Attaching Behavioral Contracts to Binary Components for Supporting Reliable Reuse*

Yang Luo, Xiaohua Yan and Jie Liu

School of Computer Science and Technology, Nanhua University, Hengyang Hunan, P.R. China

Summary
Component contract, as an interface specification, is a good idea for improving software quality. This paper describes the technique of dynamically attaching behavioral contracts *a posteriori* to binary component with no explicit contracts discipline, and presents a model based on the Common Language Infrastructure (CLI) to organize component contracts in the form of metadata and to perform efficient runtime verification. Our solution also gives a common understanding of behavioral contracts in composition even if the binary component is originally written in different programming languages. The added contract information, being easily retrieved, has a separate representation that provides flexibility, and results in raised binary component dependability and correctness on reuse and composition phase..

Key words:
components; contracts; reuse; composition; software quality; metadata.

Introduction

In order to increase productivity and speed up the development process, a large software system typically consisting of multiple interacting components, should be built through reuse rather than rewritten [1]. The resulting component libraries can then be reused across many different applications in the component-based development world, and when the software is decomposed into independently-developed third-party components, and both the source code for the component and the formal specification of the component are unavailable, constituent component quality becomes one of key factors in build a dependable software system. It is suggested that Meyer's Design by Contract^{TM} can be used to deal with the problem.

Contract is seen as a component interface specification, which is made of assertions - Boolean expressions stating individual semantic properties, between the component and its environment, specifying what the component provides its clients and what it requires from the environment in which it executes [2]. More precisely, four levels of contracts have been identified. They are syntactic, behavior, synchronization, and quantitative [3]. These contracts guarantee that methods are called properly and provide appropriate results.

Although it is now universally recognized that DbC is an important approach for improving software quality, contracts are still not a part of modern software engineering practice. Only Eiffel [4] language incorporates behavioral contracts. Researchers have been trying adding DbC to other programming languages such as Java [5, 6, 7] and C++ [8], or to other framework such as .Net [9, 10], and application developers are encouraged to think over contracts in the design phase. But DbC is seldom used in practice. One reason may be that average programmers steer clear of formal interface specification because writing and maintaining such specifications take a lot of time, and it does not give an immediate, tangible payoff. Therefore, we propose a model, which automatically attaches contracts to binary component with no explicit contracts discipline, to deal with this problem.

Our work differs from many previous works and isn't tie to specific programming languages and assertion notations. This paper intends to investigate dynamical contracts extraction techniques from already deployed component which has been built in the supporting CLI environment. The proposed model is also based on CLI, to consider contracts as an index table of metadata which separates its representation format from original component code blocks so as to provide flexibility and extensibility. The Runtime check mechanism discussed in this paper provides a common semantics of behavioral contracts in spite of (is independent of) the component development languages. We hope that the attached contract information can support reliable reuse of components.

2. Related Works

First of all, a problem to ask is whether contracts are inherent in components design; if not explicitly stated, they are lurking anyway under the cover. Karine Arnout and Bertrand Meyer' works [11] answer this conjecture. They have found some implicit program properties, including pre- and post-conditions and invariants, through checking ArrayList class from the .Net collections library.

* This work is supported by NSF of Hunan Province of China (05JJ30117).

The discovery of hidden contracts demonstrates that there are inherent contracts in component with no explicit contracts. It also lays foundations for our solution. Secondly, are there any techniques for extracting contracts from the component with no explicit contracts discipline? There have been some researches about manual and (or) automated contract extraction by finding implicit properties of program [10, 13, 12 (will change)]. Usually there are two ways, static or dynamic analysis, to perform contract extraction. Their static analysis which is theoretically complete is to inspect source code and to refine candidates. The dynamic analysis is similar to Diakon [14] and DIDUCE [15], which is efficient, and is dynamic likely invariant detector. Daikon tries to find class and loop invariants as well as routine pre- and post-conditions in the case of the source code to be available. DIDUCE does the same but not requires the program source code. It gives a perspective that it is possible to generate contracts automatically. To our knowledge, although current technologies of contracts extraction are not guaranteed to be very sound or complete, it is not impossible [14]. Our work is inspired by these researches.

3. Adding Component Contracts a posteriori

First of all, a problem to ask is whether contracts are inherent in components design; if not explicitly stated, they are lurking anyway under the cover. Karine Arnout and Bertrand Meyer' works [11] answer this conjecture. They have found some implicit program properties, including pre- and post-conditions and invariants, through checking ArrayList class from the .Net collections library.
The discovery of hidden contracts demonstrates that there are inherent contracts in component with no explicit contracts. It also lays foundations for our solution. Secondly, are there any techniques for extracting contracts from the component with no explicit contracts discipline? There have been some researches about manual and (or) automated contract extraction by finding implicit properties of program [10, 13, 12 (will change)]. Usually there are two ways, static or dynamic analysis, to perform contract extraction. Their static analysis which is theoretically complete is to inspect source code and to refine candidates. The dynamic analysis is similar to Diakon [14] and DIDUCE [15], which is efficient, and is dynamic likely invariant detector. Daikon tries to find class and loop invariants as well as routine pre- and post-conditions in the case of the source code to be available. DIDUCE does the same but not requires the program source code. It gives a perspective that it is possible to generate contracts automatically. To our knowledge, although current technologies of contracts extraction are not guaranteed to be very sound or complete, it is not impossible [14]. Our work is inspired by these researches.

3.1 Contract extraction

A detector is needed to perform contracts extraction. The detector should be similar to DIDUCE rather than to Daikon since the source code of the independently-developed third-party component is generally unavailable. In the case of the binary component, we wish to benefit from the Common Language Infrastructure (CLI) standardized by ECMA [16] and the component metadata in the contract extraction process. Implementation platform of the CLI (such as Microsoft Shared Source CLI) supports mandatory execution of components, of which all are compiled to a Common Intermediate Language (CIL). These components consist of CIL code and metadata, organized in an extensible format. The IL of any component is always available by using an IL disassembler. Furthermore, component metadata provides documentary information that makes it self-describing (e.g., the name of its class, the names of its methods, the types of its method's parameters, etc). Reflection mechanism supported by the CLI offers facility, Reflection APIs, for manipulating the metadata.

The inspection of metadata for hidden contracts is also pursued by [11, 12]. There are some possible locations to find component contracts. Preconditions inference can be implemented by parsing the CIL code to list the exceptions because preconditions tend to be buried under exception cases. A postcondition expresses properties of the state resulting from a method's execution. One can look for them in return paths of exported methods. For class invariants, constructor, interfaces and base class are good candidates.

To look for component contracts, our prototype detector that will be built in the CLI environment uses metadata to examine all classes and methods in a component, without having access to the source code. Its work principle is to discover behavioral contracts (obtain actual values) from execution traces, similar to Daikon, and is a dynamic analysis technique. By using the components metadata and reflection mechanism, the parameter information, invocation information, dependability information and security information can be obtained completely. Pre- and post-condition information can be got from evaluating metadata, attributes, and IL.

3.2 Contracts representation

We consider that the behavioral contract of a component is the collections of behavioral properties of total methods and modules in the component. In order to support runtime check, the relations between contracts and its types, methods and classes of a component are arranged into an index table, which groups all named assertions of the component. In this table, each row, mapping an assertion, consists of an index (to its owner method or type), name, flag (contracts kind) and assertion (contract context itself). This table is compiled into component metadata so that our interceptor (explained in details below) can access it by Reflection APIs. Many existing component models support users to extend metadata. The scheme based on CLI represents the added contract in the form of metadata where it becomes part of binary components and has common semantics regardless of the original development languages. It is an important issue because a common understanding of behavioral is necessary if (when) binary components are to obey contracts of one another.

3.3 Runtime verification

To perform runtime contracts check we design an interceptor, which inspects the interaction between a component and its client. The interceptor includes a component contract organized as noted above, and an exception process mechanism. When the client delivers a request to the component, the interceptor intercepts it; if behavior between component and client is consistent with contract specification, it will return directly the request to the component, otherwise it will trigger exception process (runtime violation of the contracts). *Vice versa.*

This scheme is implemented by rewriting intermediate-code. We insert the IL of an interceptor into the IL of the original component before a type or a method of the component is loaded; the resulting IL of attaching contracts is finally compiled into executable code. Obviously, the approach doesn't insert extra code into method bodies at compile time or generating wrapper methods [5, 6, 17], and doesn't write or modify a parser, either.

The interceptor is seen as a proxy which is transparent for the client (as is shown in Figure 1). This separate representation of contracts enables correct and more efficient runtime verification; Moreover, separate contract code blocks lend themselves to easy retrieval; What's more, flexibility in deciding whether verification are performed is provided when runtime verification can be expensive.

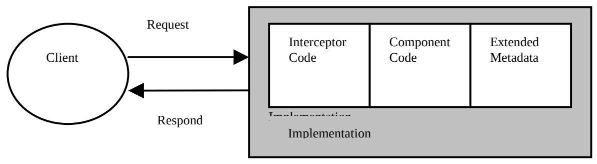

Fig. 1 Runtime check.

4. Conclusion

To achieve component quality, a common behavioral contracts system is essential. However, as stated in the introduction section, DbC is rarely applied in practice even in the presence of mission-critical tasks. Therefore, some tools, such as automatically generating component contracts based on program runs, are expected to develop. We also argue that it will become a mainstream direction in this research area. In this paper, we explore a possible way of attaching behavioral contracts a posteriori to binary component, and present a model to organize components contracts and to perform runtime check.

Using proposed model, we can improve software quality, make its reuse safer, and can facilitate in comparing and choosing among similar components. In our contract model, the contracts specification has not worked at source level except for following the DbC approach. We showed that the contract extraction is possibility of automation because of dynamic techniques. The added contracts has common semantics and representation which is independent of the source codes languages as long as the binary component has been built in CLI.

References

[1] D. Garlan, R. T. Monroe, and D. Wile. Acme: Architectural Description of Component-Based Systems. Foundations of Component-Based Systems, (Leavens G. and Sitaraman M., eds.) Cambridge University Press, pages 47–67, 2000.
[2] B. Meyer, Contracts for Components. Software Development, July, 2000.
[3] A. Beugnard, J.-M. Jezequel, N. Plouzeau, D. Watkins. Making components contract aware. Computer, 32(7), July 1999.
[4] B. Meyer. Eiffel: The Language. Object-Oriented Series. Prentice Hall, New York, NY, 1992.
[5] R.Kramer. iCountract-The Java Design by Contract Tool. IEEE Computer Society, Tools 26,1998
[6] D. Bartetzko, C. Fischer, M. Moller, et al. Jass - Java with assertions. In Workshop on Runtime Verification, 2001, held in conjunction with the 13th Conference on Computer Aided Verification, CAV'01.
[7] G. T. Leavens, K. R. M. Leino, E. Poll, C. Ruby , et al. JML: notations and tools supporting detailed design in Java. In OOPSLA 2000 Companion, pages 105-106, 2000.
[8] P.J. Maker, GNU Nana: improved support for assertions and logging in C and C++. Available from http://www.gnu.org/software/nana/manual/nana.html.
[9] K.Arnout, R. Simon. The .NET Contract Wizard: Adding Design by Contract to languages other than Eiffel. IEEE Computer Society, Tools 39, 2001, 4-23.
[10] M. Barnett, W. Schulte. Contracts, Components, and their Runtime Verification on the .NET Platform. Technical Report MS-TR-2002-38, Microsoft Research, April 2002. Available from http://research.microsoft.com/ research /pubs.

[11] K.Arnout, B.Meyer. Uncovering Hidden Contracts: The .NET Example. IEEE Computer, Nov. 2003, 36(11): 48-55.
[12] N.Milanovic, M.Malek. Extracting functional and non-functional contracts from Java classes and enterprise java beans. Proceedings of the Workshop on Architecting Dependable Systems (WADS 2004) at the International Conference on Dependable Systems and Networks (DNS 2004), Florence, Italy, 2004.
[13] J.Henkel, A.Diwan. Discovering algebraic specifications from Java classes. In L. Cardelli, editor, 17th European Conference on Software Engineering, 2003: 60-71.
[14] M. D. Ernst, J. Cockrell, W. G. Griswold, et al. Dynamically discovering likely program invariants to support program evolution. IEEE Transactions on Software Engineering, 2001, 27(2): 1-25.
[15] S. Hangal, M. S. Lam. Tracking down software bugs using automatic detection. In Proceedings of the 24th international conference on software engineering, 2002: 291-301.
[16] ECMA, Standard ECMA-335: The Common Language Infrastructure, December 2001.
[17] C.D.T. Cicalese and S. Rotenstreich. Behavioral Specification of Distributed Software Component Interfaces. Computer, July 1999.

Yang Luo is associate professor of School of Computer Science and Technology, NHU. He received the B.S. and M.S. degrees in Hunan Normal University and Central South University respectively. Since 1985, he stayed in Nanhua University to study software engineering and digital image processing.