Algebraic Specifications and Refinement for Component-Based Development using RAISE

Elsa ESTEVEZ and Pablo FILLOTTRANI
Departamento de Ciencias e Ingeniería de la Computación
Universidad Nacional del Sur
8000 Bahía Blanca, Argentina

Abstract

There are two main activities in Component-Based Development: component development, where we build libraries for general use, and component integration, where we assemble an application from existing components. In this work, we analyze how to apply algebraic specifications with refinement to component development. So we restrict our research to the use of modules that are described as class expressions in a formal specification language, and we present several refinement steps for component development, introducing in each one design decisions and implementation details. This evolution starts from the initial specification of a component as an abstract module, and finishes with the final deployment as fully implemented code.

The usage of formal tools helps to assure the correctness of each step, and provides the ground to introduce complementary techniques, such as bisimulations, for the process of component integration.

Keywords: algebraic specifications, refinement, component-based development, formal methods, software engineering

1 Introduction

Components and *Components-Based Development* (CBD) [1, 15] are the approaches that provide solutions to the new arising needs, such as Internet and its associated technologies. A *component* is an independently deliverable piece of functionality providing access to its service through interfaces [1, 9]. Components are the way to encapsulate existing functionality, acquire third-party solutions, and build new services to support emerging business processes. Within this approach, the specification of components plays a crucial role, requiring a renewed emphasis on specification and verification techniques. On one hand, if we are working on the development of components in order to construct a library for general use, we need a complete specification of what we are going to construct. On the other hand, if we are assembling our application from pre-existing components, we also need a precise specification of the behavior of the component in order to decide whether it is suitable for our design.

Many relevant techniques for writing specifications have been proposed, including formal methods. By formal methods we mean a specification language plus formal reasoning, including the use of formalisms such as logic, discrete mathematics, finite state machines, and others. Formal specifications provides a more precise definition of the functionality supplied by the software, and its well-defined syntax and semantics enable the automatization of their processing.

A structural decomposition of the behavior of the system is possible by describing the behavior of each component by formal specifications. 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. There is also the possibility of applying the notion of refinement, demonstrating that lower levels of abstraction satisfy higher levels. In this context, we use $RAISE$, *Rigorous Approach to Industrial Software Engineering*, [6, 7] 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.

Using formal methods we can write *descriptive specifications* of a system. Descriptive specifications delineate the desired properties of a system rather than its desired behavior. A natural way for precisely specify system properties is through the use of mathematical formulas. One popular descriptive specification style is based on the use of *algebra*, as the underlying mathematical formalism. Essentially, algebraic specifications define a system as a heterogeneous algebra, i.e., a collection of different sets on which several operations are defined. Algebraic specifications are used to construct software in a stepwise fashion, adding more details in each step of refinement. As new paradigms appeared, different formalisms are used to provide the theoretical foundations. For example, in the object oriented paradigm the mathematical concept of coalgebras can be used to model the behavior of classes. The concept of coalgebra as a black box, in which we can only apply functions in order to get a result or to change the state, is more suitable with the concept of encapsulation and information hiding, basis of object oriented approach.

In this work we present a classification of the refinement steps in the context of algebraic specifications for component-based development using $RAISE$, and we describe the properties these steps must exhibit in order to preserve semantics. In the next section we present the main concepts of algebraic specifications and refinement.

After that, we describe how to applied them in CBD to develop components for general use. In section 4 we present the refinement steps, and describe the application of this process in $RAISE$. Finally, we present our conclusions and related work.

2 Algebraic Specifications and Refinement

Within the formal methods area, the approach of algebraic specifications is one of the most extensively developed. In general, working with algebraic specifications implies the assumption that the correctness of the input/output behavior of a program takes precedence over all other properties. Thus, we can leave aside from concrete details of code and algorithms, and model program functionality as mathematical functions. The fundamental assumption underlying algebraic specifications is that programs are modeled as many-sorted algebras, which consist of a collection of sets of values together with functions over those sets. The overall aim is to provide semantic foundations for the development of software that is *correct* with respect to its requirements specifications. Algebraic specifications are widely used to construct traditional software systems in a stepwise fashion, adding more details in each step of refinement [11, 8, 10].

An *algebra* consists of a set of values plus operations defined over those values [12]. Thus, a program represented as a many-sorted algebra consists of a collection of sets of data values together with functions over those sets. Each algebra is associated with a *signature* that names its components, providing the basic vocabulary for making assertions about its properties. A signature Σ is the set of sorts together with some operations defined on them. The set S of sorts is called the *sort* of the signature, and the set of operations, over the set S, is usually denoted by Ω. Therefore, a *many sorted signature* is a pair $\Sigma =< S, \Omega >$, where S is a set of sorts names and Ω is an $S^* \times S$-sorted set of operation names. In this case, S^* is the set of finite, including empty, sequences of elements of S. For example, let $Numbers$ be the signature with sorts $S = \{Nat, Bool\}$, and with operations $\Omega = \{add, mult, succ, greater\_than, equal\}$ defined over S. At this point, this signature is only a syntactic structure, regardless the meaning that we can suppose based on our previous knowledge.

Let Σ be the signature of the algebra, then the class of all algebras over Σ will be denoted by $Alg(\Sigma)$. For any program P, the algebra it generates is written as $[\![P]\!] \in Alg(Sig(P))$, where $Sig(P)$ is the underlying signature of P. For example, we may have a component for implementing a buffer written in Java. In the program, we define variables b to represent the bound of the buffer, $length$ to denote the actual length of the buffer, $start$ to point to the first message allocated in the buffer, and s to model the buffer itself. There are also two functions: $send$ and $message$. The first one takes as argument a message and inserts it in s, and the second one has no arguments and returns the last inserted message. Thus, $Sig(P)$ is defined by the sorts of b, $length$, $start$, and s, plus the set containing the two functions. We can observe that program P is a particular algebra in the set of $Alg(Sig(P))$, and that there are many programs with different implementations for this signature.

In a formal approach to software development, specifications must be objects as formal as programs, thus we also need a formal language to write specifications and to derive properties from it. Whenever we write a specification SP, we determine a signature $Sig(SP)$ and a class $[\![SP]\!]$ of $Sig(SP)$-algebras. Note the overloading of the semantic bracket, in the case of a program P, $[\![P]\!]$ is an algebra, while for a specification SP, $[\![SP]\!]$ is a class of algebras. As specifications should describe only *what* is required, without constraining *how* it will be implemented, the specification SP models the class of programs which we want to view as its correct realizations. For example, the module in Java previously introduced is an algebra because it is a particular implementation for that signature. In the case of a specification, $[\![SP]\!]$ determines the class of algebras, because there are many different algebras that can implement the behavior described by that specification. So far, the semantics of a specification is in any case a class of algebras that may consist of a single algebra.

For any signature, we need a logical system for describing properties of algebras over that signature. Properties of Σ-algebras, or rather of their operations, may be described by universally-quantified equations over Σ, via the definition of what it means for a Σ-algebra A to satisfy Σ-equation ϕ, written $A \models \phi$. This also determines a notion of *logical consequence*: a set of equations Φ entails an equation ϕ, written $\Phi \models \phi$, if every algebra that satisfies all the equations in Φ also satisfies ϕ. Based on these concepts, algebraic specifications is often referred to as a "property-oriented" approach. In the scope of formal specifications, we can distinguish, at least, two classes of techniques, called respectively *model oriented* and *property oriented*. In the *model oriented* approach, the software engineer builds a *unique* model, from a choice of built-in data structures and construction primitives offered by the specification language. In the *property oriented* approach, the software engineer declares first a list of function names, and by default there may be infinite models that provide, in different ways, a function for each name. Then, the software engineer states several properties that are usually called *axioms*, because they are required properties yet not proved. Algebraic specifications are said to be property oriented because they define the signature, and also contain axioms to describe the properties that models are required to satisfy. Usually these axioms are expressed as predicates of first-order logic with equality.

Refinement provides the way to transform abstract specifications into more concrete ones [11, 3]. Given a specification SP, we need to construct a program P that is a correct realization of SP. Sannella and Tarlecki proposed [12] to proceed in a stepwise fashion, gradually enriching the original specification with more and more detail, incorporating more and more design and implementation decisions. Such decisions include choosing between the different alternative behaviors left open by the specification, such as data representation, function implementations, etc. Each decision is recorded separately, as a different step consist-

ing of a local modification to the specification. With this approach, developing a program from its specification proceeds via a sequence of such small, easily understandable and verifiable steps:

$$SP_0 \rightsquigarrow SP_1 \rightsquigarrow \cdots \rightsquigarrow SP_n$$

In this chain, SP_0 is the original requirement specification and $SP_{i-1} \rightsquigarrow SP_i$ for any $i = 1, \cdots, n$ is an individual *refinement step*. The aim is to reach a specification, in this case SP_n, that is an exact description of the program in full detail, one that incorporates all design decisions and that is a correct realization of SP.

In a formal definition of these refinement steps $SP \rightsquigarrow SP'$, we must assume that any correct realization of SP' must be a correct realization of SP. Whenever the signature of SP is equal to the signature of SP', we can define that SP' is a *refinement* of SP if and only if the class of all models of SP' are included or equal to the class of all models of SP. The definition provided by Sannella and Tarlecki states:

$$SP \rightsquigarrow SP' \quad \textbf{iff} \quad [\![SP']\!] \subseteq [\![SP]\!]$$

presupposing that $Sig(SP) = Sig(SP')$, and where $[\![SP]\!]$ represents the class of all models of the specification SP.

The definition of refinement ensures that the correctness of the final outcome of stepwise development may be inferred from the correctness of the individual steps:

$$\frac{SP_0 \rightsquigarrow SP_1 \rightsquigarrow \cdots \rightsquigarrow SP_n \quad A \in [\![SP_n]\!]}{A \in [\![SP_0]\!]}$$

If the final specification SP_n represents an individual program P, then we can note this as $[\![SP_n]\!] = \{[\![P]\!]\}$, so the conclusion that $A \in [\![SP_0]\!]$ for all $A \in [\![SP_n]\!]$ is the original statement of the program development task $[\![P]\!] \in [\![SP_0]\!]$.

In the context of CBD, we may use the idea of stepwise refinement for developing components that will be later integrated into different systems, starting the process with an algebraic specification of the component. In this way we can assure the correctness of each development step. The development of the component proceeds in a stepwise fashion introducing various additional layers of specification between the requirement definition and the final code. Each pair of successive layers reflects certain design decisions, but are still close together so that correctness proof become easier. We call each of these steps a *refinement step* between algebraic specifications and implementation [3]. Also, within algebraic specifications elementary refinement steps are called abstract implementations [5].

3 Algebraic Specifications in CBD

Component-based development provides a new design paradigm where the traditional *design and build* approach has been replaced by *select and integrate*. All aspects of software design, implementation, deployment, and evolution are affected when a CBD approach is followed. As a result, a software project can be transformed from a development-intensive grind of code writing and bug fixing, to a more controlled assembly process in which new code development is minimized, and system upgrade becomes the task of replacement of well-bounded functional units of the system. In this way, components promotes software reuse, i.e. the process of creating software systems from predefined software components. Software reuse has two sides: on one hand the systematic development of reusable components, and on the other hand the systematic reuse of these components as building blocks to create new systems.

For CBD, a component is much more than a subroutine in a modular programming approach, an object or class in an object-oriented system, or a package in a system model. In CBD the notion of a component both subsumes and expands on those ideas, and also extends the notion of abstract data type. A component is used as the basis for design, implementation, and maintenance of component-based systems. A general notion of a component is given in the following definition:

> a component is an independently deliverable piece of functionality providing access to its service through interfaces [1, 9]

This definition, while informal, stresses a number of important aspects of a component. First, it defines a component as a deliverable unit. Hence, it has characteristics of an executable package of software with the corresponding separation between the component specification from its implementation. Second, it says a component provides some useful functionality that has been collected together to satisfy some needs. So, it has been designed to offer that functionality based on some design criteria. Third, a component offers services through interfaces. Using the component requires making requests through those interfaces, not accessing its internal implementation details.

Analyzing the success of component-based technology in hardware, we can notice that it has only been possible thanks to precise techniques for specifying components, and most importantly, for quality assurance and quality control. Consequently, there are two main issues to consider: the provision of concrete and precise specification of the component interfaces and behavior, and the assurance of the component quality. Therefore, by applying formal methods and algebraic specification with refinement in the component development process we can simultaneously address both topics. Using formal specifications to represent software components presents several advantages, such as producing higher quality software, providing implementation verifiability, allowing automatic translation into code, and facilitating the retrieval in component libraries.

In the development of reusable components, formal methods can help to promote software reuse. Components that have been formally specified and sufficiently well documented can be identified, reused, and combined in a new system. Also, it is important to focus on the reuse of formally developed specifications as well as formally developed code, as such reuse can improve the generality versus specialization trade-off. Formal specifications are written at a high level of abstraction with ideally, no bias toward particular implementations. During the refinement process abstract specifications are translated into more con-

crete specifications, resulting in a representation that can be executed in a programming language. Reusing specifications rather than source code makes it possible to use different implementations in different environments, applying the most appropriate implementation for a particular environment.

4 Applying RAISE

In order to apply the previously mentioned ideas, we need to take into account how the formal language supports the notion of refinement. Different formal specification languages consider it in different ways. For example, Sannella [13] presented an implementation of the notion of refinement in *CASL* and *Extended ML*, and Derrick *et al* studied refinement in Object-Z [2, 14]. In our work, we have selected *RAISE* because it supports the entire life cycle, providing not only the formal specification language *RSL*, but also techniques and strategies for doing formal development and proofs. Also, *RSL* provides a wide variety of specification styles and notations, allowing to write model-oriented and property-oriented specifications, and in applicative (functional) and concurrent (process algebra) styles.

In CBD, we must be able to decompose the description of a system into components, and compose the system from previously developed components. In these cases, the specification of a module acts as the contract between the developer and the user. The role of the contract plays a crucial role in separate development. For example, suppose that during the development of module B we want to use module A. The initial versions of B and A are B_0 and A_0 respectively, and these are developed in n and m steps to B_n and A_m. The module A_0 acts as the contract between the two developments, and when the development of B is complete, we integrate by using A_m instead of A_0 as the final development step. Our issue is that the final system composed by $A_m + B_n$ meet the original requirements. The sufficient conditions are that the initial modules A_0 and B_0 together meet the requirements, and that each development step of A and B is an implementation step.

Let module A_0 be developed to module A_1, the key point is how we can argue that A_1 is a correct development. We say that A_1 is correct if it *implements* A_0, or equivalently if A_1 and A_0 are in the implementation relation. In the previously explained context, this means that developing A_1 from A_0 constitutes a refinement step. The notion of implementation applied in *RAISE* states that A_1 implements A_0 if and only if both of the following properties hold:

- **Property preservation**: all properties that can be proved about A_0 can also be proved for A_1.
- **Substitutivity**: an instance of A_0 can be replaced by an instance of A_1, and the resulting new specification should implement the earlier specification.

Property preservation means that if we prove some properties of a module, and also we prove that the result of a development step implements it, then we know that the resulting module also has the same properties. In fact, it ensures that the implementation is transitive: if A_2 implements A_1 and A_1 implements A_0, then A_2 implements A_0. So, we can proceed from the initial specification to the final one in a number of steps. In *RAISE* the implementation relation is formally defined as: a class expression A_1 implements a class expression A_0 if all the properties of A_0 are true in the context of A_1. That is, the properties of A_1 must imply the properties of A_0. In this case, the properties of a class expression are the collection of logical expressions that can be deduced from its definitions and axioms. The collection of logical properties of a specification is the *theory* of a specification. Thus, the essential idea of implementation in *RAISE* is that the theory of the implementation needs to imply the theory of the class being implemented.

A development in *RAISE* begins with an abstract specification and gradually evolves to a concrete implementation. All steps of the development are documented as class expressions in *RSL*. The development is based on the *refinement relation* which, as it was previously explained, it stipulates preservation of signatures and preservation of properties. Therefore, one class expression implements or refines another one if every provable consequence of the theory of the latter is a provable consequence of the theory of the former.

We propose to apply algebraic specifications for describing software components as classes in *RAISE*, in order to develop individual components by incremental refinement, starting from their abstract descriptions. Initially, we provide an abstract specification, specifying the signature and properties that must be observed. In *RSL* this means that we need to specify the types used, and the names and signatures of operations. We define the behavior in an axiomatic way, expressing properties as axioms. To assure that the specification is complete, we divide functions in two classes: *generators* and *observers*. Assuming we intend to define a module, that probably has an internal state, generators are functions that modify that state, the type of interest appears in the result of the operation. Observers are functions where this type, representing the internal state, appears as an argument and produces some result. We need to specify axioms to describe the result of each observer after applying the generators.

Once the abstract specification is provided, we incrementally introduce the different steps of refinement. We will use the specification shown in figure 1 in order to illustrate each step. We can show correctness of these refinement steps by proving the axioms of the algebraic specification, in each of the generated classes.

The refinement steps are the following:

1. Replace abstract types with concrete ones: the operations and axioms defined in abstract types give them the structure. In contrast, concrete types are built from sets, lists, functions, maps, etc, and their structures are given explicitly. Normally, refinement from abstract to concrete types involves redefining functions to take into account this explicit structure. Verification involves proving that such new definitions satisfy the axioms. In figure 2 we apply this kind of step to abstract type `State`.
2. Replace implicit definitions of functions with explicit

```
PARAMETERS
scheme RINGAB (P:PARAMETERS) =
 class
  type State
  value
                /∗ generators ∗/
    send : P.Message × State ~→ State,
                /∗ observers ∗/
    current : State → P.Node,
    message : State ~→ P.Message,
    empty : State → Bool
  axiom
                /∗ current, empty, message − send ∗/
    (∀ s : State, m : P.Message •
        empty(s) ⇒
              current(send(m, s)) = current(s) ∧
              ∼empty(send(m, s)) ∧
              message(send(m, s)) = m),
...
```

Figure 1: Algebraic specification in RSL.

```
...
  type
    State = P.Node × P.Transit
  value
    send : P.Message × State ~→ State
    send(m, s) as s1
        post current(s1) = current(s) ∧
             message(s1) = m
        pre empty(s),
...
```

Figure 2: Step 1: Concrete types.

definitions: implicit definitions express the result of a function by a logical property which holds between the values of the arguments and the value of the result. They are useful for hiding details of implementation. However, we must eventually make such definitions explicit, in order to allow for their execution. Once we have defined concrete types, we can use the specific constructors of these types to make explicit function definitions. Figure 3 shows the explicit definition of function `send` generated by the postconditions in the specification.

3. Transform partial functions into total: we can only know about the value returned by a partial function provided its pre-condition is satisfied, otherwise the value may be arbitrary. Pre-conditions allow us to postpone decisions of how the function should behave for its enlarged domain; such decisions can be done during this refinement step. Figure 4 applies this step also to function `send`.
4. Change from the applicative style to imperative: the functional style of specifying classes makes explicit all dependencies between the results of a function and the values of its arguments. In this sense, the absence of side-effects helps in the development and proofs. However, implementation often requires to

```
...
    send(m, s) ≡ let (n, t) = s in (n, P.ins(m)) end
        pre empty(s),
...
```

Figure 3: Step 2: Implicit to Explicit.

```
...
    send : P.Message × State → State
    send(m, s) ≡
       if empty(s) then
            let (n, t) = s in (n, P.ins(m)) end
       else s end,
...
```

Figure 4: Step 3: Partial to Total.

```
...
    send : P.Message → read s write s Unit
    send(m) ≡
       if empty() then
            let (n, t) = s in
                s := (n, P.ins(m)) end end,
...
```

Figure 5: Step 4: Applicative to Imperative.

introduce variables, and to allow functions to refer to and change their values. In this step we introduce variables, assignments, and loops, as it is shown in figure 5.

5. Change from the imperative style to concurrent: this step can be done following the procedure described in [7]. The modifications include the introduction of an instance of the imperative module, changing function definitions using channels for communications (as it is shown in figure 6), and the definition of additional functions to manage concurrency.
6. Translate into code: the final step is the generation of the source code. *RAISE* includes automatic tools to generate *C++* or *ADA* code, but the component can be written in any programming language.

5 Conclusions

In CBD, when we develop a system of any size we must be able to decompose their description into components and assemble the system from the developed components. These developed components may be implemented by the same development team, extracted from a library, or developed by other groups. In any case, it is needed a contract between the developers and the users. This contract represents an agreement between these two parts. For the developer, a contract says what he must provide, and so he can proceed freely as long as he preserves the properties that are stated in the contract. For the user, the contract states what he may assume. A specification of a module may act as this contract, precisely stating what are the essential properties of the artifact being specified. In some way, specifications are better than something written in a programming language because they can express the essentials ignoring irrelevances. We proposed to use formal methods, particularly algebraic specifications with refinement, to develop components for general use in CBD. This approach is useful not only in the process of component development, but also in evaluating the semantics of the integrated system.

Our approach is based on algebraic specifications, which emphasize procedural abstraction since it declares the properties the software should satisfy in terms of opera-

```
...
    send : P.Message → in any out any Unit
    send (m) ≡ CH.send!(m),
...
```

Figure 6: Step 5: Imperative to Concurrent.

tions that can observe the state, or can generate new states. We presented how to specify software components as classes in *RSL*, beginning with a very abstract specification based on algebraic specification. We introduced the notion of refinement, and applied it in *RAISE*. During the development of a component, we use abstraction to enable a stepwise development, so that we can formally prove that the refinement conditions are valid. In doing so, *RAISE* provides tools to verify that a concrete class satisfies a refinement relation with an abstract one, *i.e.* the former accomplishes all the properties of the latter. The final result is that we can produce correct source code for the component.

This approach can be complemented with another one, based on coalgebraic specification, which evaluates the behavior of integrated components [4]. In this case, coalgebras are used as the underlying formalism of specifications. Coalgebras are introduced as dual to algebras in category theory. Using coalgebras, the system is modeled as a black-box to which one only has limited access via specified operations. Jacobs *et al* [8], have proposed a coalgebraic formalism to describe some of the basic concepts of object-oriented programming. Coalgebras, are also used to represent dynamical systems with a hidden state to which the user has limited access via specified operations. Essentially, the difference between algebras and coalgebras, is the difference between construction and observation. In this sense, as coalgebras are based on observations, we can study the observable outcomes of two processes that might give rise to the same sequence of observations, without actually having the same implementations. In such case, the processes are indistinguishable. Since we modeled the systems as coalgebras, we assume that we do not have access to the internal details, thus we cannot relate the states by the equality relation.

We argue that algebraic specification with refinement, and coalgebraic specifications with bisimulation are complementary approaches that can be used during requirements specification. Algebraic specifications and refinement can be applied when developing components from scratch as it was shown in this work. Coalgebraic specifications and bisimulations can be used to reason about the behavior of the system when we assemble components from a library, and we do not have access to the internal details of them. Particularly, we apply it in CBD to compare the behavior of the abstract specification of the system, with the specification resulting after assembling pre-existing components. Thus, we provide a formal mechanism to prove that a system build up from different components presents the same behavior as the initial abstract specification.

References

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

[2] John Derrick and Eerke Boiten. *Refinement in Z and Object-Z: Foundations and Advanced Applications*. Formal Approaches to Computing and Information Technology. Springer, May 2001.

[3] Hartmut Ehrig and Hans-Jorg Kreowski. Refinement and Implementation. In E. Astesiano, H.-J. Kreowski, and B. Krieg-Bruckner, editors, *Algebraic Foundations of System Specification*, pages 201–242. Springer, 1999.

[4] Elsa Estévez and Pablo Fillottrani. Bisimulation for component-based development. *Journal of Computer Science and Technology*, 2(6):67–80, 2002.

[5] Marie-Claude Gaudel and Gilles Bernot. The Role of Formal Specifications. In E. Astesiano, H.-J. Kreowski, and B. Krieg-Bruckner, editors, *Algebraic Foundations of System Specification*, pages 1–12. Springer, 1999.

[6] The RAISE Method Group. *The RAISE Specification Language*. Prentice Hall, 1992.

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

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

[9] Bertrand Meyer. On to components. *Computer, Innovative Technology for Computer Professionals, IEEE Computer*, January(4):139–140, 1999.

[10] Kokichi Futatsugi Razvan Diaconescu. CafeOBJ Report: The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification. World Scientific, 6, 1998.

[11] D. Sannella and A. Tarlecki. Essential concepts of algebraic specification and program development. Formal Aspects of Computing, 9(3):229–269, 1997.

[12] D. Sannella and A. Tarlecki. Algebraic Preliminaries. In *Algebraic Foundations of System Specification*, pages 13–30. Springer, 1999.

[13] Donald Sannella. Algebraic specification and program development by stepwise refinement. In Annalisa Bossi, editor, *Logic Programming Synthesis and Transformation, 9th International Workshop, LOPSTR'99, Venezia, Italy, September 22-24, 1999, Selected Papers*, volume 1817 of *Lecture Notes in Computer Science*, pages 1–9. Springer, 2000.

[14] G. Smith and J. Derrick. Specification, refinement and verification of concurrent systems - an integration of Object-Z and CSP. *Formal Methods in Systems Design*, 18:249–284, May 2001.

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