


# Automatic Generation of Contracts for Concurrent Java Programs<sup>\*</sup>

Hugo Gamaliel Pereira 

NOVA University Lisbon | FCT & NOVA LINCS, Portugal  
hg.pereira@campus.fct.unl.pt

**Abstract.** It is not uncommon for larger concurrent programs to have hundreds of atomic blocks and/or locks. As these programs become more and more complex, they become increasingly more prone to contain concurrency errors, as a programmer cannot easily take into account every possible interleaving. One way to address this problem is to analyse a concurrent program and generate a *Contract for Concurrency*, which identify the sequences of statements that must be executed atomically in a concurrent program. This method can be used either as documentation, for aiding development, or by an automatic tool to verify atomicity violations. *Gluon* is a static analysis tool that verifies such contracts for Java programs. Manually generating contracts for a large-scale codebase is unfeasible, and *Gluon* lightens this burden by providing an automated but rudimentary contract generation functionality. In this paper we propose a set of heuristics for automatically generating *Contracts for Concurrency*, and evaluate their accuracy with *Gluon*. With this new contract generation heuristics, we were able to fine-tune the contract generation, considerably reducing the number of spurious (unnecessary) clauses, which consequently reduced the analysis run time.

**Keywords:** Concurrency · Atomicity Violations · Automatic Contract Generation · Static Analysis · Software Verification.

## 1 Introduction

In the early 2000's there was a disruption in the programming paradigm due to the stagnation of processor clock frequency and subsequent increase in processor core count. To leverage from the added performance of multicore architectures, the use of concurrency has become a fundamental requirement of software development. Although capable of obtaining better performances, safe data manipulation under concurrency rests on the programmer's clairvoyance and proficiency in properly protecting access to critical regions. Failing in this task will most probably result in concurrency errors, such as atomicity violations. This problem becomes more noteworthy when we use services from third-party libraries,

---

<sup>\*</sup> Supported by Research Grants PTDC/CCI-COM/32456/2017 & LISBOA-01-0145-FEDER-032456.

where we possibly ignore the dependencies between services and cannot ensure that the library’s atomicity constraints are properly respected.

The *Contracts for Concurrency* [5] address the problem of identifying the dependencies between services in a software library or module, which in a concurrent setting, shall be executed atomically. *Gluon* [12] is a static analysis tool that takes as inputs a Java program module and a *Contract for Concurrency*, and outputs a verdict reporting if the program respects this same contract (i.e., if the invocations of the services identified as dependent in the contract are properly executed atomically). *Gluon* leverages its static analysis capabilities to generate rudimentary *contracts* from small to mid-sized Java legacy programs.

In this paper, we address the automatic generation of contracts for concurrent Java programs, by proposing and evaluating strategies and heuristics that address the major shortcomings of the current *Gluon*’s contract generator: the enormous number of contract clauses, the large number of false positives, and the massive validation time.

Our evaluation using the same benchmark used initially by *Gluon* confirms that our proposed heuristics produce a significantly lower number of clauses, producing considerable improvements on the analysis runtime and with equivalent or even better accuracy.

The upcoming Section presents the *Contracts for Concurrency* and the process behind *Gluon* analysis. §3 and §4 explores the proposed generation methodology and design. §5 contains the evaluation of the proposed heuristics using *Gluon*. Lastly, §6 presents the related work, ending with the conclusions in §7.

## 2 Contracts for Concurrency

A *Contract for Concurrency* [5] is a protocol expressed as a set of clauses that, for a given software module (program or library), identifies sequences of method calls that must be executed atomically (within a critical region). Each clause identifies one particular sequence (or more than one if using the *OR* operator) of methods that must be atomically executed, and is expressed as a star-free regular expression over the set of method names  $\Sigma_{\mathbb{M}}$  of the software module. In other words, a contract is a set  $\mathbb{R}$  of clauses, where each clause  $\rho \in \mathbb{R}$  is a star-free regular expression over the set of all public method names (i.e., API) of a software module  $m \in \Sigma_{\mathbb{M}}$ .

A contract violation occurs if any of the sequences represented by the contract clauses ( $\rho \in \mathbb{R}$ ) is interleaved with the execution of an other method from  $\Sigma_{\mathbb{M}}$  over the same object.

### 2.1 Example of a Contract

In order to illustrate the whereabouts of a *Contract for Concurrency*, consider the following relations between some well-known methods provided by the *Java standard library* (`java.util.List`):

---

**Example 1** Basic Contract for Concurrency.

---

```

1   add (contains | indexOf);
2   size get;

```

---

In Example 1, the first clause (line 1) states that, after a successful `add` operation, the item must be in the list. The trueness of this condition is trivial for a sequential program, but may not hold in a concurrent setting, where another thread may remove the newly added item just before its presence is verified. To disallow such behaviours, the call of the list methods `add` followed by `contains` must be executed atomically. The same applies to the call sequence `add` followed by `indexOf`, which must return a valid index for the inserted element and, thus, must be executed atomically as well.

The second clause (line 2) states that a call to a *List's* `size` method followed by `get` should successfully return the last element in the list. This may not occur in a concurrent setting if, in between those two calls, another thread removes the given element or appends yet another element to the list. Thus, the second clause in the contract declares that such method sequences must be executed atomically.

## 2.2 Verification of Contracts

The behaviour of a program can be seen as the join of the possible individual behaviours of all the threads the program may launch. To extract the usage of a module by a thread, *Gluon* starts by extracting its control flow graph (CFG) from the source code. From the CFG of a thread  $t$ , a context-free grammar  $G_t$  is built 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  $G_t$ .

**Definition 1 (Thread Grammar).** *A thread grammar  $G_t$ , which includes all the sequences of method calls executed by a thread  $t$  is defined by  $G_t = (N, \Sigma_M, P, S)$  where:*

- $N$ , is the set of non-terminal symbols.
- $\Sigma_M$ , is the set of terminal symbols, where  $N \cap \Sigma_M = \text{NULL}$ .
- $P$ , is the set of productions, such that  $P = N \rightarrow (N \cup T)$ .
- $S$ , is the starting symbol.

Intuitively, the grammar  $G_t$  represents the control flow of the thread  $t$ , ignoring everything not related with the module's usage.

Algorithm 1 presents the pseudocode of the approach for verifying a *Contract for Concurrency* [5].

The algorithm iterates over the program's threads (line 1). For each thread  $t$ , it first generates a grammar  $G_t$  (as described above) that captures the CFG of  $t$  (line 2). From  $G_t$ , a grammar  $G'_t$  describing all sub-words of the words generated

---

**Algorithm 1** Contract verification algorithm.

---

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

```

1: for  $t \in \text{threads}(P)$  do
2:    $G_t \leftarrow \text{make\_grammar}(t)$ 
3:    $G'_t \leftarrow \text{subword\_grammar}(G_t)$ 
4:   for  $w \in C$  do
5:      $T \leftarrow \text{parse}(G'_t, w)$ 
6:     for  $\tau \in T$  do
7:        $N \leftarrow \text{lowest\_common\_ancestor}(\tau, w)$ 
8:       if  $\neg \text{run\_atomically}(N)$  then
9:         return VIOLATION
10: return OK

```

---

by  $G_t$  is obtained (line 3). 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  $m.a(); m.b(); m.c();$ ,  $G'_t$  allows recognizing a contract with a clause containing only  $\rho = bc$ .

The algorithm subsequently iterates over contract clauses  $\rho \in \mathbb{R}$  (line 4) and handles them one-by-one. To see whether a thread  $t$  may generate a contract clause  $\rho$ , representing a call sequence, it is enough to parse  $\rho$  in  $G'_t$  (line 5). 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 6). In particular, the parsing trees contain information about the location of each of the calls of contract  $\rho$  in the program. Thus, 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  $\rho$  in the parsing tree (line 7).

The algorithm then checks whether the lowest common ancestor is always executed atomically (line 8) to make sure that the whole sequence of calls is executed under the same atomic context. Since it is the lowest common ancestor, we are sure to require the minimal synchronization from the program. A parsing tree contains information about the program locations where a contract violation may occur, and so we provide the programmer with detailed instructions on where this violation may occur and how to fix it.

### 2.3 Contracts with Parameters

Contract clauses as described in §2.1 can easily produce false positives (incorrect contract violations). Thus, contract clauses' restrictiveness may be increased, by limiting their scope to simultaneous manipulations of the same object. In essence, symbols in contract clauses may restrain the clause scope by including

(anonymous) references to the data objects being used. Consequently, Example 1 can be rewritten as:

---

**Example 2** Contract for concurrency with parameters.

---

```

1   add(X) (contains(X) | indexOf(X));
2   X=size get(X);

```

---

From this section, we conclude that it is a good practice to adopt these types of contracts whenever possible, capturing in detail the relationships between calls that can be problematic and, therefore, excluding sequences of operations that do not constitute true atomicity violations.

### 3 Work Proposal

*Gluon* [12] is static analysis tool, which can verify if a Java program respects a given *Contract for Concurrency* that regulate the sequences of method calls needed to be executed atomically. *Gluon* is also capable of generating basic contracts by analysing the Java source code and applying simple heuristics. The success of the automatic generation of *Contracts for Concurrency* depends on the quality of the process of extracting and identifying method dependencies from the program under analysis.

Currently, *Gluon* analyses the method dependencies by examining the critical regions protected by atomic blocks, creating a contract clause when a sequence of methods is executed atomically more than once. Thus, a contract generated by *Gluon* is a set  $\mathbb{R}^g$  of clauses, where each clause  $\rho^g \in \mathbb{R}^g$  is a star-free regular expression over the set of all public method names (i.e., API) of a software module  $m \in \Sigma_M$ , such that every word  $w$  matching the clause ( $w \in \mathcal{L}(\rho)$ ) occurs at least twice inside atomic regions in the program. With its current contract generation heuristic, *Gluon* can handle and generate contracts for small toy examples. However, when used in large-scale programs, *Gluon* ultimately fails, being either unable to parse the project or producing unusable contracts with too many clauses, leading to very long runs and too many false positive violations. Thus, contracts for large-scale programs must be subject to human proofreading and filtering.

We propose a set of more reliable heuristics for automatic generation of *Contracts for Concurrency*, which can handle complex and fairly voluminous Java programs. Such heuristics will be implemented and evaluated in *Gluon*.

### 4 Approach & Design

The main idea behind the newly proposed heuristics arises from the insight that most critical regions are predictably protected by the existence of atomic blocks.

However, in larger-scale programs, some unduly protected regions may exist, resulting in atomic violations. As such, we consider that the greater the percentage of times a given sequence is already executed atomically, the more likely it is to be representative of a real dependency. Thus, the threshold adopted must reflect a balance between a high percentage of atomic executions and a sufficiently large interval for detecting certain improperly protected executions. If such threshold is too low, the analysis will tend to report too many false positives. If, on the other hand, the threshold is too high, cases of atomic violations are no longer displayed, and only the well-protected operations are shown. Consequently, *Gluon* loses its primary purpose, which is the verification/detection of atomic violations.

The implementation of this approach was made possible by relying on the *Soot* [13] framework, which directly analyses the Java bytecode of programs, thus allowing access to the *AST* (Abstract Syntax Tree) and consequently making it possible to infer certain conclusions of the program under testing. So, by manipulating this framework, we can understand which methods are executed atomically and, from there, obtain the information necessary to perform the appropriate checks for the creation of contracts. Furthermore, it is relevant to mention that our option for the use of *Soot* [13] was biased by its previous usage on *Gluon*, and it was reasonable to continue using the same framework, even though others, such as *ASM* [2], could have been adopted.

#### 4.1 Minimum Number of Atomic Executions Heuristic

The approach for the generation of contracts, adopted by *Gluon*, is based on the search of atomic blocks and on the extraction of the methods placed inside those blocks into a contract clause associated with the program under consideration.

For a better understanding of this technique, please consider the Example below, alluding to some operations executed atomically, followed by the respective contract.

---

**Example 3** Method containing an atomic block with several methods invoked

---

```

1     void f(Object o) throws Exception {
2         ...
3         synchronized(this) {
4             int index = list.indexOf(o);
5             list.remove(index);
6             Thread.sleep(100);
7         }
8         ...
9     }
```

---

---

**Example 4** Clause generated for the Example 3

---

```

1   indexOf remove sleep;

```

---

In large-scale programs with many atomic invocations, there are generated clauses for all methods invoked in atomic blocks, which results in an endless number of clauses and an excessive analysis time. So, a given clause might only be associated with one atomic execution, which in the context of this paper, we considered spurious and irrelevant, compromising the final analysis time of the program.

In order to address this problem, we may require a sequence to be executed atomically at least  $n$  times before giving rise to a clause, which results in the heuristic previously supported by *Gluon*. In cases where the number of atomic executions is greater or equal than two ( $n \geq 2$ ), we are sure that there is associated a repetition pattern, so these executions are probably correlated. Thus, we assume that those methods must always be executed atomically, and a clause is created.

Finally, we expect that the greater the number of atomic executions ( $n$ ) required, the smaller the number of clauses produced and, consequently, the shorter the analysis time. However, false negatives become more likely to occur when we ignore sequences that, even though not fulfilling the requirements, were somewhere atomically protected. Furthermore, this technique is not devoid of false positives, as some clauses may not be representative of real dependencies. Also, we only refer to existing dependencies by their name, ignoring whether or not the same object is being manipulated during the execution, which may not correspond to a valid transgression.

## 4.2 Threshold Heuristic Applied to the Whole Program

The concept of threshold, applied under the automatic generation of contracts, is defined as the need for every word  $w$ , matching the clause ( $w \in \mathcal{L}(\rho)$ ), to be executed atomically a certain percentage of times in the program source code. As previously mentioned, the higher this ratio, the more likely for  $w$  to be a real dependency. Thus, limiting the generation of contracts to those frequently protected throughout the program seems to be a worthy contribution to obtaining clauses more representative of the program's correlations.

The clauses resulting from this new heuristic,  $\mathbb{R}^b$ , are expected to be a subset of the ones presented in the section above and previously outputted by *Gluon* [12],  $\mathbb{R}^a$ , when requiring the same number of minimal executions,  $\mathbb{R}^b \subset \mathbb{R}^a$ .

## 4.3 Threshold Heuristic Applied to Program Modules

In some contexts, the previous heuristic can be too restrictive, compromising the capture of some dependencies. These situations are most likely when the program under testing is developed by various programmers and consists of multiple

third-party modules. In this case, several modules tend to have their dependent methods correctly protected, while others, due to the lack of knowledge of the dependencies, may have them improperly guarded or even unguarded.

As such, we propose a new heuristic based in the percentage of times an exact sequence is atomically executed inside a module. Thus, if a particular sequence of methods is recurrently protected in one program module, then it should probably continue to be executed atomically in the rest of the program, and so a contract clause is produced.

Furthermore, using this heuristic, without even recurring to a subsequent human analysis, can be a very useful attempt to understand the program dependencies, while saving plenty of time in large programs, since a massive amount of manual scrutiny could be replaced by this quick and simple procedure.

#### 4.4 Generation of Parameterized Contracts

Now that the generation of fewer clauses has been addressed, it is necessary to solve the existing problem of the analysis resulting in many false positives, which was one of the principal issues when analysing real-world concurrent programs [12], where dozens of violations encountered did not correspond to valid transgressions.

Thence, it is necessary to remember the concept of parameterized contracts, exposed in §2.3. Relying on these restrictive representations of dependencies ensures that only the same object is being manipulated. Therefore, the resulting analysis of the contracts is not compromised, producing the correct verdicts.

The work carried out this principle as a premise and sought to produce such restrictive contracts with return values and parameters.

#### 4.5 Partial Verification of Contracts

So far, the heuristics presented are based on the common principle that dependencies are extracted from atomic blocks. Furthermore, the work developed directly depends on the analysis and comparison of all the methods invoked within these blocks. However, relying on a complete match may not be feasible for contract generation, as for convenience, the programmer(s) may place more methods within these critical regions than those absolutely necessary, thus distorting the required sequence and causing false negatives.

This way, some real dependencies may not be generated and consequently not be considered in the program's analysis. To address this problem, we proposed a strategy that includes generating contract clauses with partial expressions, i.e., extracting the operations invoked in a given atomic block and combining them into smaller clauses, each covering only two methods while maintaining their order of execution. In essence, we expect this contribution to be beneficial in situations where the program presents large critical regions with multiple operations invoked.

For a better understanding of this new technique, please consider the Example 3, alluding to some operations executed atomically. The previously existing



technique required a match of the clause presented in Example 4, which now may be parameterized and expressed as the Example below:

---

**Example 5** Parameterized clause generated by a complete match

---

```
1 X=indexOf(_) remove(X) sleep;
```

---

Although, when applying this new technique the clauses produced would be the following ones:

---

**Example 6** Clauses generated by a partial match

---

```
1 X=indexOf(_) (remove(X) | sleep);
2 remove sleep;
```

---

With this approach, we can match with a potential critical region that has protected the method's `indexOf` followed by `remove`. Furthermore, it is still relevant to mention that by transitivity, the clause presented in Example 5 would continue to be obtained from the clauses of Example 6.

#### 4.6 Default Contract

In order to further improve the accuracy of *Gluon*, contracts were generated and analysed for a wide variety of programs, and we observed that there is a set of clauses from well-known libraries whose use extends to almost all large-scale software. So, we decided to create a default contract containing parameterized clauses, allowing the user to quickly check whether these procedures are being complied with or not.

In short, we believe that, by providing various forms of contract generation, it is possible to build a more robust tool capable of being used under different heuristics and adjusted to the distinct designs of the programs under testing. Lastly, *Gluon* aims to produce fewer contracts, although more representative of the existing dependencies.

## 5 Evaluation

In this section, we expose and analyse some experimental results of automatic contract creation, in the following programs: `Cassandra 2.0.9`, `Derby 10.10.2.0`, `Lucene 4.6.1`, `OpenJMS 0.7.6`, and `Tomcat 6.0.41`. These same programs and

versions have already been studied in [12], so we can more easily compare and contrast our results with these previous ones.

As a way of questioning the assumptions used in this work, with respect to the need to create fewer clauses and reduce the number of false positives, we used different heuristics to obtain the set of dependencies that each of them would find, requiring the sequence to be executed atomically at least once.

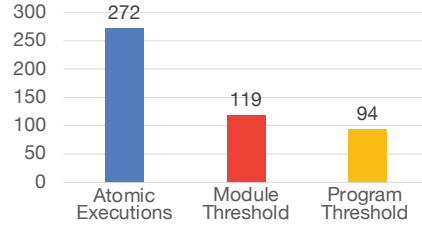


Fig. 1: Number of clauses generated for **Cassandra 2.0.9** with different heuristics.

To better understand the subsequent results, let us discuss Figure 1, where *Without Threshold* refers to the previous heuristic used by *Gluon*, only based on the number of times a sequence is executed atomically, as described in §4.1; *Module Threshold* refers to the generation of clauses based on the percentage of times (75%) the arrangement of methods is executed atomically within a given module, as explained in §4.3; and *Program Threshold* references to dependencies that run atomically across the entire project more than 75% of the time, as discussed in §4.2.

Besides getting a substantial decrease in the number of dependencies found when using the new heuristics, we can see that the number of correlations to be analysed, continues to be high, which would still take a considerable time to examine. Thus, we evaluated the impact of the minimum number of atomic executions on the number of clauses created, by testing the amount of clauses generated using different heuristics, presenting the results in Figure 2. It seems that assuming as an axiom that a given dependency is performed at least twice atomically can be convenient, reducing the number of clauses generated by more than 80%. As such, we will assume that, if the same sequence of calls is executed atomically in two different program locations, it may indicate that these methods are related and should always be executed atomically, and the corresponding contract cause will be generated.

The next step was to assess the efficacy of our new heuristics, which is depicted in Figure 3. These results are supported by contracts automatically generated by the *Gluon* tool, which considered the code from the *Java Library* itself, so it also includes the analysis of the critical regions within this library.

Our new heuristics should not compromise the quality of the analysis undertaken by the *Gluon* tool. Since the **Lucene** program had no reported bugs, reducing the number of clauses to be analysed would be beneficial, which was

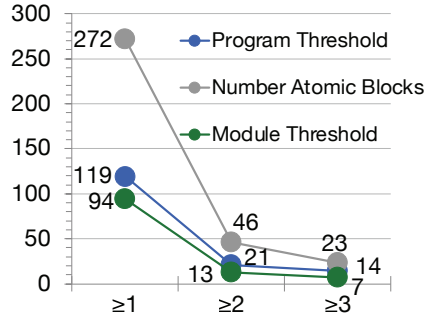


Fig. 2: Impact of the minimum number of atomic executions throughout the different heuristics used for **Cassandra 2.0.9**.

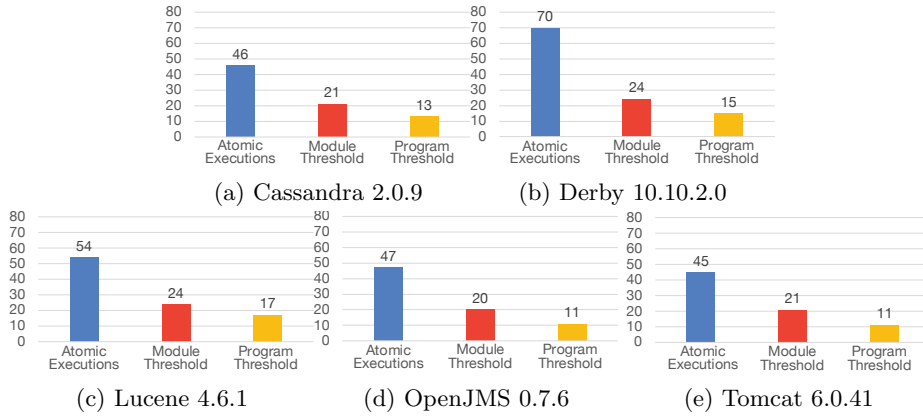


Fig. 3: Relation of the number of clauses throughout the different programs.

verified. For **Cassandra**, all violations would continue to be reported regardless of the heuristic used. In **Derby**, although the *Module Threshold* heuristic (§4.3) is able to capture the clauses representing the violations presented, using the missing heuristic, *Program threshold* showcased in the §4.2, this does not check. Nevertheless, in **OpenJMS** and **Tomcat**, only a few of the bugs could be detected, despite the heuristic used. However, it is necessary to mention that some of the clauses used to test this program in 2014, came from a manual generation. As such, a portion of the dependencies found would not be captured by the existing heuristics.

When skipping the analysis of the *Java Library*, we verified that finding the representative clauses of the bugs became far more complicated, generating less than half a dozen clauses for the new heuristics in each of the programs considered. Therefore, it was only possible to obtain similar results in **Lucene** (due to the absence of bugs) and **OpenJMS**. These results may indicate that the source code of the programs under analysis has large atomic blocks with several

invocations. So, total correspondence between these regions is difficult to occur, which contrasts with the code of the *Java Library*, which may have atomic blocks more compacted, allowing these to be determined with ease.

Given these results, it was necessary to use another way of capturing dependencies mentioned in §4.5. This new strategy subdivides the invocations inside an atomic block into smaller arrangements, thus requiring a less restrictive correspondence. Through this technique and not considering the *Java Library* classes, it was found that some clauses, lost in the results presented above, were identified again, particularly in the *Derby* program, where the reported bug would be found using any of the heuristics.

Combining this new technique with the analysis considering the code of the *Java Library*, the results obtained were similar to those presented before, except for the *OpenJMS* program, where it was possible to capture the missing clause. However, it is necessary to mention that the number of clauses generated has increased a lot (most of the time to the hundreds), making the subsequent analysis quite time-consuming and certainly unfeasible from the users' perspective. Thus, it is pertinent to mention that a quick analysis using the default contract can be a convenient alternative to test some of the most common dependencies discovered in concurrent Java programs. Therefore, the abundance of the atomic violations, which led to the bugs reported in the programs mentioned, would be captured using this alternative.

Nevertheless, we believe that the real benefit of this work stems from the combination of the techniques introduced, which now provide the user greater flexibility in the choice of various options that might make the most sense in the program under study. Besides, it is also particularly relevant to mention that the adopted threshold can be changed, thus directly affecting the number of clauses in the contract and the atomic violations *Gluon* will detect.

Along those lines, for the evaluation of the outcomes, a threshold of 75% was used after having carried out a variety of specific tests for the scrutiny of these programs, meaning that it may not always be the best value to adopt in other programs under analysis. For the *OpenJMS* program, it was verified when incrementing the threshold by 10% that at 85%, an atomic violation was no longer detected, thus compromising the accuracy of the tool. Similarly, for the *Cassandra* program, as well as for *Derby* and *Tomcat*, when adopting a threshold of 75%, most of the clauses corresponding to the detected bugs would appear. Thus, it was considered that the best value to adopt in this evaluation would be 75%. Although, it is also relevant to mention that these experiments also took into account the code and, consequently, the dependencies of the *Java Library*.

Now that the generation of fewer clauses resulting from the creation of new heuristics has been addressed, it is necessary to demonstrate the reduction in the analysis time of the clauses produced by the different techniques.

Figure 4 presents the time required for the analysis of the clauses generated by the different heuristics, for *Cassandra 2.0.9* (corresponding to Figure 3a). As expected, there was a significant improvement in the time required to check for atomic violations of the program under analysis, given its clauses. Thus,

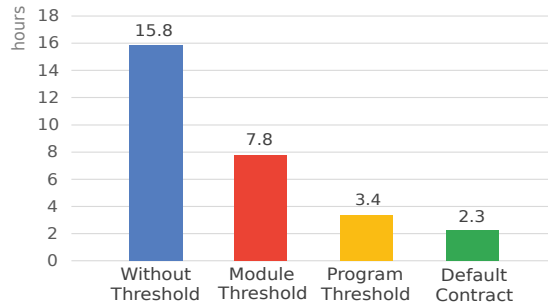


Fig. 4: Relation between the different heuristics and the time analysis, in hours, for *Cassandra 2.0.9*.

when comparing the outcome with the previously existing heuristic (§4.1), an improvement of approximately 50% is obtained while using the *Module Threshold* heuristic presented in §4.3. The improvement becomes even more significant whenever compared with the remaining heuristic, *Program Threshold* illustrated in §4.2, with a reduction of almost 80% of the time. Furthermore, when using the default contract presented in §4.6, it is possible to obtain a notable improvement, close to 85%, in the time required for the analysis of dependencies. As such, the value represented by this technique, as a quick and broad search for atomic violations, is reinforced.

Lastly, now that a reduction in analysis time has been addressed, it was necessary to improve another main obstacle, the quality of the results obtained, namely the number of false positives. As such, the clauses generated are now parameterized whenever possible, which notably contributes to reducing the number of false positives.

## 6 Related Work

The approach used in this work was based on the methodology of programming by contracts introduced by *Bertrand Meyer* [10] in 1992, based on defensive programming where the client and the provider are responsible for guaranteeing the consistency of the contract by defining the description of the class interface and its routines, through a set of pre and post-conditions.

R. Kramer provided Java support for these contracts, which until then were only available in the Eiffel language, and presented the *iContract* [7], which is a preprocessor capable of verifying if, in a given method, the constraints presented by the tags `@pre`, `@post`, and `@invariant` are respected. Later, other proposals of extensions to the DbC methodology appeared, including specification languages for modules in Java [8] describing its protocol through the use of Java annotations, and precompilers that dynamically test program annotations [4, 1].

Cheon et al. [3] propose an approach separating protocol from functional assertions and introducing a new specification clause referring to the order of

dependencies throughout method calls, using regular expression notations, allowing to specify the protocol that a given Java object should have. In order to support their work, they extended the *JML* [8] compiler to recognize the new clause at runtime, presenting a dynamic analysis tool. Later, this work was extended by *Hurlin* [6] to support concurrency scenarios using static analysis based on a proof theorem that tends to be quite limited due to the inefficiency of automatic program generation from contracts to prove their correctness.

Many works can be found on atomic violations, wherein Peng Liu et al. developed a tool (*ICFinder* [9]) that is similar to *Gluon*, relying on static analysis to infer incorrect module calls. Subsequently, the tool filters them using dynamic analysis, due to the generation of many false positives. However, it is noteworthy to mention, that the bugs reported by *Gluon* in the programs cited in the previous section, were not able to be detected by *ICFinder*.

Finally, given the lack of popularity of this type of methodology in the development of concurrent programs, we can't find much work done in the automatic generation of contracts. Furthermore, given the contract specification adopted in this tool, resulting only in a sequence of methods that need to be atomically executed, it becomes difficult to compare the work done with other software. As such, we hope this work will contribute, even if at a small scale, to the dynamization of this methodology, which has already been proven to yield good results.

## 7 Conclusion

In this paper, we explored the topic of automatic contract generation for concurrent programs in Java, improving the previous heuristics in both efficiency and efficacy, and ultimately making the *Gluon* tool more complete and reliable. Consequently, *Gluon* can now generate and analyse parameterized contracts and produce a verdict on whether or not atomic violations have occurred, promoting the production of robust and reliable software.

The methodology, based on the usage of different heuristics and the use of parameters representing the distinguishable objects on the execution flow, demonstrated, through the results obtained, that it is possible to generate fewer clauses without compromising the identification of dependencies and the accuracy of the tool.

This work offers room for further improvement, in the form of presenting even fewer clauses using parameterized contracts to detect the number of times a given sequence is atomically executed. From the point of view of analysing the occurrence of atomic violations, it is only essential to consider the case in which the same object is manipulated. Although, this situation is controlled *a posteriori* due to the generation of parameterized contracts and by a subsequent analysis that will only take into consideration the manipulation of the same object. Nonetheless, it could be an effective way to further restrict the number of clauses presented and make the creation of clauses more representative, as only

those are truly capable of producing real atomic violations since they manipulate the same object and, by definition, need to be atomically protected.

In addition, when analysing large programs, *Gluon* cannot scrutinize the whole program scope due to space state explosion, which results in an exponential increase of the corresponding time and resources used when exploring the parsing trees of the class combinations. As such, it would be interesting to modify the contracts, so it becomes possible to allude to the class where the method comes from, allowing us to refer to dependencies of multiple classes, despite limiting the analysis to the combination of the modules mentioned in the contracts.

Finally, it is rather important to mention that the large quantity of clauses obtained when using the technique exposed in §4.5, is due to the fact that all possible combinations for the operations invoked on atomic blocks are being taken into account. However, this factor could be optimized and simply create clauses for methods that handle the same object. This way, it would be possible to obtain a significant reduction in dependencies produced and consequently on the analysis time, leaving the capture of real dependencies uncompromised.

## Acknowledgments

This report is an extended version of the paper "Automatic Generation of Contracts for Concurrent Java Programs" [11] to be presented in INForum 2022. I would like to express my gratitude to Diogo Sousa, the initial developer of *Gluon*, and Filipe de Luna, who is currently working on his MSc dissertation, for their multiple suggestions, help, and permanent availability, which undoubtedly impacted the final outcome of this project. I also want to thank my advisers' Profs. João Lourenço and Jeremy Bradbury (Ontario Tech University, Canada) for their incredible support and mentoring throughout this work. From the first day, I was welcomed with open arms and provided with the best conditions so I could develop my skills and knowledge in this subject. Moreover, I would like to express my appreciation to Prof. João Lourenço for the immediate availability and opportunity to attend his MSc classes in Concurrency and Parallelism course, particularly relevant to the development of this work, where its influence was undoubtedly notable. I am very grateful to them for providing me with my first contact with research, which I will always bear in mind with great fondness.

## References

1. Bartetzko, D., Fischer, C., Möller, M., and Wehrheim, H.: Jass — Java with Assertions<sup>1</sup> <sup>1</sup>This work was partially funded by the German Research Council (DFG) under grant OL 98/3-1. *Electronic Notes in Theoretical Computer Science* 55(2), 103–117 (2001). DOI: [10.1016/s1571-0661\(04\)00247-6](https://doi.org/10.1016/s1571-0661(04)00247-6)
2. Bruneton, E., Lenglet, R., and Coupaye, T.: ASM: a code manipulation tool to implement adaptable systems. *Adaptable and extensible component systems* 30(19) (2002)

3. Cheon, Y., and Perumandla, A.: Specifying and checking method call sequences of Java programs. *Software Quality Journal* 15(1), 7–25 (2007). DOI: [10.1007/s11219-006-9001-4](https://doi.org/10.1007/s11219-006-9001-4)
4. Corbett, J.C., Dwyer, M.B., Hatcliff, J., and Robby: A Language Framework for Expressing Checkable Properties of Dynamic Software. In: *SPIN Model Checking and Software Verification*, pp. 205–223. Springer Berlin Heidelberg (2000). DOI: [10.1007/10722468\\_13](https://doi.org/10.1007/10722468_13)
5. Dias, R.J., Ferreira, C., Fiedor, J., Lourenço, J.M., Smrcka, A., Sousa, D.G., and Vojnar, T.: Verifying Concurrent Programs Using Contracts. In: *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pp. 196–206 (2017). DOI: [10.1109/ICST.2017.25](https://doi.org/10.1109/ICST.2017.25)
6. Hurlin, C.: Specifying and checking protocols of multithreaded classes. In: *Proceedings of the 2009 ACM symposium on Applied Computing - SAC '09*. ACM Press (2009). DOI: [10.1145/1529282.1529407](https://doi.org/10.1145/1529282.1529407)
7. Kramer, R.: iContract-the Java/sup TM/ design by Contract/sup TM/ tool. In: *Proceedings. Technology of Object-Oriented Languages. TOOLS 26 (Cat. No.98EX176)*. IEEE Comput. Soc. DOI: [10.1109/tools.1998.711021](https://doi.org/10.1109/tools.1998.711021)
8. Leavens, G.T., Baker, A.L., and Ruby, C.: Preliminary design of JML. *ACM SIGSOFT Software Engineering Notes* 31(3), 1–38 (2006). DOI: [10.1145/1127878.1127884](https://doi.org/10.1145/1127878.1127884)
9. Liu, P., Dolby, J., and Zhang, C.: Finding incorrect compositions of atomicity. In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2013*. ACM Press (2013). DOI: [10.1145/2491411.2491435](https://doi.org/10.1145/2491411.2491435)
10. Meyer, B.: Applying 'design by contract'. *Computer* 25(10), 40–51 (1992). DOI: [10.1109/2.161279](https://doi.org/10.1109/2.161279)
11. Pereira, H.G., Sousa, D.G., Bradbury, J., and Lourenço, J.M.: Automatic Generation of Contracts for Concurrent Java Programs. In: *INForum 2022 - Atas do 14<sup>o</sup> Simpósio de Informática (2022)*
12. Sousa, D.G., Dias, R.J., Ferreira, C., and Lourenço, J.M.: Preventing Atomicity Violations with Contracts. (2015). DOI: [10.48550/ARXIV.1505.02951](https://doi.org/10.48550/ARXIV.1505.02951)
13. Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., and Sundaresan, V.: Soot - a Java Bytecode Optimization Framework. In: *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research. CASCON '99*, p. 13. IBM Press, Mississauga, Ontario, Canada (1999)