Reuse Contracts As Component Interface Descriptions

Koen De Hondt, Carine Lucas, and Patrick Steyaert

Programming Technology Lab
Computer Science Department
Vrije Universiteit Brussel
Pleinlaan 2, B-1050 Brussel, Belgium
www: http://progwww.vub.ac.be/
email: kdehondt@vub.ac.be, clucas@vub.ac.be, prsteyae@vub.ac.be

Abstract. Current interface descriptions are poor in describing components, because they only provide an external view on a component and they do not lay down how components interact with each other. Suggestions to improve component interface descriptions at last year's workshop are reconsidered and reuse contracts are put forward as a solution that goes one step further.

1 Introduction

One of the major issues at last year's Workshop on Component-Oriented Programming was the need for more information about how a component relies on its context, than traditionally provided by the current state of the art interface description languages.

Ólafsson and Bryan [3] argued that, apart from the provided interface, a component interface description should also state the "required interfaces". A required interface is the interface of an acquaintance component that is required to enable a component to interact with that acquaintance component.

Although they argue that required interfaces are essential to understand the architecture of a component-based system, we claim that they in fact contain too little information to get a good understanding of the architecture, since an interface does not say what actually happens when one of its methods is invoked. For instance, an interface does not state the call-backs to the originating component. In our opinion, what is crucial in order to get a good understanding, is a description of the interaction structure, or the software contracts in which components participate. For this reason required interfaces are also insufficient to support component composition correctly, for they allow the composition of components that have compatible provided and required interfaces, but not the correct interaction behavior. We believe that information on interaction structure should be part of the interface of a component, so that it can be used to make the architecture clear, to help developers in adapting components to particular needs, and to verify component composition based on their interface instead of auxiliary (and perhaps informal) documentation.

In this paper, reuse contracts [4] are applied to the domain of components. It will be shown that reuse contracts are not interface descriptions to which components have to comply exactly. Instead they can be adapted by means of reuse operators. These reuse operators state how a reuse contract is adapted. By comparing reuse operators applied to a reuse contract, conflict detection can be performed and composability of components can be validated. This capacity makes reuse contracts more than just enhanced interface descriptions.

2 Reuse Contracts

Essentially, a reuse contract is an interface description for a set of collaborating participant components. It states the participants that play a role in the reuse contract, their interfaces, their acquaintance relations, and the interaction structure between acquaintances. Reuse contracts employ an extended form of Lamping's specialisation clauses [1] to document the interaction structure. While Lamping's specialisation clauses only document the self sends of an operation, specialisation clauses in reuse contracts document all inter-operation dependencies. In their most basic form, specialisation clauses in reuse contracts just list the operation signatures, without type information or semantic information, such as the order in which operations are invoked.

Reuse contracts are defined formally by Lucas [2]. Since such formal specifications are hard to read, a visual representation of reuse contracts was developed. A participant is depicted by a rectangle containing the participant's name and interface. An acquaintance relationship is depicted by a line connecting two participants. Invoked operations, together with the operations that invoke them, are notated along this line. For clarity, the line can also be annotated with the name of the acquaintance relationship. As a shortcut, self-invocations are notated in the interface of a component, instead of along an acquaintance relation with itself.

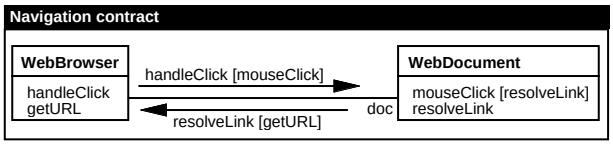

Fig. 1. Example Reuse Contract

Figure 1 shows a reuse contract for navigation in a web browser. `handleClick` on `WebBrowser` invokes `mouseClick` on `WebDocument`. `WebDocument` invokes its `resolveLink` operation when the mouse was clicked on a link (the details of the detection of the link is of no importance here). `resolveLink` invokes `getURL` on `WebBrowser` in order to get the contents of the web page pointed to by the link. For simplicity, no arguments of operations are shown here.

A reuse contract documents the assumptions each participant makes about its acquaintances. For instance, in Fig. 1 the `WebBrowser` can safely assume that the `WebDocument` may invoke `getURL` when it invokes `mouseClick`. When a component developer builds a component, he can rely on these assumptions to implement the component according to the participant descriptions. However, requesting that a component is fully compliant with the interface and interaction structure descriptions, would make reuse contracts too constraining, and consequently too impractical to use. Instead, components may deviate from the reuse contract, but the component developer has to document how they deviate exactly, so that this information can be used later on to perform conflict checking.

Therefore, reuse contracts are subject to so-called reuse operators, or modifiers, actions that adapt participants and the interaction structure between these participants. In practice, a developer performs several adaptations at once in order to reuse a component. A few basic reuse operators were identified into which such adaptations can be decomposed [2]. More general adaptations are aggregations of the basic reuse operators. Each reuse operator has an associated applicability rule, that is, a reuse operator can only be applied when certain conditions apply. Applying a reuse operator on a reuse contract results in a new reuse contract, called the derived reuse contract.

Typical reuse operators on reuse contracts are extension and refinement, and their inverse operations, cancellation and coarsening. These operators come in two flavors: one flavor handles the operations on a participant, while the other flavor handles the operation on the context of a reuse contract, being the set of participants and their acquaintance relationships. A participant extension adds new operation descriptions to one or more participants in a reuse contract. A context extension adds new participant descriptions to a reuse contract. A participant refinement adds extra operation invocations to the specialisation clauses of already existing operations. A context refinement adds extra acquaintance relationships to a reuse contract.

The top of Fig. 2 shows how a web browser component with a history to store the already viewed URLs changes the original reuse contract given in Fig. 1. This new reuse contract is the result of applying the following reuse operators to the original reuse contract: a participant extension to add `addURLtoHistory` to the interface of the browser component and a participant refinement to add `addURLtoHistory` to the specialisation clause of `getURL`. Note that the browser component's name has changed to `HistoryWebBrowser`. This is achieved through a renaming operation.

The bottom of Fig. 2 shows another adaptation of the original reuse contract. The rationale behind this adaptation is that the new document component, called `PDFViewerPluginDocument`, only contains links that point to places within the PDF document and the targets of these links can thus be retrieved by the component itself. This retrieval is achieved with a new operation `gotoPage` instead of delegating this responsibility to the browser component through `getURL`. Therefore the original navigation reuse contract is adapted as

follows: a participant coarsening removes the invocation of `getURL` from the specialisation clause of `resolvelink`, a participant extension adds the new operation `gotoPage` to the interface of `PDFViewerPluginDocument`, and a participant refinement adds the invocation of `gotoPage` to the specialisation clause of `resolvelink`. A renaming operation is also required to change the name of the document component.

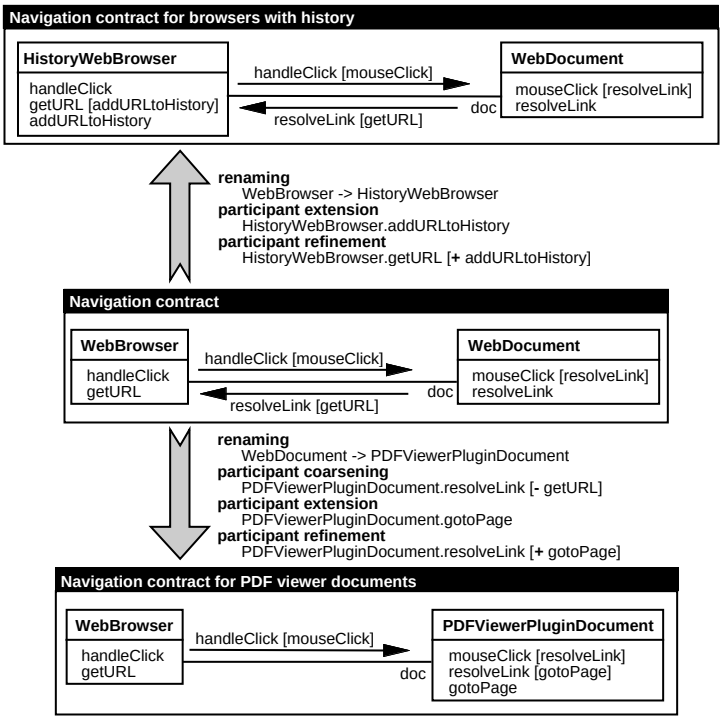

Fig. 2. Two adaptations of the original reuse contract

3 Component Composition

When a reuser now wants to combine the `HistoryWebBrowser` with the `PDFViewerPluginDocument`, he runs into trouble, because his application will not behave correctly. Since link resolving is done by the `PDFViewerPluginDocument` instead of by the `HistoryWebBrowser`, the `HistoryWebBrowser`'s history will not be updated when the user clicks on a link in a `PDFViewerPluginDocument`.

With standard interface definitions, this problem would not have been detected until the application was running, because `HistoryWebBrowser` and `PDFViewerPluginDocument` have compatible provided and required interfaces.

With reuse contracts however, this problem is detected when the two components are composed. By comparing the reuse operators that were used to derive the two reuse contracts in Fig. 2, one can easily determine what inhibits composition of `HistoryWebBrowser` and `PDFViewerPluginDocument`. The top reuse contract is derived by applying a combination of an extension and a refinement on the original reuse contract. The extension adds `addURLtoHistory` to the interface of the browser component, while the refinement adds an invocation of this operation to the specialisation clause of `getURL`. The bottom reuse contract is a coarsening of the original reuse contract: the invocation of `getURL` was removed from the specialisation clause of the document component. Based on this comparison we can conclude that `getURL` and `addURLtoHistory` have become *inconsistent operations*[2][4]: `HistoryWebBrowser` assumes that `getURL` will be invoked, so that the history can be updated (through `addURLtoHistory`), while this assumption is broken by `PDFViewerPluginDocument`.

This example illustrates but one of many problems that may inhibit component composition. A complete list of conflicts can be found elsewhere [2].

4 Conclusion

In this paper we have presented reuse contracts as enhanced component interface descriptions. Since we believe that the interaction structure between a component and its acquaintances is crucial to get a good understanding of the component architecture, and to ensure correct composition, reuse contracts not only provide the interface of a component, but they also document what interface a component requires from its acquaintances and what interaction structure is required for correct inter-component behavior.

Component evolution is an integral part of the reuse contract approach. Reuse operators define relations between reuse contracts and their derivations. When reuse contracts are evolved in parallel, the applied reuse operators can be compared to perform conflict detection. When conflicts occur, this indicates that some components cannot be composed.

References

1. John Lamping: Typing the specialization interface. Proceedings of OOPSLA'93 (Sep. 26 - Oct. 1, Washington, DC, USA), volume 28(10) of ACM Sigplan Notices, pages 201–214. ACM Press, October 1993
2. Carine Lucas: Documenting Reuse and Evolution with Reuse Contracts. PhD thesis, Vrije Universiteit Brussel, 1997
3. Asgeir Ólafsson and Bryan Doug: On the need for "required interfaces" of components. In Max Mühlhäuser, editor, Special Issues in Object-Oriented Programming, Workshop Reader of the 10th European Conference on Object-Oriented Programming, ECOOP'96, Linz, pages 159–165. dpunkt Verlag, 1997
4. Patrick Steyaert, Carine Lucas, Kim Mens, and Theo D'Hondt: Reuse contracts: Managing the evolution of reusable assets. In Proceedings of OOPSLA'96 (Oct. 6-10, San Jose, California), volume 31(10) of ACM Sigplan Notices, pages 268–285. ACM Press, 1996