Formal Specifications in Component-Based Development

Elsa Estévez Pablo Fillottrani

Departamento de Ciencias e Ingeniería de la Computación
Universidad Nacional del Sur
Av. Alem 1253 – (8000) Bahía Blanca
Argentina

e-mail: {ece,prf}@cs.uns.edu.ar

1 Introduction

Software engineering has entered a new era, the Internet and its associated technologies require a different conceptual framework for building and understanding software solutions. Users ask to develop applications more rapidly, and software engineers need to ensamble systems from preexisting parts. *Components* and *Components-Based Development*(CBD)[16, 15], are the approaches that provide solutions to these arising needs. Components are the way to encapsulate existing functionality, acquire third-party solutions, and build new services to support emerging business processes. Component-based development provides a design paradigm that is well suited to the new requirements, were the traditional *design and build* has been replaced by *select and integrate*. Within this approach, the specification of components plays a crucial role. If we are working on the development of components in order to construct a library for general use, we need to start from a concrete and complete specification of what we are going to construct. If we are assembling our application from pre-existing components, we need a precise specification of the behaviour of the component in order to select it from the library.

The specification is a statement of the requirements of a system. In general, we can view a specification as the statement of an agreement between the producer and a consumer of the service. In particular for software engineering, this agreement holds between the software engineer and the user. Depending on the context, these two roles may present important differences, so the nature of the specification is different. It is obvious we need to specify the system that is our final product, but also all the intermediate elements such as subsystems, components, modules, case tests, etc. Therefore, we can speak about *requirements specification* to represent the agreement between the user and the system developer, *design specification* in terms of the agreement between the system arquitect

and the implementers, *module specification* or *component specification* as the contract between the programmer using the module and the programmers who develops it. As it is shown, the term *specification* is used in different stages of system development. In all cases a specifiation at some level states the requirements for the implementation at a lower level, and we can view it as a definition of what the implementation must provide. The relation between the specification and the implementation is often explained in terms of *what* the system must provide and *how* the system will be implemented in order to provide the services. The specification must states *what* a component should do, while the implementer decides *how* to do it.

The specification activity is a critical part of the software production process. Specifications themselves are the result of a complex and creative activity, and they are subject to errors, just as are the products of other activities, like coding. As a result, we can write good specifications, or bad ones. In this sense, most of the qualities required for software products are required for specifications. The first qualities required are that they should be clear, unambiguous, and understandable. These properties are quite obvious, but sometimes specifications are written in natural language and usually hide subtle ambiguities, specially informal specifications. The second major quality required for specifications is consistency, meaning that it should not introduce contradictions. The third prime quality required for specifications is that they should be complete. Because of the difficulties in achieving complete specifications, the use of the incremental principle is esencially important in deriving specifications. That is, one may start with a fairly incomplete specification document and expand it through several steps.

There are many relevant techniques for writing specifications, so we classify them according to different specification styles. Ghezzi, Jazayeri and Mandrioli [2] classify them according to two different orthogonal criteria. Specifications can be stated *formally* or *informally*. Informal specifications are written in a natural language; they can, however, also make use of figures, tables, and other notations to help understanding. They can also be structured in a standardized way. When the notation gets a fully precise syntax and meaning, it becomes a formalism. In such a case, we talk about *formal specifications*. It is also useful to talk of *semiformal specifications*, since, in practice, we sometimes use a notation without insisting on a completely precise semantics.

2 Research Topics

On this ground, we restrict our research to the use of formal methods for the specification of software. By formal methods we mean a specification language plus formal reasoning, that includes the use of formalisms such as logic, discrete mathematics, finite state machines, and others. Formal specifications are expressed in languages whose syntax and semantics are formally defined, so they provide hierarchical decomposition and stepwise development. The main goals of using formal methods in requirements specification is to clarify customer's needs and to reveal ambiguity, inconsistency and incompleteness. By using formal specification for requirements, we can make a structural decomposition of the behaviour of the system by specifying the behaviour of each component. Furthermore, there is also the possibility of refinement by demonstrating that lower levels of abstraction

satisfies higher levels. Formal methods enables the verification of software by proving that an implementation satisfies its specification, and likewise it makes easier the validation facilitating testing and debugging. In this context we use *RAISE*, *Rigourous Approach to Industrial Software Engineering*, [3, 4] as the formal method to develop and to specify software. The *RAISE* Method provides a methodology to develop software, a formal specification language *RSL*, and automated tools for proofs and code generation.

Within the formal methods area, algebraic specification is one of the most extensively-developed approaches. The most fundamental assumption underlying algebraic specification is that programs are modelled as many-sorted algebras consisting of a collection of sets of values together with functions over those sets. The overall aim of work on algebraic specification is to provide semantic foundations for the development of software that is *correct* with respect to its requirements specifications. This means that the component developed must exhibit the required input/output behaviour. Algebraic specifications are used to construct software in a stepwise fashion, adding more details in each step of refinement [14, 8, 12].

As new paradigms appeared, different formalisms are used for the theoretical foundations. For example, the object oriented paradigm uses the mathematical concept of coalgebras to model the behaviour of classes [13, 8, 6, 7]. The concept of a class as a black box, that is fully encapsulated and cannot be examined directly, presents a typically co-algebraic view. In dealing with coalgebras, we use destructors to get a result or to change the state of an object. This is more suitable with the concept of encapsulation and information hiding, basis of the object-oriented paradigm. As another example, in component integration we need to asbtract implementation details and concentrate only on behaviour aspects. The co-algebraic view is also suitable for modeling the component as a black box.

On our research we describe software components as class expressions in RAISE. Each class consists of the state, operations for reading and writing this state and the axioms to relate such operations. On the one hand a class represents a data structure, giving rise to the definition of an abstract data type. This presents a typically algebraic view which allows us to construct and manipulate particular values of the state [14]. On the other hand, a class represents dynamic behaviour: how the component generated from this class can interact with its environment, how its operations invoked externally to produce observable attributes of the state, new states (which attributes we can investigate again) or both. Our studies are based on these two complementary approaches; the algebraic view is more adequate when we need to construct a class from its specification (by refinement)and the co-algebraic view is more suitable for reuse and composition of class expressions. In the latter we require definition of behavioural abstraction which is not supported by refinement. For this purpose we propose to use bisimulations [11, 9]. Bisimulation is a relation between the states of two dynamic systems representing that the two systems cannot be distinguished by interacting with them. It is a central concept in concurrency theory, and it was also considered on the grounds of logic [5], category theory [1], games [10] and co-algebras [8]. We study the application of bisimulation in the framework of RAISE, an industrial-strength formal method, in the role of behavioural abstraction for building software from pre-existing components. The goal is to prove how bisimulation ignores what is inessential from the point of view of behaviour, such as differences in operation definitions, data

representations, implementation structures and specification styles. We can thus establish a criteria for comparing components from the behavioural equivalence view.

References

[1] A. Joyal, M. N., and Winskel, G. Bisimulation for Open Maps. *Information and Computation 127* (1996), 164–185.

[2] Carlo Ghezzi, Mehdi Jazayeri, D. M. *Fundamentals of Software Engineering*. Prentice-Hall International, 1991.

[3] Group, T. R. M. *The RAISE Specification Language*. Pr. Hall, 1992.

[4] Group, T. R. M. *The RAISE Development Method*. Prentice Hall, 1995.

[5] Hennessy, M., and Milner, R. Algebraic Laws for Nondeterminism and Concurrency. *Journal of the ACM 32* (1985), 137–161.

[6] Jacobs, B. Coalgebraic Reasoning about Classes in Object-Oriented Languages. *Electronic Notes in Theoretical Computer Science 11* (1998), 1–12.

[7] Jacobs, B. Coalgebras in specification and verification for object oriented languages. *Newsletter of the Dutch Association for Theoretical Computer Science 3* (1999).

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

[9] Milner, R. A Calculus of Communicating Systems. *LNCS 92* (1980).

[10] Nielsen, M., and Clausen, C. Bisimulation, Games and Logic. In *CONCUR94* (1994), vol. 836 of *LNCS*, Springer-Verlag, pp. 385–400.

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

[12] Razvan Diaconescu, K. F. CafeOBJ Report: The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification. *World Scientific, 1998. AMAST Series in Computing, 6* (1998).

[13] Reichel, H. An approach to object semantics based on terminal co-algebras. *Mathematical Structures in Computer Science 5* (1995), 129–152.

[14] Sannella, D., and Tarlecki, A. Essential Concepts of Algebraic Specification and Program Development. *Formal Aspects of Computing 9(3)* (1997), 229–269.

[15] Szyperski, C. *Component Software Beyond Object-Oriented Programming*. Addison Wesley Longman Limited, 1998.

[16] W.Brown, A. *Large-Scale Component-Based Development*. Prentice Hall International, 2000.