-2):31–46, Jan. 1987.

[39] M. Vaziri, F. Tip, and J. Dolby. Associating synchronization constraints with data in an object-oriented language. In *ACM SIGPLAN Notices*, volume 41, pages 334–345. ACM, 2006.

[40] C. Von Praun and T. Gross. Static detection of atomicity violations in object-oriented programs. *Journal of Object Technology*, 3(6):103–122, 2004.

[41] J. Yi, C. Sadowski, and C. Flanagan. SideTrack: Generalizing Dynamic Atomicity Analysis. In *Proceedings of the 7th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*, PADTAD '09, pages 8:1–8:10, New York, NY, USA, 2009. ACM.

[42] Y. Yu, T. Rodeheffer, and W. Chen. Racetrack: efficient detection of data race conditions via adaptive tracking. *SIGOPS Oper. Syst. Rev.*, 39(5):221–234, 2005.