Extracting Functional and Non-functional Contracts from Java Classes and Enterprise Java Beans

Nikola Milanovic, Miroslaw Malek
Humboldt Universität zu Berlin
{milanovi,malek}@informatik.hu-berlin.de

Abstract

We explore possibility of manual and automated contract extraction from Java classes and Enterprise Java Beans. Contracts are extended component interfaces that are defined using Contract Definition Language. They describe functional and non-functional properties. We examine how to extract hidden contracts and express them formally, modeling software components as abstract machines, hoping to achieve increased dependability at the early phases of the software development lifecycle, and to support easy and safe reuse of components later.

Indexed terms: contracts, components, composition, reuse, formal specification

1. Introduction

The software component marketplace is still in its embryonic phase. Application servers have been around for years, and still we are mostly left to ourselves to develop, test and deploy custom components, rewriting code and solving the same problems all over again. Therefore, is component marketplace a myth?

Ed Roman [7] tries to explain why independent software vendors are not shipping components to the market. He identified three reasons: maturity, politics and questionable value. He argues that since components live in application servers, application servers themselves must be mature enough before we see a market for components written for those servers. Then, there is the question of proprietary application servers, and some providers see this as a competitive advantage, resulting in non-compatible application servers or intentional information withdrawal. At the end, there is no metric to determine how good a component is, or does it really fit customer requirements.

We intend to provide contract-based formal framework for software development and reuse. It enables developers to specify functional and non-functional properties during early design and development phases, making components more reliable and less error-prone. Later, contract information may be used for safer reuse and composition.

2. Related Work

Our work is motivated in part by the research in the area of Design by Contract [5]. This paradigm develops three key questions that every component must be able to answer to: What does it maintain? What does it expect? What does it guarantee? It also lays foundations for contract extraction techniques, trying to find implicit contracts in .NET Framework libraries [4].

On the other hand, our previous work was done in the area of composability [9, 10, 6], so we tried to approach the problem from a different standpoint: what specification do we need in order to enable correct component composition? In this paper we try to use Design by Contract methods to extract contracts from Java classes and EJBs, and show how to use obtained information to develop formal component specification.

3. Contract Definition Language

Contract is how outside world perceives a component. It describes functional and non-functional properties, expectations and guarantees. Logical contract structure is shown on Figure 1. We differentiate several classes of contracts. *Base contract* specifies basic information about the component: name, URI, description, price, and available methods and events. *Method contract* describes one method of a component. It contains information about parameters, invocation, preconditions, postconditions, invariants, declared events and assertions. Invocation determines whether a component is synchronous or asynchronous. Asynchronous messages and/or callback functions are defined here. Preconditions declare obligations of a client, while postconditions declare obligations of a component. In-

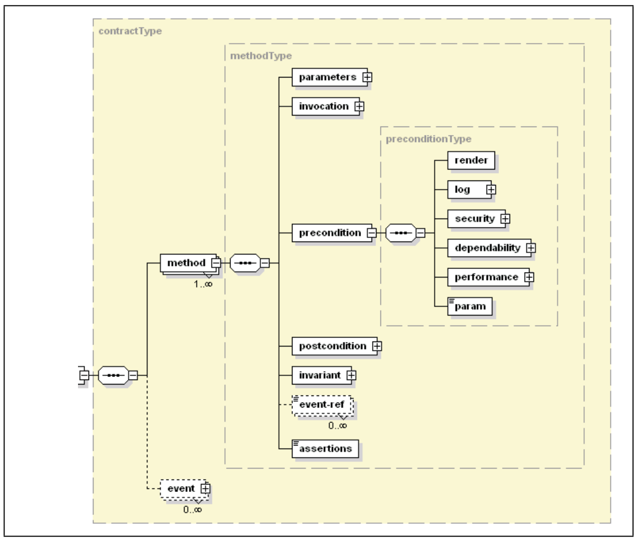

Figure 1. Logical contract structure

variants are static component properties that must hold before and after every method invocation. All three types have the same substructure, as they can describe conditions for parameters, performance, dependability, security, logging or rendering. *Event contract* is used for exposing events to which other components can subscribe. They will receive notification when an event occurs.

We also developed an XML schema called Contract Definition Language (CDL) that enables description of attributes shown on Figure 1. The XML file corresponding to this schema is the end result of the extraction process. What is the difference between CDL and Web Service Description Language (WSDL)? WSDL addresses connectivity issue. We presume that WSDL-like description is already in place, and then focus on definition of semantic and non-functional properties. The true challenge of dependable software components is not in connectivity only, but in correctness.

Before we introduce the formal model for contract elements, we will summarize why do we need specification expressed like this: contracts facilitate reuse and make it much safer, they can help in comparing and choosing between similar components, while formal specification ensures correctness.

4. Adding Formal Specification

We introduce second form of contracts: abstract machines. We need XML representation (CDL) to transport contracts over a network and to allow for easy parsing, but for formal analysis of a contract (correctness), we require mathematical representation.

We model components as *abstract machines* [1]. An abstract machine is characterized by *statics* and *dynamics*. Statics correspond to the definition of the state, while dynamics correspond to operations.

Mapping from CDL to abstract machine is done in the following way: parameters and non-functional properties become state variables, while methods become state functions. Preconditions, postcondition and invariants map directly from CDL. For each state variable a domain is defined. For example, parameter domains are defined by their language type (integer, real, etc.), while exceptions are modeled as sets. The algorithm we use for mapping properties from XML to abstract machine is outside the scope of this paper, since we concentrate on contract extraction techniques.

An abstract machine is not to be executed, but to be submitted to mathematical analysis. Let us observe the following machine:

```
MACHINE M(X,x)
CONSTRAINTS C
SETS S; T={a,b}
PROPERTIES P
VARIABLES v
INVARIANT I
ASSERTIONS J
INITIALIZATION U
OPERATIONS u <- O(w) = PRE Q THEN V END
END
```

The `CONSTRAINTS` clause specifies conditions that must hold for machine parameters. `SETS` defines allowed values for sets or deferred sets. `VARIABLES` lists state variables. `PROPERTIES` takes form of conjoined predicates involving constants and given sets, and has the role of the invariant for defined sets. `INVARIANT` also has a form of conjoined predicates stating invariants for state variables. `ASSERTIONS` can be deduced from `INVARIANT` and `PROPERTIES` clauses. It is used to ease the invariant preservation proofs. `INITIALIZATION` establishes starting values for variables. `OPERATIONS` defines abstract machine operations, with preconditions (`PRE`) and postconditions (`THEN`).

The reason we introduce formal notation is to ensure predictability and correctness of component properties. We establish both with proof obligation. It states the following: initialization and operation body must establish the invariant, while assertions must be deducible from properties and invariants. Formally we can denote this as (where $[V]I$ we denote substitution V which preserves the invariant I):

$$C \land P \land I \Rightarrow J$$

$$C \land P \Rightarrow [U]I$$

$$C \land P \land I \land J \land Q \Rightarrow [V]I$$

Using proposed model, we can improve software dependability on three levels:

- Design phase: facilitating system design through formal treatment of specification and requirements
- Early development phase: checking correctness of component implementation by proof obligation when writing code based on specification from the design phase
- Reuse and composition: comparing components based on functional and non-functional properties and guaranteeing correctness of composition by proof obligation when writing wiring code.

5. Contract Extraction

Since contracts are not a part of modern software engineering, at least in mainstream languages like Java or C#, we have to add contracts *a posteriori*, to already deployed components. Therefore, we must identify locations where to look for hidden specifications.

Elements we will be looking for are preconditions, postconditions and invariants. Preconditions are linked to exported methods and determine obligations of a client. A method is guaranteed to work correctly if and only if the client satisfies precondition. An exported method can turn precondition to its advantage, since it can assume precondition, without having to check it. This type of interaction is called *generous specification*, since we presume that client code is correct and that it respects precondition. Postcondition describes what a method guarantees, if precondition holds. Class invariants are properties that must hold before and after invocation of each exported method. They describe general, static properties of a class.

5.1. Extraction From Java Classes

We identified following locations as good candidates to look for class invariants: documentation, constructors, implemented interfaces, and base class. Invariant detection consists of two phases: identifying candidates and proving them invariant. Reading documentation is a logical place to start looking for candidates. However, it immediately confronts us with the first obstacle: it is almost impossible to provide a formal tool for documentation analysis. Therefore, human intervention will be necessary to provide insight on possible invariant candidates. After identifying candidates, we refine and augment this list by source code inspection. We look into constructors and inheritance structure, trying to find conditions applying to candidates. Sometimes new invariant candidates can be found in implemented interfaces and base class. After a list of candidate properties is established, we prove them invariant by showing that every exported method of a component preserves that property.

Preconditions are associated with exported methods, and can be extracted from documentation, conditions in exported methods that check input parameters, and exception conditions. In order to discover all preconditions, we examine all exported methods. Documentation can be of help here, but we observed a tendency that preconditions are rarely explicitly documented. In a language that supports exception handling, preconditions are usually coupled with throwing an exception. Therefore, we look for exceptions thrown by a method, and reverse conditions that precede exception throwing, or we explore calls inside a `try... catch` blocks that can raise an exception. Using this scheme we can construct what are the favorable conditions for a method, under which it will not raise an exception. This process can be automated.

Postconditions describe what a method will guarantee, presuming that preconditions are ensured. We can look for them in documentation, and return paths of exported methods. We observed that the method postconditions tend to be well documented. Sometimes, documentation is not specific about possible outcomes of a method execution, and then we must consider all return paths of all exported methods.

We can use Javadoc comments to extract contract information from Java classes. `@throws` and `@exception` tags can be used for extracting preconditions, by identifying exceptions that a method can raise. `@param` tag can be used for extracting method signature and precondition candidates. If a constructor is commented with any of these tags, we can use it to extract invariant. `@return` tag can be used for forming method postconditions. Javadoc encourages multiple return values for special cases, which facilitates tracking multiple return paths through a method. `@see` tag can be useful in tracking inheritance and dependance behavior of a given class, checking if there are conflicting requirements for identified preconditions, postconditions or invariants.

5.2. Extraction from Enterprise Java Beans

We now expand the issue of contract extraction by considering Enterprise Java Beans. Apart from locations already identified, bean preconditions, postconditions and invariants can also be found in `ejbCreate` and other CRUD (create, read, update, delete) methods, setter methods, primary key classes, finder methods, and deployment descriptors.

Knowing whether a component is session, entity or message-driven bean, we can limit the scope and focus our search. For session beans, looking into `ejbCreate` method makes sense only for stateful beans, since `ejbCreate` for stateless beans does not accept parameters. For entity beans, `ejbCreate` usually calls setters, so this is the right place to look for invariants. Depending on the type of entity bean (bean managed or container managed persistence), we check either `ejbCreate` or setter methods or underlying SQL or EJB-QL statements in deployment descriptor.

For entity beans, it is good to examine constructor of the primary key class and finder methods. The problem is the same as with `ejbCreate` method: depending on the type of persistence, sometimes it will be required to go to the level of deployment descriptor to extract useful information.

We will address hierarchy of EJB exceptions and consider which to take into account when scanning for method preconditions. Since beans are distributed by definition, it is useful to make a distinction between system-level and application-level exceptions. Every bean must throw a remote exception, indicating some special error, e.g., network or database failure. These exceptions are of no interest to us when we look for contracts. Sometimes, they are not even propagated all the way back to the client, but can be intercepted by EJB objects that act as a middleware between the client and the bean. Those exceptions are system-level exceptions. Application-level exceptions on the other hand indicate 'regular' problems, such as bad parameters passed to a bean method. Therefore, we must check for all exceptions that are propagated to the client, including all exceptions that the bean defines, and `javax.ejb.CreateException` and `javax.ejb.FindException`.

In our contract model we allow for the following types of non-functional properties: invocation, security (authentication and authorization), dependability (transactions, checkpointing, replication, exceptions), performance, rendering and logging. However, current J2EE specification defines only bean management, persistence, transactions and security. Other functions, such as load-balancing, clustering and logging, are vendor specific and we do not consider them here.

We use bean management information to form invocation part of a contract. We look into bean deployment descriptor and extract information about remote, home and local interfaces. We need this information to create and destroy beans. Then we extract information whether bean is synchronous or asynchronous. At the end, for session beans, we store information on how bean handles states (stateful or stateless). For entity beans we use the information about persistence (bean managed or container managed).

We describe component transactional behavior using transaction manager, resource, resource manager, compensate methods, transaction attribute and isolation elements. We add compensate methods to usual J2EE transaction attributes, since we want to allow components to participate in split transactions [8].

Security information encompasses authentication and authorization. An important element in J2EE authentication architecture are the login modules. Each login module implements one authentication mechanism. Therefore, one component can support multiple authentication mechanisms. We can obtain a list of login modules from configuration module. We extract the name of configuration module, as well as all login modules. Then we extract security roles. If a bean uses declarative authentication, we read `security-role-ref`, `role-name` and `role-link` elements. However, if a bean uses programmatic authentication, we must scan the source code for `getCallerPrincipal()` and `isCallerInRole(roleName)` methods. The first establishes the identity of a client, while the second one checks whether it fits in a desired role. By checking all `isCallerInRole` calls we can identify all the roles that a component supports.

5.3. Static and Dynamic Extraction

There are two ways to perform contract extraction: using static or dynamic analysis. Static analysis examines program source code and tries to reason about possible execution outcomes. We build a model of program execution state, e.g., what possible values variables can have. Then we proceed to track how they change and infer specification. Static analysis is theoretically complete [2], but can be inefficient. On the other hand, dynamic analysis is a runtime analysis of a program. We try to obtain information from program executions. Instead of trying to model execution state, we observe actual values that a running program produces. Dynamic analysis is efficient, but it is not general. Therefore, we try to combine the two methods in a hybrid approach, similar to [3]. The process we use is shown on Figure 2.

We first perform dynamic analysis, and then refine it with static analysis. In the dynamic analysis part, we identify candidate contract elements using heuristics that we described in previous sections. After this step is completed, we try to refine and/or augment identi-

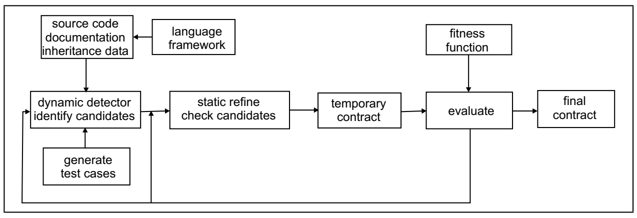
source code documentation inheritance data
language framework
fitness function
dynamic detector identify candidates
static refine check candidates
temporary contract
evaluate
final contract
generate test cases

Figure 2. Extracting functional contracts using combination of dynamic and static analysis.

fied candidates with static analysis. We inspect code, and try to prove that identified preconditions, postconditions and invariants are real. We either assume negation and prove it not possible, or cover all modeled program states with assumed candidate. In this step we can identify additional candidates.

After this phase we construct a temporary contract. The next step is evaluation of a temporary contract using fitness function. We test how well a derived specification reflects actual component behavior, trying to predict results for a given test pattern using specification. If a temporary contract is not valid, we can either perform dynamic analysis again, possibly with another set of test cases, or try to determine what is missing using static analysis, by looking deeper into source code or documentation. Once a temporary contract is evaluated as valid, we promote it into final contract, and use it as a component specification.

The main issue in contract extraction is possibility of automation. Some steps of this process cannot be fully automated. For example, finding invariants in documentation cannot be automated because of the lack of standard documentation format. Since J2EE application server configuration files are vendor specific, automatization in this area can be achieved on a vendor basis only. On the other hand, inverting conditions that cause exception throwing shows good results when automated, and there are many proposed methods for automatic generation of test cases.

6. Conclusion

The component-based software partially fails to fulfill its dependability promises due to unstructured formal specification methodologies. We propose component contracts as a solution. Using contracts we formally specify functional and non-functional properties and describe what a component requires and delivers. Our main goal was to show how to extract contract information from Java classes and EJBs.

Including contract a priori, at the start of the development process, is a great benefit and asset to any software project. Extracting contracts a posteriori can become very tedious, and we showed that in many stages it cannot be automated. Therefore, we advocate the approach where as much contractual information as possible is included at early development stages. It ensures easier extraction of remaining properties, and enables thorough formal treatment, resulting in improved component dependability and correctness.

Apart from enhancing reliability of single components, this approach can be used for providing correct component compositions. We are currently developing a composable component architecture, based on contractual descriptions [6] that allows for predicting and guaranteeing results of component composition.

References

[1] J.R. Abrial. *The B Book*. Cambridge University Press, 1996.

[2] P. M. Cousot and R. Cousot. Automatic synthesis of optimal invariant assertions: Mathematical foundations. In *Proceedings of the ACM Symposium on Artificial Intelligence and Programming Languages*, Rochester, NY, 1977.

[3] M. D. Ernst, W. G. Griswold, Y. Kataoka, and D. Notkin. Dynamically discovering program invariants involving collections. *Technical Report, University of Washington*, 2000.

[4] B. Meyer K. Arnout. Uncovering hidden contracts: The .net example. *IEEE Computer*, 36, No. 11, pp 48-55, November 2003.

[5] B. Meyer. Contracts for components. *Software Development*, 2000.

[6] N. Milanovic, V. Stantchev, J. Richling, and M. Malek. Towards adaptive and composable services. In *Proceedings of the International IPSI2003 Conference*, Sveti Stefan, Montenegro, 2003.

[7] E. Roman. *Mastering Enterprise Java Beans*. Wiley Computer Publishing, 2002.

[8] F. Tartanoglu, V. Issarny, A. Romanovsky, and N. Levy. Coordinated forward error recovery for composite web services. In *Proceeding of the 22nd International Symposium on Reliable Dependable Systems, SRDS 2003*, Florence, Italy, 2003.

[9] M. Werner and J. Richling. Komponierbarkeit nichtfunktionaler Eigenschaften - Versuch einer Definition (engl: Composability of non-functional properties — an attempt of a definition). In *GI Fachtagung Betriebssysteme*, Berlin, 2002.

[10] M. Werner, J. Richling, N. Milanovic, and Vladimir Stantchev. Composability concept for dependable embedded systems. In *Proceedings of the International Workshop on Dependable Embedded Systems in conjuction with SRDS 2003*, Florence, Italy, 2003.