Resource Contracts for Java

Rody Kersten
CMU Silicon Valley
rody.kersten@sv.cmu.edu

Martin Schäf
SRI International
martin.schaef@sri.com

Temesghen Kahsai
CMU Silicon Valley / NASA Ames
teme.kahsai@sv.cmu.edu

ABSTRACT

Writing specifications about program behavior is hard. Writing specifications about non-functional effects such as resource usage is often even harder. If manually instrumenting the program is not an option, programmers have to rely on comment-based specification languages like JML to introduce ghost variables and other fairly abstract concepts that are complicated and hard to maintain. Even worse, most static analysis tools nowadays operate on bit- or bytecode and cannot process those type of specifications.

To address this problem, we propose a library-based specification formalism for time complexity in Java. The approach is inspired by the success of assertion libraries like CodeContracts and Guava. Our library provides a set of methods and enumerated types that allow the user to write complexity assertions in a 'human' readable form and without instrumenting the code. On the backend, we provide a bytecode rewriting tool that uses these assertions to automatically instrument the program with counter variables. The transformed program can then be checked via testing or off-the-shelf automated static analyzers.

Keywords

Complexity Analysis; Resource analysis; Code contracts

1. INTRODUCTION

Annotating code with specification information is a hard problem. While the benefit of providing additional information about source code behavior for maintenance, testing and static analysis is evident, most modern languages provide no or only limited support for adding such specifications. In most languages, assertions are the only means to express additional semantic information about the program state at a certain point.

Specification languages such as JML [18] or ACSL [3] are declining in popularity as more and more analysis tools operate on intermediate representations like Java bytecode or LLVM bitcode which do not have access to the source comments. Also, these languages usually are not supported directly by IDEs and compilers, which means there is no direct support for amenities like auto-completion or on-the-fly syntax checking which usually results in poor maintenance of the specification as the code changes (as with all other comments).

As an alternative to these comment-based specification languages we have seen an increase in popularity of specification libraries like Code Contracts [8], or more light-weight alternatives like Google's Guava[1], Apache Commons[2], or AssertJ[3]. These libraries provide a set of static methods to express more advanced assertions, like postconditions, invariants or even universal quantifiers. That means the specification is integrated in the actual code, which improves the chances that it gets maintained as code changes. Another benefit is that these libraries usually provide a switch to distinguish between a debug mode where violated properties raise exceptions and a release mode where the specification statements simply have no effect. Further, static analyzers that operate on bit- or bytecode can easily pick up these specification languages since they turn into static method invocations.

The downside of library-based specification languages is that their expressiveness is limited by what the programming language allows. Ghost variables, existential quantifiers, and many other constructs are not easy to model with these libraries.

In this paper, we discuss how we can build a flexible library-based specification language for Java that allows us to specify time complexity of algorithms. This is motivated by the increasing demand for checking for security vulnerabilities in applications, such as denial-of-service or side-channel attacks where an algorithms exceed their expected running time for certain inputs and thus not do not provide the necessary service or reveal information that is supposed to be private.

Checking complexity of an algorithm usually requires instrumenting the program with counters and then asserting that the counter is within a certain bound. However, as discussed earlier, comment-style specification languages are not an option for us, library-based specification languages do not provide support for ghost variables, and we do not want to add variables or counters to our program either (because that would require work by the developer and might be tricky to maintain). Instead, we provide specification constructs to talk about the size of collections, arrays, and other frequently used Java concepts, and methods to refer to the cost of the code within a certain scope. These constructs allow us to state complexity of algorithms as assertions in an intuitive and human readable way.

The intended usage model for our complexity assertions is as follows: during normal execution of the program, these assertions have no effect (i.e., always evaluate to true). For debugging, we describe a procedure that scans the bytecode for these complexity assertions and rewrites the bytecode by introducing counter variables and proper assertions that speak about these counters as well as debug printing support. The transformed bytecode can now be either passed to an assertion checker like JayHorn [14] to prove that the upper bound is always respected, or it can be run (on random inputs) to collect profiling information about expected and actual complexity.

[1] `https://github.com/google/guava`
[2] `https://commons.apache.org/`
[3] `http://joel-costigliola.github.io/assertj/`

We exercise all steps of our approach using a case study and demonstrate that the approach is able to handle non-trivial examples. We outline how this approach can be used to connect tools that estimate complexity bounds with assertions checkers to obtain an interesting new class of security checkers. Finally, we discuss the limitations of the current approach and the future directions we may take to make such a library usable.

2. RUNNING EXAMPLE: HASH TABLE

We will use an implementation of a hash table as a running example. An excerpt of its source code is shown in Listing 1.

```
1  public class HashTable {
2    private BucketNode[] mTable;
3
4    // ...
5
6    private Entry findEntry(final String s) {
7      for (Entry e : mTable[hash(s)]) {
8        if (e.key.equals(s))
9          return e;
10     }
11     return null;
12   }
13
14   // ...
15
16   class BucketNode
17       implements java.lang.Iterable<Entry> {
18     Entry e;
19     BucketNode next;
20
21     // ...
22   }
23 }
```

Listing 1: Excerpt of a hash table implementation.

A hash table is a mapping from keys to values in which values are stored in a certain *bucket*, depending on their hash value. The idea is that, when a good hash function is used, elements are distributed approximately uniformly over the buckets. Since the number of buckets is finite, collisions occur, where multiple elements will be stored in the same bucket. To handle such cases, the bucket itself usually contains a data-structure that can hold multiple key-value pairs. In our running example, each bucket contains a linked list of pairs.

When adding a key-value pair to the hash table, finding out which bucket to use is $\mathcal{O}(1)$. However, look-up in the linked list for that bucket requires $\mathcal{O}(n)$ key comparisons. As in our example keys are strings, we can be more precise. The `equals` method for strings iterates over the characters in both strings (in case the strings are of equal length). If a character is found that is not equal, it returns `false`. The worst case for this method is when either the strings are equal or only the last character differs. The worst case for look-up in our hash table is when all elements end up in the same bucket because of collisions and all keys are equal to the input key except for the last character (and have equal length). In this case, look-up takes $n \cdot k$ character comparisons, where n is the number of elements in the hash table and k is the length of the keys.

3. ASSERTING RESOURCE BOUNDS

Assume that we have a given resource usage bound in the form of a function over data sizes. Such a bound can be determined manually by the programmer, inferred using a tool such as COSTA [1] or learned by a symbolic execution engine. For our example in Listing 1 we want to state that the cost of executing `findEntry` is less than or equal to the size of `mTable[hash(s)]` times the length of the key string `s`.

To allow a programmer to express this contract, we provide a class `RC` that contains a set of `enum` types and static methods to refer to the size of various types (e.g., Collections, arrays, String, etc). The programmer can now use `RC` to state the complexity assertion for `findEntry` as follows:

```
RC.contract(RC.MethodCost(s) <=
RC.size(mTable[hash(s)]) * RC.size(s));
```

That is, the programmer can use `RC` to state the resource bound in a fairly human-readable fashion using standard language constructs and without introducing counter variables. This approach is inspired by the way programmers can specify method postconditions in CodeContracts [8]. The implementation of `RC` is simply an empty shell, i.e. this statement does not change the execution of the program. That is, the call to `RC.contract` could be optimized away during compilation.

To actually check this contract, we describe a procedure that scans the bytecode of a program for uses of `RC` and instruments it with the necessary counter variables and assertions. The call to `RC.MethodCost(s)` in the contract tells us a couple of things. First, it allows us to specify a scope for which to track resource consumption (the enclosing method), adding the context of the call stack to allow differentiation of resource contracts. Second, the s parameters tells us that iteration over the string s is what we want to track. While the length of the list obviously contributes to the total costs, this is reflected in the total iterations over s implicitly and should not be double counted.

Consider the running example with two contracts added in Listing 2. The first contract checks the number of unwindings of the `BucketNode` list only. The second contract checks the total consumption in terms of iterations over string characters in `String.equals`. Specifying the scope as `Method` allows to create a mapping from objects to usage accumulators instead of having just one global accumulator where it would not be possible to combine both these checks in a single program.

```
1  import RC;
2
3  public class HashTable {
4    private BucketNode[] mTable;
5
6    // ...
7
8    private Entry findEntry(final String s) {
9      RC.contract(RC.MethodCost(mTable[hash(s)]) <=
10       RC.size(mTable[hash(s)]);
11
12     RC.contract(RC.MethodCost(s) <=
13       RC.size(mTable[hash(s)]) * RC.size(s));
14
15     for (Entry e : mTable[hash(s)]) {
16       if (e.key.equals(s))
17         return e;
18     }
19     return null;
20   }
21
22   // ...
23 }
```

Listing 2: The hash table example, now with added resource contracts.

We show an excerpt from the RC class in Listing 3. It consists of a collection of cost methods for different scopes, size methods for various data structures, methods to update the costs, and a method to assert the contract.

```
1  import java.util.HashMap;
2
3  public class RC {
4
5    static HashMap<Object,Integer> costMap =
6      new HashMap<Object,Integer>();
7
8    static int MethodCost(Object o) {
9      return costMap.get(o);
10   }
11
12   // ... similar cost methods for other scopes
13
14   static int size(Iterable<?> i) {
15     int size = 0;
16     for (Object o : i)
17       size++;
18     return size;
19   }
20
21   static int size(String s) {
22     return s.length();
23   }
24
25   // ... similar size methods for arrays
26
27   static int inc(Object o, int cost) {
28     return costMap.put(o, costMap.get(o) + cost);
29   }
30
31   // ... more cost update methods
32
33   static void contract(boolean b) {
34     assert b;
35   }
36 }
```

Listing 3: Excerpt from the RC class, which provides various methods for intuitive description of program costs.

In the following we discuss the necessary steps of our tool to instrument bytecode to track resource consumption based on these contracts.

4. BYTECODE INSTRUMENTATION

In the previous section, we have seen how the RC can be used to write contracts about time complexity of a method. In this section, we discuss how we can use these contracts to rewrite the program and introduce counter variables and Java assertions that can be checked by the JVM or a static analysis tool.

As discussed, contracts written using our class RC have no semantics when the program is executed. However, they provide us with necessary information about which components need to be instrumented with counters. To perform this instrumentation, we use Soot [4]. This bytecode transformation library provides several utilities to parse, analyze and rewrite bytecode such as points-to analysis and various types of data-flow analyses.

We will discuss how we rewrite resource contracts written with RC using the two contracts in Listing 2. The first contract starts with a call to RC.MethodCost(mTable[hash(s)]). This indicates that the scope of the contract is the enclosing method and we want to add the final resource assertion to the end of it. It also tells us that it is iterations over the list in mTable[hash(s)] that we want to track. Our bytecode rewriting introduces a fresh static counter variable for this method (more specifically for the assertion) and initializes it to zero at the beginning of the method body.

Currently, this is the only scope that we support but others, e.g., RC.Block, can easily be added. Next, the assertion contains a call to RC.size. Here, the method for arguments of type Iterable is called, because mTable[hash(s)] is of type BucketNode which implements Iterable.

To achieve this, our bytecode rewriting scans for all interactions with the iterator of mTable[hash(s)] that are reachable from the enclosing method (because we set the scope with RC.MethodCode). Each of these interactions with the iterator is then replaced by a method that mimics the behavior of the original iterator and also increases the static counter that we have introduced earlier. Note that this transformation does not rewrite the actual iterator of BucketNode because that would also count up the cost if we iterate over other objects of that type different from mTable[hash(s)]. Instead it replaces only those iterator interactions where the base points to the same address as mTable[hash(s)]. This requires a bit more in-depth reasoning (and may not be precise because aliasing is not decidable in the presence of many Java constructs).

Currently, we only increase the cost by one in each iterator step. In the future, we plan to support more flexible cost models, where user provided cost functions can be taken into account.

The second contract contains two calls to RC.size. The left hand side, RC.MethodCost(s), specifies that it is the cost w.r.t. string s that we are after, which means it is iteration over the characters in s that should be counted. The contribution of the length of the list to the total costs is counted implicitly, as this determines the number of calls to String.equals.

The String class is a bit more complicated to instrument, because it keeps an array of characters internally and does not implement the iterator interface. In theory, we could automatically instrument the interactions with s in the same way we did for iterators. However, since String is a frequently used library class, we treat it as a special case for efficiency: we replace the type s by a subtype of String that counts up our cost variable in all relevant methods (e.g., equals, replaceFirst, etc).

Now we have instrumented all places where we need to increment our cost counters. Hence, our bytecode instrumentation can now add actual assertions before each exit of the findEntry in the hash table example, that ensure that our cost variables are less than or equal to mTable[hash(s)].size() for the first contract and mTable[hash(s)].size() ∗ s.length() for the second contract. These assertions can now be checked using a static checker as shown in the next section. Alternatively, we could also instrument the code with logging statements to track the actual cost when executing the bytecode.

Note that this approach is still in an early stage and has several limitations. First and foremost our instrumentation is limited by the problem that we can only approximate code that is reachable from within a method (this limitation applies to all types of static analysis for Java). Second, focusing on iterators might not be sufficient to count costs. Just as in the case of the String class, programmers might not implement the iterator interface and use custom data structures. In theory, this case can be handled by using a taint analysis to identify loops whose exit condition depends on variables for which we want to count cost. This is part of future work.

[4] http://sable.github.io/soot

5. CONTRACT PROOFS WITH JAYHORN

Now that we have generated an executable program that is instrumented to keep track of its own resource consumption, we can use an off-the-shelf assertion prover to verify the resource contract. One tool that is very suitable for this purpose is JayHorn [14], as it works directly on Java bytecode, is light-weight, open source and extendible. Moreover, it provides a good platform to implement the required code transformations in. As JayHorn is still under development, it does not currently support the full instrumented program as described above. We were, however, able to prove a simplified version of the running example, motivating further research in this area.

On an abstract level, JayHorn takes a Java program with one or more assertions as input and outputs "SAFE" in case it can prove that the assertions hold or "UNSAFE" otherwise. It works by translating the Java code to Constrained Horn Clauses (CHCs) in several steps, which are then passed to a solver, e.g. Z3 [6] or Eldarica [24].

Most of the program transformations in JayHorn are done using Soot. The first step is a series of behavior-preserving transformations such as elimination of exceptional flow, de-virtualization and control-flow simplification. Since they are behavior-preserving, correctness of the applied transformations can be tested using Randoop [21]. Next, JayHorn applies a series of sound abstractions to memory access, arrays and library calls. The program is then in a simplified intermediate representation that can be translated directly to CHCs.

Note that, since JayHorn mostly consists of a collection of code transformations using Soot, it is the perfect platform for implementation of the presented procedure. Particularly, it has infrastructure that allows comparison of Randoop test results of the transformed code against those for the original. As such program transformations are potentially buggy or imprecise (e.g. due to aliasing of data structures within a method), this can be used to increase confidence in their correctness, similar to [27]. In the case of resource contracts, it could also be useful to compare resource consumption against the specified bound dynamically. If any measured consumption is higher, we have very cheaply found a counter-example to the resource contract. If all are much lower, this could signify that the given resource bound over-approximates a lot and could potentially be made more precise.

6. RELATED WORK

The proposed approach in this paper draws inspiration from different fields such as *behavior-driven development*, *contract-based development*, *specification languages* and *termination* analysis of imperative programs.

The notion of a *contract* has a long history in software engineering and traces its roots to rely-guaranteed approaches introduced by Hoare, Dijkstra and others [7, 11, 13]. It is adopted in earnest in the *design by contract* methodology [12, 20], which has been applied in different areas of software development and verification. Newer programming languages such as Dafny [19] incorporate formal contracts and compile-time contract checking as native features. Formal contracts have also been integrated into popular programming languages, via the addition of *ad hoc* specification languages, e.g., ACSL [3] for C/C++, JML [18] for Java, or SPARK [2] for Ada. ACSL in particular has a notion of *behavior* in function contracts.

Several techniques have been proposed for the verification of *resource bounds*, *termination* and other *liveness* properties of imperative programs. A leading methodology in forming a termination argument for imperative programs is the synthesis of *ranking functions*, i.e. in any potentially looping computation, prove that some well-founded metric strictly decreases every time around the loop. In fact, state of the art termination checkers for imperative programs compose termination arguments by repeatedly invoking ranking function synthesis tools (e.g. [4, 5, 22]). Ranking functions can be crucial in calculating bounds on runtime or usage of resources such as heap-space [17] or energy [15]. Recently, a new approach [28] has been developed for the (non)-termination analysis based on a safety verifier that uses Horn clauses as intermediate verification language.

A sheer number of approaches has been proposed in the literature for the inference of resource bounds e.g., [9,10], including several for which a guess-and-check approach is suggested [1, 16, 25, 26]. Resource contracts could provide a way to express and check bounds inferred by these tools. Another field in which resource bounds are of importance is worst-case execution time (WCET) analysis. An overview can be found in [29]. Our approach is orthogonal to this field, as we allow straight-forward expression of bounds that could be inferred and/or checked by such tools.

7. CONCLUSIONS AND FUTURE WORK

We have proposed a first step towards a library-based specification language for resource usage bounds in the spirit of code contracts. Using our `RC` class, a programmer can write resource specifications in a compact, human-readable, and easy to maintain way, without instrumenting the program with helper variables.

The approach is very light-weight in that library methods do not have a semantics and thus they do not affect the execution of the original program.

For checking the specification we present a bytecode transformation to instrument a given program with counters and runtime checks based on the resource specifications found in the (byte)code. After applying this transformation, the specification can be checked using off-the-shelf testing or verification tools.

Using library methods as a specification language has the additional benefit that it is easy to incorporate complexity assumptions generated by other tools. For example, symbolic execution could be used by Symbolic PathFinder [23] to infer an upper bound for the complexity of our method and add an assertion to the source code. Alternatively, JayHorn could synthesize a resource constraint on the bytecode level and add it directly to the transformed class file. This eliminates a lot of painful transformation work between source- and bytecode representation of objects and variables.

While the steps presented in this paper work on actual examples, this is only a preliminary result. The design of our `RC` class is not finalized and its expressiveness is currently limited to what we have presented in this paper. Also, the bytecode transformation used in Section 4 is by no means the only or best way of turning specifications into assertions.

Our first experiments suggest that the idea is promising, but the road to a usable tool is still long. We hope to start a discussion with the community on how such a resource specification language should look and what its capabilities and limitations should be in terms of expressiveness.

8. ACKNOWLEDGEMENTS

This work is funded in parts by AFRL contract No. FA8750-15-C-0010. Furthermore, this material is based on research sponsored by DARPA under agreement number FA8750-15-2-0087. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

9. REFERENCES

[1] E. Albert, R. Bubel, S. Genaim, R. Hähnle, G. Puebla, and G. Román-Díez. Verified resource guarantees using COSTA and KeY. In *PEPM '11*, pages 73–76. ACM, 2011.

[2] J. Barnes. *High Integrity Software, The SPARK Approach to Safety and Security*. Praxis Critical Systems Limited, 2006.

[3] P. Baudin, P. Cuoq, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto. *ACSL: ANSI/ISO C Specification Language, ver. 1.5*, 2010.

[4] A. R. Bradley, Z. Manna, and H. B. Sipma. Linear ranking with reachability. In *CAV '05*, pages 491–504. Springer, 2005.

[5] M. Colón and H. Sipma. Synthesis of linear ranking functions. In *TACAS '01*, pages 67–81, 2001.

[6] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS '08*, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[7] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[8] M. Fähndrich and F. Logozzo. Static contract checking with abstract interpretation. In *FoVeOOS '10, Revised Selected Papers*, pages 10–30. Springer, 2011.

[9] B. S. Gulavani and S. Gulwani. A numerical abstract domain based on expression abstraction and max operator with application in timing analysis. In *Computer Aided Verification*, LNCS 5123, pages 370–384. Springer, 2008.

[10] S. Gulwani. SPEED: Symbolic complexity bound analysis. In *CAV '09*, pages 51–62. Springer, 2009.

[11] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 583, Oct. 1969.

[12] J. Jézéquel and B. Meyer. Design by contract: The lessons of Ariane. *IEEE Computer*, 30(1), 1997.

[13] C. Jones. *Development Methods for Computer Programs including a Notion of Interference*. PhD thesis, Oxford University, 1981.

[14] T. Kahsai, P. Rummer, H. Sanchez, and M. Schaf. JayHorn: A framework for verifying Java programs. In *CAV '16*, 2016.

[15] R. Kersten, P. Parisen Toldin, B. van Gastel, and M. van Eekelen. A Hoare logic for energy consumption analysis. In *FOPARA '13*, LNCS 8552, pages 93–109. Springer, 2014.

[16] R. Kersten, B. van Gastel, O. Shkaravska, M. Montenegro, and M. van Eekelen. ResAna: a resource analysis toolset for (real-time) JAVA. *Concurrency and Computation: Practice and Experience*, 26(14):2432–2455, 2014.

[17] R. W. J. Kersten, O. Shkaravska, B. E. van Gastel, M. Montenegro, and M. C. J. D. van Eekelen. Making resource analysis practical for Real-Time Java. In *JTRES '12*, pages 135–144, 2012.

[18] G. T. Leavens, A. L. Baker, and C. Ruby. JML: a java modeling language. In *Formal Underpinnings of Java Workshop (at OOPSLA '98)*, 1998.

[19] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR '16*, LNCS 6355, pages 348–370. Springer, 2010.

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

[21] C. Pacheco and M. D. Ernst. Randoop: Feedback-directed random testing for java. In *OOPSLA '07*, pages 815–816. ACM, 2007.

[22] A. Podelski and A. Rybalchenko. A Complete Method for the Synthesis of Linear Ranking Functions. In *VMCAI*, pages 239–251, 2004.

[23] C. S. Păsăreanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape. Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In *ISSTA '08*, pages 15–26, 2008.

[24] P. Rümmer, H. Hojjat, and V. Kuncak. Disjunctive interpolants for Horn-clause verification. In *CAV'13*, pages 347–363. Springer, 2013.

[25] R. Sharma, S. Gupta, B. Hariharan, A. Aiken, P. Liang, and A. Nori. A data driven approach for algebraic loop invariants. In *Programming Languages and Systems*, LNCS 7792, pages 574–592. Springer, 2013.

[26] O. Shkaravska, R. Kersten, and M. Van Eekelen. Test-based inference of polynomial loop-bound functions. In *PPPJ'10*, pages 99–108. ACM, 2010.

[27] G. Soares, R. Gheyi, and T. Massoni. Automated behavioral testing of refactoring engines. *IEEE Trans. Softw. Eng.*, 39(2):147–162, Feb. 2013.

[28] C. Urban, A. Gurfinkel, and T. Kahsai. Synthesizing ranking functions from bits and pieces. In *TACAS '16*, pages 54–70, 2016.

[29] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem – overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, May 2008.