UCSB Computer Science Technical Report 2014-09

SuperContra: Cross-Language, Cross-Runtime Lightweight Contracts As a Service

Stratos Dimopoulos · Chandra Krintz · Rich Wolski · Anand Gupta

14-04-15

Abstract Despite the potential of Design-by-Contract (DbC) for increasing the reliability and robustness of software, it has to date experienced only limited use. One reason for this, is that each DbC solution targets a single language (or language run-time) and thus employs different syntax and implementations for specifying contracts for each. Given that modern applications increasingly employ multiple components, each written in multiple programming languages, writing contracts imposes significant burden on programmers.

To reduce this overhead and to encourage greater use of DbC as part of both testing and production systems, we present a new DbC framework called *SuperContra* that implements and enforces lightweight contracts across different programming systems, as-a-service. *SuperContra* is unique in that developers employ a familiar, high-level language to write contracts regardless of the programming language used to implement the component under test. Moreover, contract evaluation occurs as-a-service, as opposed to at each client, simplifying clients and facilitating multi-client contract auditing. We evaluate *SuperContra* using widely used, open-source software and compare its performance against existing DbC frameworks. Our results show that *SuperContra* performs on par with non-service-based DbC approaches and in some cases similarly to code running without contracts.

Keywords software reliability, DbC, behavioral software contracts, cross-language contracts, cross-runtime contracts, lightweight contracts, contracts as a service

1 Introduction

The level of complexity in modern software systems has significantly increased. A single software system often includes a large number of cooperating compo-

Stratos Dimopoulos
Department of Computer Science, University of California at Santa Barbara
552 University Road, Santa Barbara CA 93106
E-mail: stratos@cs.ucsb.edu

Chandra Krintz E-mail: ckrintz@cs.ucsb.edu · Rich Wolski E-mail: rich@cs.ucsb.edu

nents, developed with multiple programming languages. This system is, usually, the outcome of the collaborative work of several engineers who utilize different programming frameworks, according to their preferences. In this context, it is increasingly important to detect bugs early in order to increase software reliability and robustness.

A mature software development methodology that aims to increase software reliability by eliminating bugs and improving exception handling is Design by Contract[1] (DbC) (Meyer 1992). DbC specifies the intended behavior of a software component or a web service as an interface contract between a client and a supplier. At the heart of a DbC contract are assertions. Assertions are expressed on the code as pre- and post-conditions of methods and class invariants; their violation indicates bugs in the client or the service provider code.

With assertions being part of the code, the onus is on developers to write the contracts. Since web-based systems expose their functionality through multiple application programming interfaces (APIs) to support several, multi-language client programs, developers must learn multiple DbC specification languages and be familiar with the variety of supported features to make use of DbC for web services. For example in existing DbC frameworks, assertions can be written as stylized comments, as in JML (Burdy et al 2005; Leavens et al 2006), or they can be embedded in the code as in Spec# (Barnett et al 2005). The various specification languages can be dependent on a programming language (ex JML depends on Java), or may independent of the source language (Spec#, CodeContracts). Learning and understanding different DbC technologies to apply them to modern, multi-language applications and software systems places a burden on programmers and to date has played a key role in the limited adoption of DbC.

To address this problem, we have developed *SuperContra*. A cross-language, cross-runtime DbC-as-a-service framework, that performs lightweight contract evaluations at runtime with minimal performance overhead. Lightweight contracts are predicates without method calls and object accesses. The effectiveness of using lightweight contracts to detect bugs has been previously recognized (Hatcliff et al 2012; Barnett and Schulte 2003; Briand et al 2003). Restricting the range of contract evaluations to lightweight checks, allows us to implement *SuperContra* as a service and to use a unified specification language independent of the underlying programming language. A service approach, not only enables the cross-language and runtime evaluation of contracts, but also comes with the additional benefits of increased re-usability of the contracts, ease of use, and loose-coupling between services and clients, service components, and components and contract enforcement. Developers can build clients for their systems in different programming languages without having to rewrite the contracts for each and every client. Such an approach reduces programming effort and the possibility for inconsistencies and errors in the contracts themselves. By providing decoupling clients from contract evaluation, each can evolve independently and contract evaluation can support new features (e.g. multi-client auditing) and contracts (e.g. for access control), and amortize the overhead of contract checking across large populations of clients.

The *SuperContra* design includes a dependency injection mechanism, a runtime interceptor, a reusable contract evaluator, and a cross-language communicator. The dependency injection mechanism, identifies the annotated contracts

[1] Trademark by Eiffel Software in the United States

and injects the interceptor code. The interceptor, forms the contracts from the annotations and delegates their validation to the evaluator. The contract evaluator, evaluates the contracts and returns the result to the interceptor. Finally, the communicator allows for the seamless communication between the interceptor and the evaluator across programming languages, by transforming, serializing and transferring the contracts and the corresponding outcomes.

To implement *SuperContra*, we leverage on existing open-source frameworks. The interceptor is based on the DBC Guice[2] DbC framework for Java that expresses pre-conditions, post-conditions and invariants as Java annotations and integrates Google Guice[3] to identify the contracts and inject evaluation code at runtime. The contract evaluator, is a modified version of the PyContract's[4] DbC framework for Python and evaluates the contracts that are expressed in a Python-like specification language. Finally the communicator parses the contracts, resolves incompatibilities between data types and uses Apache Thrift[5], as the RPC framework, to enable cross-language communication between the interceptor and the evaluator.

We evaluate the performance of *SuperContra* using three different experimental configurations using contracts that we apply widely used open-source systems (Synapse[6], HBase[7]). We compare the performance of *SuperContra* with that of existing DbC frameworks (Cofoja[8] and DBC Guice) and unmodified code (the same programs without contracts). We also employ community benchmarking and performance evaluation tools (Yahoo! Cloud Serving Benchmark[9], HBase Performance Evaluation Tool[10] and Apache JMeter[11]). Our results show that *SuperContra* performs similarly to non-service-based DbC approaches and in many cases similarly to code running without contracts.

In summary, the contributions of this paper are twofold:

- We present the design and implementation of a new approach to lightweight contract specification and enforcement within a framework called *SuperContra*. *SuperContra* decouples contract evaluation from client-side execution, and gives developers a single, familiar, yet universal language (a simple subset of Python) that they use to write (and reuse) contracts across components implemented in different programming languages.
- We empirically evaluate *SuperContra* using a wide range of configurations, software technologies, and precondition/postconditions. We compare *SuperContra* against traditional (client-integrated, single language/runtime) technologies and evaluate its overhead.

The remainder of this paper is organized as follows: Section 2 illustrates the design of *SuperContra* and Section 3 describes *SuperContra's* implementation. Section 4 discusses our experimental setup and presents the performance evaluation

[2] https://code.google.com/p/dbcguice/
[3] https://code.google.com/p/google-guice/
[4] http://andreacensi.github.io/contracts/
[5] http://thrift.apache.org/
[6] http://synapse.apache.org/
[7] http://hbase.apache.org/
[8] https://code.google.com/p/cofoja/
[9] https://github.com/brianfrankcooper/YCSB/
[10] http://wiki.apache.org/hadoop/Hbase/PerformanceEvaluation/
[11] http://jmeter.apache.org/

results. Section 5 discusses *SuperContra's* possible extensions and limitations of our approach. After reviewing related work in Section 6, Section 7 concludes.

2 SuperContra

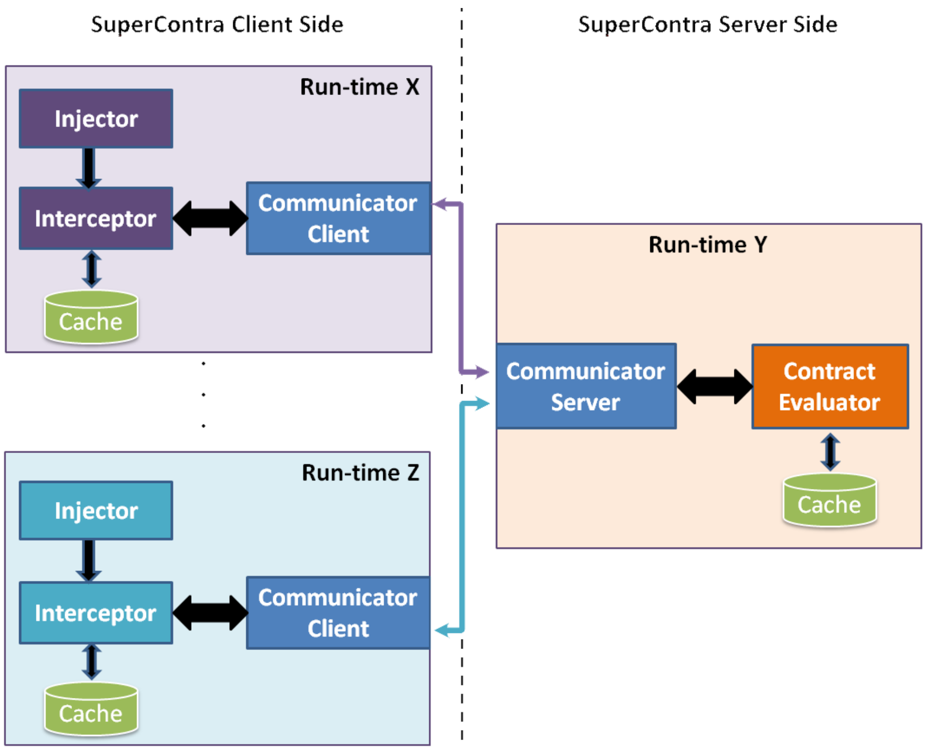


Fig. 1: The *SuperContra* system design

The *SuperContra* framework (Figure 1) is a client-server architecture and includes a dependency injector, a run-time interceptor, a contract evaluator and a cross-language communicator. In the heart of *SuperContra* is the contract evaluator, a centralized service that accepts a unified specification language and is responsible for evaluating the contracts and send the answers back to the clients. These clients can execute via different runtime environments and include a dependency injector and a runtime interceptor. The injector component identifies the contracts all the way up to the class hierarchy and injects the interceptor code that simply delegates the contract evaluation to the central evaluation engine. Finally, the communicator, is responsible for the communication between the interceptor and the evaluator across different run-time environments and consists of a client and a server component.

Contracts are expressed with a simple, common specification language independent of the programming language used. For each of the supported programming languages though, there is an injector and an interceptor written in this particular language that identify the contracts on the code, intercept the program execution at runtime and send the contracts for evaluation to the common contract evaluation service. The evaluator can be written in a different programming language or even running on a different runtime environment than other components. The system caches contract evaluations on the interceptor and on the contract evaluator to improve performance.

2.1 Specification Language

SuperContra specification language is a strict subset of the language used on the PyContracts framework. The types that SuperContra supports can be seen in listing 1. Listing 2 has two simple examples illustrating the usage of the wildcards * and #, used to match with any type or no type respectively.

Listing 1: Types supported by SuperContra

```
str, list, float, int, long, bool, bytearray, None
```

Listing 2: Wild-cards usage examples

```
@Precondition("{'discount': '*'}") //the discount can be of any type
@Precondition("{'discount': '#'}") //always returns error
```

PyContracts also provides built-in expressions specific to lists, tuples, sequences, dictionaries, arrays and maps. To examine the possibility of providing this useful feature also in SuperContra, we added support of the list type in the current implementation. Therefore, we can take advantage of the built-in expressions of PyContracts to specify constraints on list elements. Some examples of these list specific expressions are shown in listing 3.

Listing 3: List specific expressions

```
list[x] //Examines if a list has x elements length
list(int) //Argument is a list of integers
list[x](int) //Argument is a list of x integers
list[x](int, >y) //Argument is a list of x integers greater than y
list[>=x](int, >y) //Argument is a list of at least x integers great than y

//Example usage of list[x] in a contract
@Precondition("{'l': 'list[2](int, >0)'}")
      public boolean listExample(List<Integer> l)
```

SuperContra supports all the built-in functions of Python that accept as an argument one of the supported types mentioned above, or the object type. Some built-in functions very specific to the python environment are not displayed in the listing 4. For example, function *id(object)* returns an integer that is guaranteed to be unique for the object during its lifetime. Though, there is nothing prohibiting a developer to use this function inside a lambda expression of a contract in SuperContra, its usefulness in the context of contracts is questionable and thus we omit this and similar functions from our listings.

Listing 4 contains a list with the supported built-in functions while Listing 5 provides specific examples to illustrate their usage.

Listing 4: Built-in Python function supported by SuperContra

```
abs(x), bin(x), bool([x]), chr(i), cmp(x,y), complex([real[, imag]])
divmod(a, b), float(x), format(value[, format_spec]), hash(object), hex(x)
isinstance(object, classinfo), int(x), len(x), long(x), max(x,y), min(x,y)
oct(x), ord(c), pow(x,y), range(x), repr(object), slice(start, stop[, step])
str(object), type(object), unichr(i), unicode(object), xrange(start,stop[, step])
```

Listing 5: Built-in Python functions usage examples

```
@Precondition("{'name': 'lambda name: isinstance(name, str) and len(name)>4'}")
public boolean addPerson(String name, int age)

@Precondition("{'distance': 'lambda distance: abs(distance) < 5'}")
public boolean neighborhood(int distance)

@Precondition("{'arg': 'lambda arg: isinstance(float(arg), float) and
     bool(float(arg) > 0)'}")
public boolean example(String arg)
```

We next provide examples of how we use the language to express boundary conditions checks (Listing 6), non-nullness checks (Listing 7) as well as postconditions (Listing 8). In the last listing we see the use of lambda expressions to define type checks, a particularly useful feature for weakly typed languages, like Python.

Listing 6: Preconditions for numeric variables boundaries check

```
@Precondition("{'currentPrice': '>0', 'discount': '>=0', 'bonusCount':
     '>=0', 'bonusNo': '>=0'}")
public Double calculateDiscountPrice(double currentPrice, int discount,
     int bonusCount, int bonusNo)
```

Listing 7: Using lambda expressions for non-nullness checks

```
@Precondition("{'row': 'lambda row: row is not None', 'family': 'lambda
     family: family is not None'}")
public long incrementColumnValue(final byte [] row, final byte []
     family,final byte [] qualifier, final long amount, final boolean
     writeToWAL)
```

Listing 8: Post-condition check

```
@Postcondition("{'returns': 'lambda price: price>0'}")
public Double calculateDiscountPrice(double currentPrice, int discount,
     int bonusCount, int bonusNo)
```

Listing 9: Type-checking for weakly typed languages

```
@Precondition("{'name': 'lambda name: isinstance(name, str) and
     len(name)>4', 'age': 'int,>10'}")
@Postcondition("{'returns': 'bool'}")
```

2.2 Lightweight Contracts

SuperContra supports the evaluation of lightweight contracts. Light-weight contracts are any expression that does not contain method calls or object references. By focusing on lightweight contracts, we preclude the need for the server side to implement the object model of the client. Moreover, types are converted to the closer type supported by the server's contract evaluation framework (Currently PyContracts), similarly to other cross-language frameworks like Apache Thrift. Nevertheless, light-weight contracts as type, boundary and nullness checks can be extremely effective in detecting the plethora of system bugs encountered in practice (Hatcliff et al 2012; Barnett and Schulte 2003; Briand et al 2003).

3 Implementation

The current implementation of the *SuperContra* framework (Figure 1) exemplifies our cross-language and cross-runtime approach using Java and Python. The client side of SuperContra is implemented in Java, consisting of an injector, an interceptor and, the client piece of the communicator and the server side is implemented in Python consisting of a contract evaluator and the server piece of the communicator.

3.1 SuperContra Injector

The *SuperContra* injector, in the current implementation of the SuperContra client side, is the Google Guice injection mechanism, assigned with the task to identify the classes with contract annotations and inject the code of the *SuperContra* interceptor.

3.2 SuperContra Interceptor

The *SuperContra* interceptor is a modified version of the DBC Guice framework for Java. DBC Guice, makes use of the Java annotations to define preconditions, postconditions and invariants. It integrates with the Google Guice injection mechanism in order to intercept the function calls, containing such annotations, and to inject a contract evaluator implementation at runtime. The default module provided for contract evaluation is based on the BeanShell[12] scripting language. We have used the default evaluation module of DBC Guice as a reference for comparison reasons (see section 4) and built our own evaluation module for *SuperContra.* We leveraged on a modified version of the DBC Guice interceptor, added caching and developed the necessary functionality that parses the contracts and the argument values of the method invocation and sends them for validation through the communicator client (see section 3.4).

[12] http://www.beanshell.org/

3.3 SuperContra Evaluator

The *SuperContra* evaluator is responsible for evaluating the contracts and returning a positive result or throwing an exception if there is a contract violation. To accomplish this, it checks the cache for available evaluation results of the same contract and argument values and directly returns the answer if such an entry exists. Otherwise, it parses the request, verifies that it does not contain any invalid type and creates a new contract. This contract should be in a format accepted by the contract evaluation engine, which in our current implementation, is a slightly modified version of the PyContracts DbC framework.

3.4 SuperContra communicator

The *SuperContra* communicator in the current implementation is Apache Thrift. Consequently, the communicator client and server correspond to a Thrift client and server respectively. Apache Thrift is a lightweight, language-independent software stack with an associated code generation mechanism that allows developers to build RPC clients and servers by just defining the data types and service interfaces in a simple definition file. Given this file as an input, code is generated to build clients and servers that communicate seamlessly across programming languages. Thrift provides clean abstractions for data transport, data serialization, and application level processing and it supports a variety of languages including C++, Java, Python, PHP and Ruby.

Thrift comes with a plethora of protocol and transport layer choices as well as with different types of supported servers. Nevertheless, not all the choices are available for all the supported languages. For our current setup that uses Java for the Thrift client and Python for the Thrift server, and after experimenting with the available options, we end up using the TBinaryProtocol for the protocol layer with the TBinaryProtocolAcceleratedFactory on the server side and the TSocket option for the transport layer with the TBufferedTransportFactory on the server side. Finally, we used the TThreadPoolServer and the TForkingServer for our threaded and forking *SuperContra* versions respectively. The Thrift version we used is 0.9.0.

As we can see in Figure 1, the communicator is responsible for transferring the contract evaluation requests and responses between the *SuperContra* interceptor and the contract evaluator. It converts the arguments of the method to be evaluated from strings to the corresponding python values and creates a contract in an appropriate format, accepted by the contract evaluation engine.

4 System Evaluation

In this section we describe our experimental setup and present the results of our performance evaluation. We first explain the methodology we followed to evaluate SuperContra, describe our testbed, the assertions added on the code and the DbC frameworks we used for comparison reasons and then present and explain the extended results we retrieved.

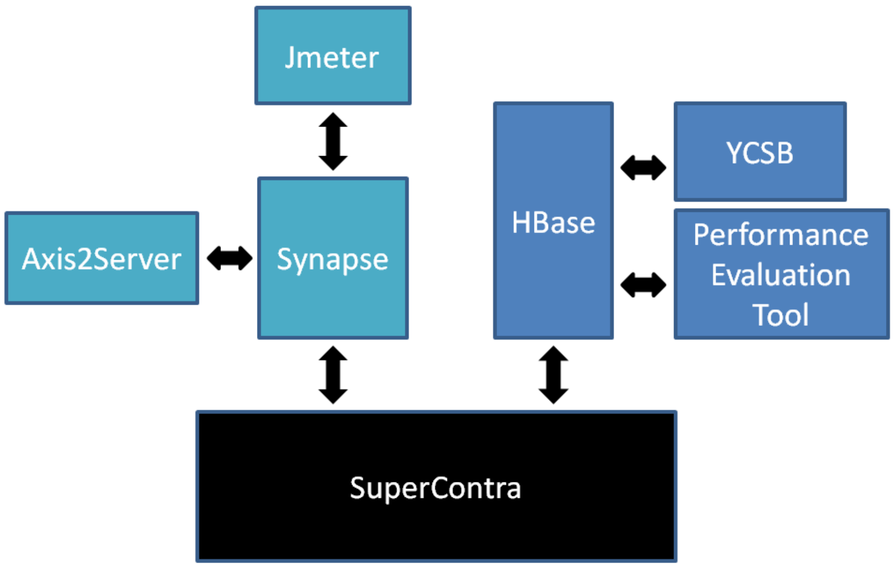


Fig. 2: Experimental setups for the *SuperContra* evaluation. The setup on the left consists of a contracted version of Apache Synapse and uses JMeter as the traffic generator (Section 4.1.1). The setups on the right both use contracted versions of HBase and generate transactions with the Performance Evaluation Tool of HBase (Section 4.1.2) and YCSB (Section 4.1.3)

4.1 Experimental Setup

We evaluate the performance and scalability of our approach by using the 3 experimental configurations depicted in Figure 2. With the first setup, we evaluate the performance of our system by adding contract checks to a mediation sample of the Apache Synapse Enterprise Service Bus and using Apache JMeter to generate traffic. In the next two configurations, we run performance evaluations on Apache HBase, which we instrument with contract checks on the part of the API that is called by our evaluation procedures. The instrumentation of the methods and classes with contracts is implemented through suitable wrappers, leaving the original API unaffected and ensuring that any other operation of HBase that use parts of the code we modified does not influence the performance results. Using the second configuration, we run the performance tests using a modified version of the HBase Performance Evaluation Tool and on the third we use benchmarks provided by YCSB (Cooper et al 2010).

For each configuration, we compare the performance of *SuperContra* against the unmodified version of the code, ie without contract assertions and also against two open source contract frameworks for Java, namely DBC Guice and Cofoja. We have evaluated two different implementations of SuperContra, one that is using Thrift's thread pool server and one using Thrift's forking server. Though, since the difference in their performance is insignificant, we present only the performance results obtained with the forking server that performed overall equal or slightly better. Overall, the performance evaluation tests employ four different contract configurations.

- *Without Contracts*, the original code running without contract assertions
- *DBC Guice*, an open source project that implements DbC for Java using Google Guice
- *Cofoja*, Contracts for Java or Cofoja is a popular DbC framework made by a team of Google engineers
- *SuperContra*, the *SuperContra* framework

The following three subsections provide additional information regarding the architecture and the code instrumentation for each experimental setting.

4.1.1 Synapse/ JMeter

Apache Synapse is a free, open source, lightweight Enterprise Service Bus (ESB) and mediation engine that comes with a plethora of working samples, that are general comprised of a sample configuration, an Axis2-based Web Service to which Synapse sends messages and an Axis2-based service client which sends requests to Synapse. Apache JMeter is a load testing tool that we use to generate HTTP requests (replacing the role of the Axis2 client) and to gather the results.

With this experimental setup we want to investigate the performance and scalability of our system on a networked environment with the different testing participants isolated. Thus, the client (JMeter), the Synapse mediation engine running the instrumented code and the Axis2 Server are located on different machines. We use Synapse with one of the provided configurations[13] and we modify the DiscountCodeMediator class by adding contracts on the method that calculates the discounts on the input price. To evaluate the performance under different contract load we create 3 different versions of this method by adding contract preconditions to one argument, to every argument of the call (4 arguments in total), and by combining preconditions on every argument and a postcondition to check the validity of the returned result. According to this scenario, JMeter generates the HTTP requests and sends them to the Axis2 server. The server then, generates the results but before sending them back to the client, passes them to the mediator who is responsible to perform the contract checks and calculate the discounted price that is eventually returned to the client. We used Synapse version 2.1.0 and JMeter version 2.9 to perform the experiments.

4.1.2 HBase/ Performance Evaluation Tool

The HBase PerformanceEvaluation class is part of the HBase distribution code and provides a number of different tests for the HBase API, like sequential and random reads and writes, scans etc.

For this set of experiments we wanted to add contracts on one of the basic HBase API calls and compare against the original code of HBase to demonstrate the effectiveness of our approach under a simple scenario that does not require the addition of contracts in every function call of the complicated HBase flow. A suitable HBase API operation for this reason, was the incrementColumnValue

[13] http://synapse.apache.org/userguide/samples/sample380.html

operation, which checks whether the row and the family provided by the caller is not null and throws an exception otherwise. We created three modified versions of this method by replacing the conditional checks with contracts written with DBC Guice, Cofoja and *SuperContra.* Since, the PerformanceEvaluation class of HBase does not provide tests for increments, we extended the tests provided by the tool to also support sequential increment tests and added four different commands to the tool in order to call the original version and also the 3 modified contracted versions of incrementColumnValue. For our experiments we used HBase 0.94.6.1 in standalone mode.

4.1.3 HBase/ YCSB

YCSB, Yahoo! Cloud Serving Benchmark, is a framework and common set of workloads that a developer can use to evaluate and compare the performance of different key-value store systems, like BigTable, Azure, DynamoDB, HBase etc.

With this set of experiments we wanted to facilitate evaluation using YCSB workloads. We used 3 of the core YCSB workloads, the update-heavy (50/50 Read/Update), read-heavy (95/5 Read/Update) and read-only workload. For each workload, we modified all the HBase methods affected by the call to the HBase read or write, by adding contract checks to their arguments. After the modifications, every read API call invokes 5 different methods with contract checks on 5 arguments in total and every write API call invokes 3 different methods with contract checks on 7 arguments in total. We also modified the hbase-binding of YCSB to redirect the calls to the instrumented versions of the methods. We used YCSB version 0.1.4.

4.2 Results

The results gathered from all our experimental setups show that *SuperContra* outperforms the performance of DBC Guice and achieves similar performance with the unmodified HBase code (that runs without contracts checks) and the Cofoja framework.

With Cofoja, contracts are compiled into separate contract files and a Java agent re-writes the original code during the class loading time achieving significantly better performance compared to DBC Guice and slightly better performance than *SuperContra.* DBC Guice, uses BeanShell as the scripting language to define the contracts. BeanShell, is dynamically interpreted Java, plus a scripting language and a flexible environment in a single package. Since it is an interpreter, execution is significantly slower compared to compiled Java code.

On the other hand, *SuperContra* performs slightly worse than Cofoja when we evaluate multiple preconditions and postconditions because the contracts are not only checked but also created at runtime. Each time a method with a contract annotation is encountered at runtime by Google-Guice, the necessary code is injected in order to create a contract according to the specified precondition, then send to the *SuperContra* evaluation engine through Thrift, evaluated by the PyContracts engine and finally an answer is send back to the method that triggered the contract.

4.2.1 Synapse/ JMeter

We first present the results for the Synapse/JMeter configuration in Figure 3. In these experiments, we create three versions of the Synapse discount service: One that performs a precondition check on one of the discount function arguments (Fig. 3a), one with precondition checks in every argument (Fig. 3b) and one that has preconditions in every argument and a postcondition assertion on the returned result (Fig. 3c). JMeter sends 10000 requests to Synapse, which in turn forwards the requests to the Axis2 server. The response message created by the Axis2 server is passed through the class mediator, to the instrumented with contracts Java class that calculates the discount.

In the case of only one precondition check (Figure 3a), the time needed to evaluate the contracts is so small that there is almost no performance difference between the different frameworks tested. When the contract checks are heavier though (Figures 3b and 3c), *SuperContra* performs better than DBC Guice and very close to the optimal performance of the code running without contracts. When using 10 or more threads on the client the performance of *SuperContra* has almost no difference with the code running with Cofoja and the unmodified code.

4.2.2 HBase Performance Evaluation Tool

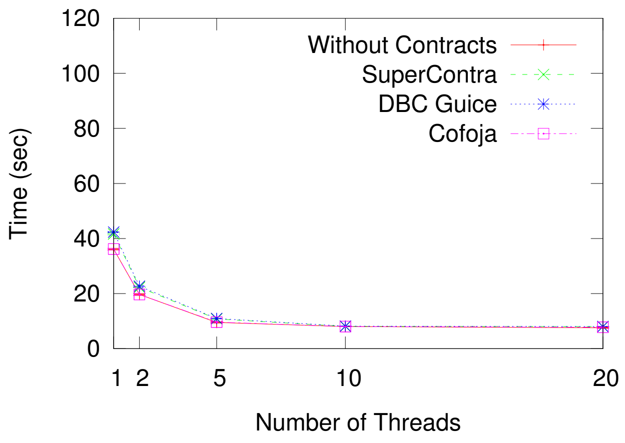
Figure 4a shows the total time required for an HBase Increment to increase the values of one column in 10.000 different rows. *SuperContra* outperforms the performance of DBC Guice and achieves similar performance with the unmodified HBase code and the Cofoja framework.

Caching Results. In many cases (e.g HBase batch calls), the same operation with the same contract and values is repeated. To reduce the overhead when this occurs, we have implemented a cache mechanism on both sides of our architecture, i.e. on the contract evaluator side and on the client side, that is accessed before the contract is sent to the server for evaluation and before the evaluation is performed. When there is a single client with multiple threads that access the same evaluation server the caching mechanism on both sides is redundant. However, for multiple clients, server side caching can be effective. Figure 4b shows that caching improves performance by up to 22% for HBase row increments.

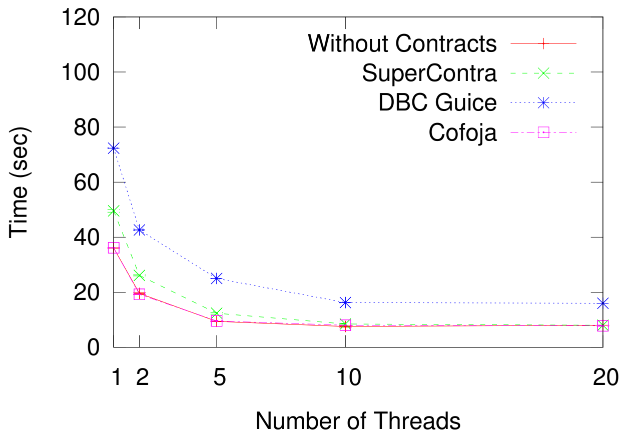
4.2.3 HBase/ YCSB

Figure 5 presents the results for the experimental setting described on section 4.1.3 that uses three YCSB workloads. In all cases *SuperContra* outperforms DBC Guice and scales well, adding a small overhead compared to the results of Cofoja and the code without contracts. The graph in Figure 5a is for the read-only workload, the graph in 5b is for the read-heavy workload, and the graph in 5c is for the update-heavy workload.

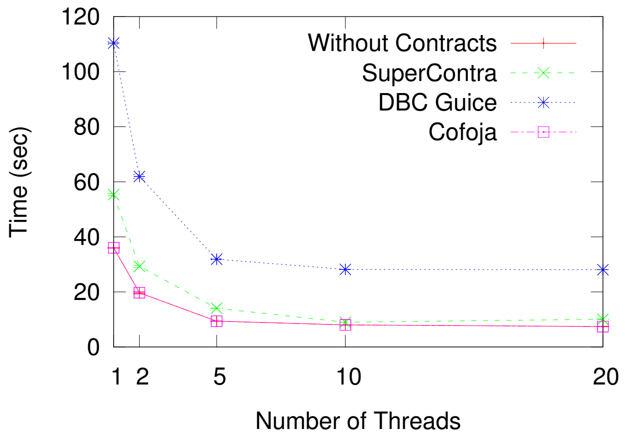
The update-heavy results show DBC Guice time reducing significantly without the same thing happening for *SuperContra.* One reason for this reduction is that the update latency of HBase is smaller compared to the read latency, because updates are buffered into memory (Cooper et al 2010). For this reason the



(a) Precondition Check

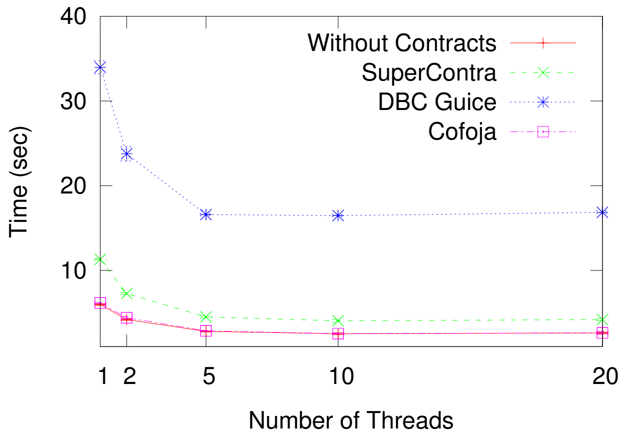


(b) Heavy Preconditions

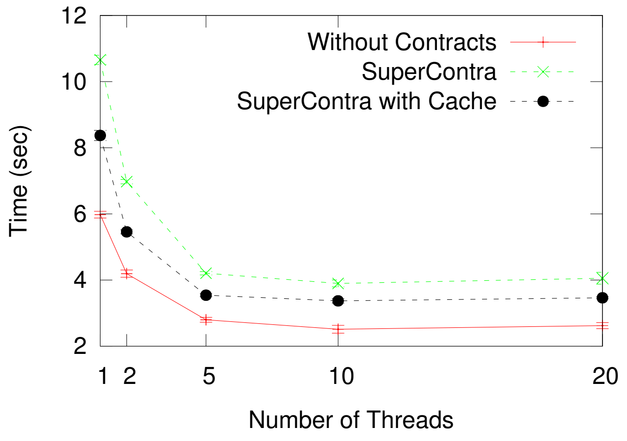


(c) Heavy Preconditions+Postcondition

Fig. 3: Synapse/ JMeter: Total execution time for 10k operations for one precondition check (Fig. 3a), multiple precondition checks ((Fig. 3b) and the combination of multiple precondition checks plus a postcondition (Fig. 3c). The graphs contain 4 different curves representing the results from the original code running without contracts, the code using *SuperContra*, the code using the DBC Guice framework and finally the code using Cofoja as the DbC framework. In all cases *SuperContra* performs better than DBC Guice and similarly to the code running with Cofoja and the unmodified code without contract checks



(a) 10k Increments on Different Rows

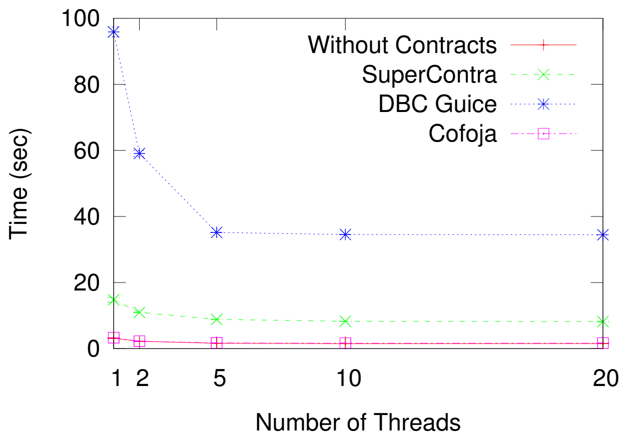


(b) 10k Increments on the Same Row

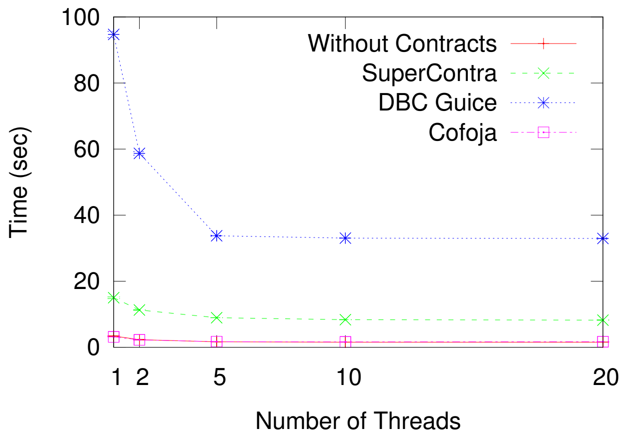
Fig. 4: Total execution time of 10k HBase Increments on different rows (Figure 4a) and on the same row (Figure 4b). *SuperContra* performance is better than DBC Guice and similar to the code running with Cofoja contracts and the unmodified code without contracts. With cache enabled on contract evaluator we can have up to 22% performance gain in the best case (all increments on the same row)

performance of the update-heavy workload (Figure 5c) is expected to be better compared to the read-heavy workload (Figure 5b). This holds for the DBC Guice framework but not for *SuperContra*.

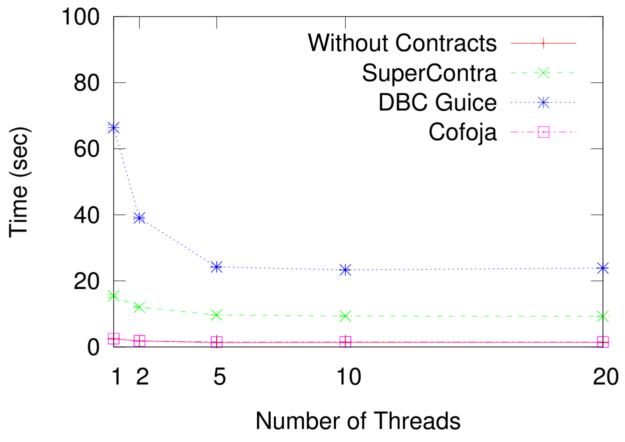
To investigate this anomaly, we ran additional experiments with the same number of contract checks both for reads and updates (only one contract check on each case) and we plot the system's latency as the number of threads increases, in Figure 6. With only one contract check, the read latency is, as expected, higher than the update latency for both DBC Guice and *SuperContra*. When evaluating all the contracts (Figure 7), with five or more threads, the update latency of



(a) Read-Only Workload

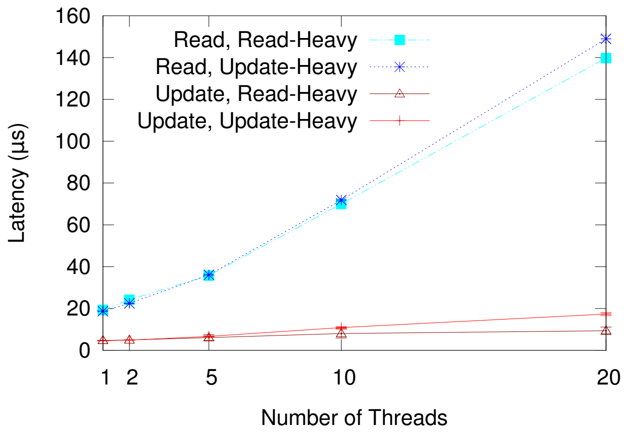


(b) Read-Heavy Workload

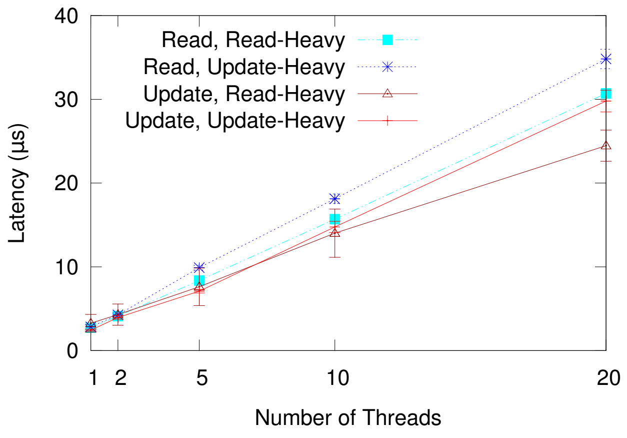


(c) Update-Heavy Workload

Fig. 5: HBase/ YCSB: Total execution time for 5k operations for the read-only (Fig. 5a), read-heavy (Fig. 5b) and update-heavy (Fig. 5c) workloads of YCSB run on HBase



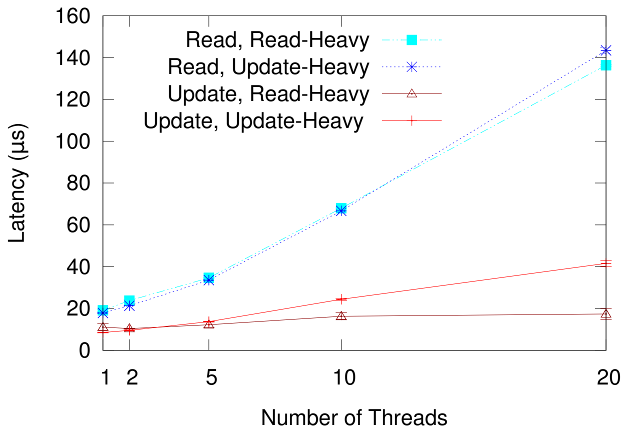
(a) DBC Guice



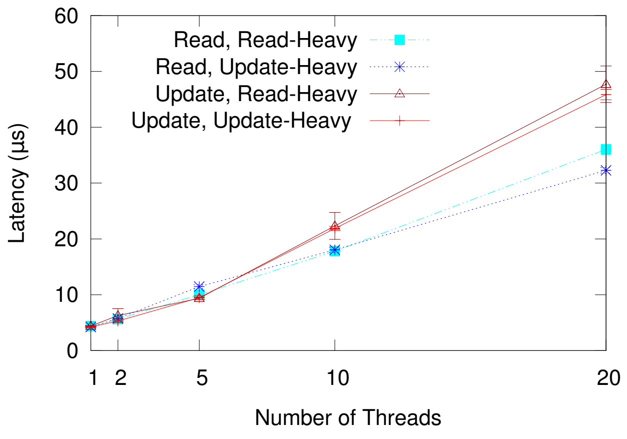
(b) SuperContra

Fig. 6: One contract evaluation: Update latency (solid lines) versus Read latency (dashed lines) as the number of threads increases. Only one contract per read and one per update is evaluated. The number of contracts evaluated for reads and updates is the same. Read latency is higher than the update latency for both DBC Guice and SuperContra

SuperContra becomes higher than the read latency. This means that in this case the *SuperContra* evaluator capacity becomes a bottleneck and thus it cannot obtain further performance gain on the update-heavy scenario. For the experiments running without contracts and for Cofoja their is also around 10% performance improvement.



(a) DBC Guice



(b) SuperContra

Fig. 7: All contracts evaluation: Update latency (solid lines) versus Read latency (dashed lines) as the number of threads increases. The number of contract evaluations for reads and updates is different (see section 4.1.3). DBC Guice has greater read latencies than update latencies as in the case of one contract check (Figure 6a). Thus, it can obtain significant performance gains when running the update-heavy workload. One the other hand, SuperContra's update latencies are greater than the read latencies both for the read-heavy and the update-heavy scenarios. So, there is no performance gain observed on Figure 5 between the read-heavy and update-heavy workloads but a slight decrease in performance

5 Discussion

SuperContra is a service that supports lightweight contracts. This design decision restricts the range of contract validations that can be performed with *SuperContra* versus JML and Spec#. However, such language-dependent systems often fail to support all the language features (Hatcliff et al 2012) which limit their effectiveness. In addition, a programmer that wants to develop multiple clients using different programming languages that consume the same service, is forced to learn different DbC frameworks to do so with these language-dependent approaches. *SuperContra* is not precludes these issues.

Lightweight contracts have been previously recognized to be an effective mechanism for detecting programming errors (Hatcliff et al 2012; Barnett and Schulte 2003; Briand et al 2003). Hatcliff and Leavens (Hatcliff et al 2012) argue that lightweight annotations like those employed by *SuperContra* can effectively be used to identify and isolate bugs. Moreover, Barnett states that "the bulk of specifications are expected to be simple" (Barnett and Schulte 2003) which means that *SuperContra* can be useful in most practical scenarios. The simplicity of writing contracts with *SuperContra* reduces the barrier to entry of using DbC.

Apart from improving code robustness, offering the evaluation of contracts as a service has other benefits:

Ease of use. The contracts can be written in one common specification language independent of the programming language used and can be evaluated across languages and runtime environments. Developers do not need to be familiar with the different features, syntax and semantics supported on the variety of DbC frameworks that exist for each programming language;

Contract re-usability. The exposed interfaces of a service that support clients in different languages do not differ significantly. Thus, since the specification language is independent of the underlying programming language, the same contract assertions can be used for all the clients regardless of the programming language they are developed in. Moreover, the service provider can now include contracts as part of his interface documentation in order to reduce the programming effort needed to develop the clients for his service;

Loose coupling. The contract evaluator engine can evolve in order to support more features without the need to recompile any of the clients or change the previous contracts, as long as the old features are still supported; and

Amenability to use on resource-constrained devices. The evaluation of contracts in resource-constraint devices such as mobile phones would require more time, CPU and battery consumption. Off-loading the evaluation process to an external service that could potentially run in the cloud, makes feasible the addition of contracts on code running on mobile devices, both for profiling and debugging purposes.

Finally, a centralized service like *SuperContra* could have some alternative applications, including:

Data aggregation. Having a central contract evaluator for all the clients that use a service, allows to piggyback on the contract evaluation, the aggregation and processing of the evaluated values. Thus, the gathered data can be used to enhance profiling, analytics and service management. For example, increased contract violations for a particular method could indicate an error or unclarity on the published documentation provided for this method; and

Dynamic contract variables. *SuperContra* can be used for the evaluation of contracts, against values that are controlled and dynamically updated by a central authority. To illustrate this, lets think of the scenario that a service provider offers SuperContra as a DbC framework to his clients to help them evaluate the conformance of their code with the service. Since, SuperContra runs on his own premises, he could register specific variable names and be able to dynamically modify them at will, or automatically, according to some global system values that he monitors. In this case, sufficient documentation should be provided to the developer to let them know about the dynamic variables they can use.

6 Related Work

The idea of DbC has been extensively applied on the implementation level with numerous behavioral interface specification languages as an outcome (Hatcliff et al 2012) for sequential (Meyer 1992; Burdy et al 2005; Barnett et al 2005; Fähndrich et al 2010; Barnett and Schulte 2001) and concurrent (Dahlweid et al 2009; Araujo et al 2008; Nienaltowski et al 2006) programs. There are programming languages like Eiffel (Meyer 1988) or Spec# (Barnett et al 2005) that are natively equipped with contracts and DbC frameworks that are an extension to an existing programming language such as JML (Burdy et al 2005) and Cofoja (Minh 2010), while other frameworks are dependent on the programming language they support. Contracts can be checked at compilation time (Flanagan et al 2002; Barnett et al 2005; Xu et al 2009), at runtime (Minh 2010; Cheon and Leavens 2002) or both (Burdy et al 2005). *SuperContra*, in its current implementation supports a basic python-like type system, and evaluates contracts at runtime.

To avoid the hassle of learning a new specification language for each different DbC tool, Fahndrich et al (Fähndrich et al 2010) suggest the embedding of contracts into the programming language, and prove their approach by providing .Net libraries for C#, Visual Basic, F# and C++ (Fahndrich et al 2012). Despite having a unified approach for .Net supported languages, the specification syntax for the different programming languages still differs and cannot be written just once and used multiple times for clients implemented in multiple programming languages. In another work, Barnett et al present ASML as the language to write specifications for all the .Net supported languages. Having the same starting point, reducing the hassle of writing contracts with many different specification languages and tools, *SuperContra* not only is language agnostic, but can also support the evaluation of contracts for programming languages with different runtime environments. We illustrate our approach by evaluating contracts written for Java code using a contract evaluator written in Python.

Recognizing the complexity of current DbC frameworks and the runtime overhead they introduce, Dimoulas et al propose option contracts, allowing service producers to tag contracts as an option and giving service consumers the ability to chose whether to exercise the option or accept it, taking the responsibility if things go wrong (Dimoulas et al 2013). Referring to the complexity of manually writing contracts as the main reason for the lack of adoption of DbC, Qi and Yi suggest change contracts, where they express as a contract the intended behavior of software changes (Qi et al 2012; Yi et al 2013). SuperContra shares the same motivation with the aforementioned research, but addresses the problem in a different way. To decrease the runtime overhead and to minimize the programming effort needed to write the contracts, *SuperContra* restricts its evaluation space on light weight contracts and uses a common specification language across run-times to enable easier writing of contracts and increase their reusability.

7 Conclusions

This paper presents the design, implementation and evaluation of *SuperContra*. *SuperContra* is a DbC as-a-service framework that evaluates lightweight contracts across programming languages and runtime environments. We evaluate *SuperContra* using popular, open-source, production software and compare its performance against existing DbC frameworks with the use of well-known benchmarks and tools. Our results show that *SuperContra* performs similarly to or outperforms extant approaches (single language/runtime) to DbC and in some cases performs similarly to native code without contracts. We also observe a significant reduction in programming effort for the specification of contracts across components written in different programming languages making *SuperContra* lightweight, effective, and easy-to-uses solution for improving software reliability.

8 Acknowledgements

This work was funded in part by NSF (CNS-0905237 and CNS-1218808) and NIH (1R01EB014877-01).

References

Araujo W, Briand L, Labiche Y (2008) Concurrent contracts for java in jml. In: Software Reliability Engineering, 2008. ISSRE 2008. 19th International Symposium on, IEEE, pp 37–46

Barnett M, Schulte W (2001) The abcs of specification: asml, behavior, and components. Informatica (Slovenia) 25(4)

Barnett M, Schulte W (2003) Runtime verification of. net contracts. Journal of Systems and Software 65(3):199–208

Barnett M, Leino KRM, Schulte W (2005) The spec# programming system: An overview. In: Construction and analysis of safe, secure, and interoperable smart devices, Springer, pp 49–69

Briand LC, Labiche Y, Sun H (2003) Investigating the use of analysis contracts to improve the testability of object-oriented code. Software: Practice and Experience 33(7):637–672

Burdy L, Cheon Y, Cok DR, Ernst MD, Kiniry JR, Leavens GT, Leino KRM, Poll E (2005) An overview of jml tools and applications. International journal on software tools for technology transfer 7(3):212–232

Cheon Y, Leavens GT (2002) A runtime assertion checker for the java modeling language (jml). In: Proceedings of the International Conference on Software Engineering Research and Practice (SERP02), Las Vegas, Nevada, USA, pp 322–328

Cooper BF, Silberstein A, Tam E, Ramakrishnan R, Sears R (2010) Benchmarking cloud serving systems with ycsb. In: Proceedings of the 1st ACM symposium on Cloud computing, ACM, pp 143–154

Dahlweid M, Moskal M, Santen T, Tobies S, Schulte W (2009) Vcc: Contract-based modular verification of concurrent c. In: Software Engineering-Companion

Volume, 2009. ICSE-Companion 2009. 31st International Conference on, IEEE, pp 429–430

Dimoulas C, Findler RB, Felleisen M (2013) Option contracts. In: Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications, ACM, pp 475–494

Fähndrich M, Barnett M, Logozzo F (2010) Embedded contract languages. In: Proceedings of the 2010 ACM Symposium on Applied Computing, ACM, pp 2103–2110

Fahndrich M, Barnett M, Leijen D, Logozzo F (2012) Integrating a set of contract checking tools into visual studio. In: Developing Tools as Plug-ins (TOPI), 2012 2nd Workshop on, IEEE, pp 43–48

Flanagan C, Leino KRM, Lillibridge M, Nelson G, Saxe JB, Stata R (2002) Extended static checking for java. In: ACM Sigplan Notices, ACM, vol 37, pp 234–245

Hatcliff J, Leavens GT, Leino KRM, Müller P, Parkinson M (2012) Behavioral interface specification languages. ACM Computing Surveys (CSUR) 44(3):16

Leavens GT, Baker AL, Ruby C (2006) Preliminary design of jml: A behavioral interface specification language for java. ACM SIGSOFT Software Engineering Notes 31(3):1–38

Meyer B (1988) Eiffel: A language and environment for software engineering. Journal of Systems and Software 8(3):199–246

Meyer B (1992) Applying'design by contract'. Computer 25(10):40–51

Minh N (2010) Contracts for Java: A Practical Framework for Contract Programming. Tech. rep., Google Switzerland GmbH, `http://cofoja.googlecode.com/files/cofoja-20110112.pdf`

Nienaltowski P, Meyer B, Ostroff JS (2006) Contracts for concurrency. Report-University of York Department of Computer Science YCS 405:27

Qi D, Yi J, Roychoudhury A (2012) Software change contracts. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, ACM, p 22

Xu DN, Peyton Jones S, Claessen K (2009) Static contract checking for haskell. In: Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM, New York, NY, USA, POPL '09, pp 41–52, DOI 10.1145/1480881.1480889

Yi J, Qi D, Tan SH, Roychoudhury A (2013) Expressing and checking intended changes via software change contracts. In: Proceedings of the 2013 International Symposium on Software Testing and Analysis, ACM, pp 1–11