Bisimulation Abstraction for Selecting Software Components in RAISE

Elsa Clara Estévez and Tomasz Janowski

Departamento de Ciencias de la Computación
Universidad Nacional del Sur
United Nations University - International Institute for Software Technology

e-mail: eestevez@criba.edu.ar

April 27, 1999

Abstract

Some questions and problems arise when working with Component-Based Design. One of these is the abstraction needed to decide suitability of a component for the design. Bisimulation is a relation between the states of two systems expressing that we cannot distinguish between them by observation. RAISE is a formal approach to industrial software engineering based on higher-order logic. In this paper we present how to apply bisimulations to compare RAISE modules. We present concepts of strong and weak bisimulations and we present examples. We conclud discussing how bisimulation supports the concept of self-contained, persistent components in RAISE, similar to object in distributed object computing.

1 Introduction

Component-based design consists on assembling software components in order to produce a software product. There is no generally accepted definition of components. A possible one is that a component is a program element

with two properties: the element may be used by other program elements (clients), and the clients and their authors do not need to know the element's authors, [Mey99]. A true component must be usable by sofware developers who build new systems not foreseen by the component's author and who are not personally known to that author. In this context of component-based design many problems arise, such as: how to specify semantics to define the behaviour of the component; abstraction to decide suitability of a component for the design; reuse in the sense of maintaining and retrieving components from a software library, and so on.

In this work we discuss the problem of how to decide if a component suits in our design, and in which way we can use bisimulation in the abstraction process to select components.

Bisimulation is a kind of invariant between the states of two dynamic systems expressing the fact that no external observer is able to distinguish between them. After it was introduced by Park [Par81] and Milner [Mil80], the concepts of bisimulation appeared in different ways such as strong, weak [Mil89], co-algebraic [JR97]. Usually, bisimulation refers to the labelled transitions $P \xrightarrow{\alpha} P'$ which represent the ability of a process P (representing a system in a certain state) to perform the action α and thereafter behave like process P'. Among the actions is the internal action τ for communication between the components of P. Given processes P and Q, a bisimulation B is a relation on their states which is preserved under the application of equally-labelled transitions, with an arbitrary sequence of internal steps between them: $(P, Q) \in B$ iff

$$\text{if } P \xrightarrow{\alpha} P' \text{ then exists } Q',\ Q \xrightarrow{\tau^*}\xrightarrow{\alpha}\xrightarrow{\tau^*} Q' \text{ and } (P', Q') \in B$$
$$\text{if } Q \xrightarrow{\alpha} Q' \text{ then exists } P',\ P \xrightarrow{\tau^*}\xrightarrow{\alpha}\xrightarrow{\tau^*} P' \text{ and } (P', Q') \in B$$

The existence of such a B, $(P, Q) \in B$, means that P and Q are equivalent, written $P \approx Q$.

RAISE, Rigorous Approach to Industrial Software Engineering, [Gro92, Gro95] is a formal approach to industrial software engineering. It provides a formal specification language *RSL*, RAISE Specification Language which allows to write software descriptions with different levels of abstraction; a refinement method to prove correctness of the different development steps; and tools to edit prove and transform into code descriptions writen in RSL. The basic block of specification in RSL is a module, which contains (in the part of the language we consider) declarations of various kinds: values, types, axioms and other modules.

In this paper we will investigate how to compare abstract and concrete modules writen in RSL using bisimulations, allowing for replacement of ab-

stract by concrete modules without noticing any external difference. Technically, bisimulation is a relation on the states of two modules which gives the same results under all observer functions and is preserved under the application of state-changing functions, with arbitrary sequence of internal functions between them. Bisimulation and its conditions could also be expressed in RSL. Bisimulation is a powerful tool for abstraction, hiding the details of implementation of modules from their users. They support, as we shall see, the concept of self-contained, persistent software components in RAISE.

The rest of this work is as follows. Section 2 introduces elements of the applicative part of RSL, its value, type and class expressions. We will also describe how that expressions can be compared, all illustrated by examples. Section 3 considers the definition of bisimulation for comparing class expressions in RSL, in particular the abstract and concrete class expressions as described earlier. We start with strong bisimulation assuming that all functions are external and total, then we will consider partial and internal functions in the definition of weak bisimulation. The final Section 4 concludes with some discussion on how bisimulations support the concept of self-contained, persistent components in RAISE.

References

[Gro92] The RAISE Method Group. *The RAISE Specification Language.* Pr. Hall, 1992.

[Gro95] The RAISE Method Group. *The RAISE Development Method.* Prentice Hall, 1995.

[JR97] B. Jacobs and J. Rutten. A tutorial on (co)algebras and (co)induction. *EATCS Bulletin*, 62:222–259, 1997.

[Mey99] Bertrand Meyer. On to Components. *Computer, IEEE Computer Society*, 32(1):139–143, 1999.

[Mil80] R. Milner. A Calculus of Communicating Systems. *LNCS*, 92, 1980.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice-Hall International, 1989.

[Par81] D. Park. Concurrency and automata on infinite sequences. *LNCS*, 104, 81.