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

Stratos Dimopoulos, Chandra Krintz, Rich Wolski, Anand Gupta
Department of Computer Science
University of California, Santa Barbara
stratos@cs.ucsb.edu

Abstract—This paper presents SuperContra - a Design-by-Contract (DbC) framework that can ship with future PaaS offerings to enforce 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. 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.

I. INTRODUCTION

The wide number of libraries and application programming interfaces (APIs) offered by current Platform-as-a-Service providers, has enabled the rapid deployment of software developed with multiple languages and runtimes. In order to ensure the reliability and robustness of these complex, multi-language components, in large-scale PaaS settings, thorough testing is necessary. A mature testing methodology that developers use to eliminate bugs and improve exception handling is Design-by-Contract (DbC) [1]. Nevertheless, current PaaS offerings do not provide a built-in DbC service to allow a unified evaluation of software contracts across languages and runtimes.

To address this need, we have developed SuperContra, a cross-language, cross-runtime DbC framework for PaaS that offers contracts as-a-service and enforces them efficiently at runtime. With SuperContra, developers specify contracts for each application component using a single, familiar specification language, regardless of the programming language they use for component implementation. Such unification of contract specification across languages saves developers the time and effort required to learn and make use of potentially multiple, per-language DbC frameworks. Moreover, the same contracts can be re-used when code is ported to another language and can constitute an up-to-date documentation that drives new code development. Finally, offering contracts as-a-service promotes a loose-coupling between the different contract service components, giving PaaS providers the ability to incrementally add new capabilities to the contract evaluation engine.

SuperContra includes a dependency injection mechanism, a run-time interceptor, a reusable contract evaluator, and a cross-language communicator. The dependency injection mechanism, identifies the annotated contracts 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. The current implementation of SuperContra exemplifies its cross-language and cross-runtime capabilities by evaluating contracts between Java clients and a Python contract evaluator.

We evaluate the performance of SuperContra using Apache JMeter [2], a popular open-source load testing application, with which we generate traffic on an instrumented with contracts version of Synapse [3], a widely-used open-source Enterprise Service Bus (ESB). We compare it’s performance to existing DbC frameworks (Cofoja [4] and DBC Guice [5]) and unmodified code (the same programs without contracts). 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 DbC framework that enables uniform contract evaluation and contracts as-a-service. 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 evaluate SuperContra with widely used software technologies and a wide range of preconditions/postconditions. We compare SuperContra against traditional (client-integrated, single language/runtime) technologies and evaluate its overhead.

II. SUPERCONTRA

We overview the SuperContra design in Figure 1. The framework 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 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
Fig. 1: The SuperContra system design

the way up to the class hierarchy and injects the interceptor code that simply delegates the contract evaluation to the 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 current implementation of the SuperContra framework exemplifies our cross-language and cross-runtime approach using Java and Python. To implement SuperContra, we leverage on existing open-source frameworks. The interceptor is based on the DBC Guice [8] DbC framework for Java that expresses preconditions, post-conditions and invariants as Java annotations and integrates Google Guice to identify the contracts and inject evaluation code at runtime. The contract evaluator, is a modified version of the PyContracts [9] 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 [10], as the RPC framework, to enable cross-language communication between the interceptor and the evaluator. We describe our implementation choices in detail in an extended tech report [6].

A. 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.

We can also specify constraints on lists, tuples, sequences, dictionaries, arrays and maps. Some examples for list specific expressions are shown in listing 2.

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.

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.

We next provide examples of how we use the language to express boundary conditions checks (Listing 4), non-nullness checks (Listing 5) as well as postconditions (Listing 6). 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 1: Types supported by SuperContra

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

### Listing 2: List specific expressions

| list[x]  //Examines if a list has x elements |
| 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)

### Listing 3: 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)

We next provide examples of how we use the language to express boundary conditions checks (Listing 4), non-nullness checks (Listing 5) as well as postconditions (Listing 6). 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 4: Preconditions for numeric variables boundaries check

@Precondition("{'currentPrice': '>0', 'discount': '>=0', 'bonusCount': '>=0', 'bonusNo': '>=0'}")

### Listing 5: Using lambda expressions for non-nullness checks

@Precondition("{'row': 'lambda row: row is not None', 'family': 'lambda family: family is not None'}")

### Listing 6: Using lambda expressions for postconditions

@Precondition("{'row': 'lambda row: row is not None', 'family': 'lambda family: family is not None'}")
B. 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 ([7], [8], [9]).

III. System Evaluation

We evaluate the performance and scalability of our approach using the Apache Synapse Enterprise Service Bus (ESB) v2.1.0. We drive this PaaS service using Apache JMeter v2.9 and employ a configuration in which Synapse interoperates with an Axis2 server [10]. We also evaluate SuperContra for a NoSQL service implemented via Apache HBase. Due to space constraints however, we omit these results herein. A complete description of our evaluation setup and these and other results can be found in an extended technical report version of this paper [6]. We compare the performance of SuperContra against unmodified implementations of the services. We employ DBC Guice and Cofoja, two popular DbC frameworks for the Java language, to evaluate and compare different contract implementations.

For contract evaluation in Synapse, we modify the DiscountCodeMediator within this sample configuration by adding contracts to the method that calculates the discounts on the input price. The contracts we consider are (i) a precondition on one argument, (ii) a precondition on each of four arguments, and (iii) configuration ii with the addition of a postcondition on the return value. A JMeter client sends HTTP requests to the Axis2 server which then communicates with Synapse, at which point the contracts are evaluated prior to returning the result to the client.

We present performance results (in seconds) for each of our three configurations in Figure 2 for 10000 JMeter requests. With only one precondition check (Figure 2a), the time needed to evaluate the contracts is so small that there is almost no performance difference between the different frameworks tested. When there are more contract checks (Figures 2b

Fig. 2: Synapse/ JMeter: Total execution time for 10k operations for one precondition check (Fig. 2a), multiple precondition checks ((Fig. 2b) and the combination of multiple precondition checks plus a postcondition (Fig. 2c).
and 2c), the services experience some overhead. In all settings, SuperContra outperforms DBC Guice and imposes negligible overhead. With 10 or more threads, the performance of SuperContra is similar to the code running with Cofoja and the unmodified code.

IV. Related Work

DbC has been extensively studied, e.g. [7], for sequential [1], [11], [12], [13], [14] and concurrent [15], [16], [17] programs. There are programming languages [18], [12] that consider contracts as first-class structures, and DbC frameworks that extend existing programming languages [11], [19]. Contracts can be checked at compilation time [20], [12], [21], at runtime [19], [22], or both [11]. SuperContra supports a python-like type system and specification language regardless of the language used to implement the program or component, and evaluates contracts at runtime.

Previous research tried to ease the burden of learning a new specification language for each different DbC tool [13], [23] by embedding contracts into the programming language. SuperContra goes beyond such approaches to provide DbC that is both language and run-time agnostic. Other DbC advances target runtime overhead [24] and the complexity of writing contracts [25], [26]. SuperContra shares this motivation, 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 targets lightweight contracts and uses a common specification language across run-times to simplify contract specification for multi-language applications.

V. Conclusions

This paper presents the design, implementation and evaluation of SuperContra, a DbC framework that provides programming language agnostic contracts as-a-service, appropriate in PaaS settings. We compare the performance of our approach and prototype implementation against existing DbC frameworks via popular, open-source tools. Our results show that SuperContra performs similarly to or outperforms extant DbC approaches. The negligible runtime overhead and the reduced programming effort needed to specify the contracts across multi-language components make SuperContra an effective and easy-to-use option for improving software reliability and robustness of PaaS applications.

VI. Acknowledgements

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

References