Bisimulation for Component-Based Development

Elsa Estévez Pablo R. 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

Abstract. Guaranteeing that assembled components will behave as required is one of the main aspects in working with Component-Based Development. In this paper we present a formal approach for tackling this problem by applying the concept of bisimulation, originally presented in the study of concurrency theory. Bisimulation allows us to abstract details that are irrelevant from the behavioral point of view, such as data representations and implementation structures, providing a powerful formalism for proving software correctness properties. Thus, our approach facilitates to demonstrate the behavioral equivalence between the integrated system and the required specification. We introduce these concepts with the help of an example described in RAISE.

1 Introduction

The specification of a software system is a precise statement of its requirements. In general, we can view a specification as the statement of an agreement between the producer and a consumer of the service. In 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, and so on. 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 architect and the implementers, *module specification* or *component specification* as the contract between the programmer using the module and the programmers who develops it. Therefore, the term *specification* is used in different stages of system development. In all cases, a specification 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 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 required quality is 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. The second major required quality is consistency, meaning that it should not introduce contradictions. The third prime quality for specifications is that they should be complete.

There are many relevant techniques for writing specifications, and they can be classified according to different specification styles. Ghezzi, Jazayeri and Mandrioli [Carlo Ghezzi, 1991] classify them according to two different orthogonal criteria. Specifications can be stated *formally* or *informally*. Informal specifications are written in a natural language, however they may 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.

The second major distinction between different specification styles is between operational and descriptive specifications. *Operational specifications* describe the intended system by describing the desired behavior. Usually, this is done by providing a model of the system, *i.e.*, an abstract device that in some way can simulate its behavior. Data Flow Diagrams [Marco, 1979,Yourdon, 1989] and Finite State Machines [Davis, 1993] are the most widely known techniques for expressing operational specifications. By contrast, *descriptive specifications* try to state the desired properties of the system in a purely declarative fashion, *i.e.* by describing system properties rather than the desired behavior. A natural way of precisely specifying system properties is through the use of mathematical formulas. Unlike natural language, mathematical formulas have a formal syntax and semantics. Many mathematical formalisms have been proposed for the description of system properties. We mention two major approaches: one based on the use of mathematical logic–in the style of the so-called Floyd-Hoare specifications ponercitas, and the other based on the use of algebra.

Essentially, algebraic specifications [Astesiano and Kreowski, 1999] define a system as a heterogeneous algebra, *i.e.*, a collection of different set 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 you 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 this approach.

Nowadays software engineering has entered a new era, the Internet and its associated technologies require a different paradigm for building and understanding software solutions. Users ask to develop applications more rapidly, and software engineers need to construct systems from preexisting parts. *Components* and *Components-Based Development*(CBD) [Brown, 2000,Szyperski, 1997] are the approaches that satisfy the

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 well suited to today´s approach, were the traditional *design and build* has been replaced by *select and integrate*.

In the process of assembling components one of the main problems is to prove that the observable behavior of the composed system is equivalent to what is specified. Since we are possibly assembling third-party components, in these cases we cannot evaluate the internal aspects of them, just because we only have the executable code. In this paper we present the application of the concept of *bisimulation* to the study of component behavior. Intuitively, we can say that two processes are *bisimilar* if they are indistinguishable with respect to their behavior. Based on formal specifications and co-algebras we consider the problem of behavioral abstraction for deciding which components can be used in a given design. These concepts are presented using class expressions in RAISE [Group, 1992,Group, 1995], which not only provides a formal specification language (RSL) but also includes automatic tools for verifying proofs.

The paper is organized as follows. In the next section we present component-based development, followed by an introduction to bisimulation and the general class definition in RSL to which the bisimulation concepts will be tailored. Section 3 contains the presentation of the case study we use as an example. In section 4 we present an initial definition of bisimulation in the context of CBD, and then extend it through several steps in order to be applicable to more scenarios. Finally we compare it with related formalisms, and present conclusions and future work.

2 Component-Based Development

The key to understand CBD is to gain a deeper appreciation of what is meant by a component, and how components form the basic building blocks of a software solution. The notion of a component subsumes and expands the ideas of a subroutine in a modular programming approach, or a class in an object-oriented system, or a package in a system model. They are used as the basis for design, implementation, and maintenance of component-based systems.

A general broad definition of a component[Brown, 2000,Meyer, 1989] states that

> a component is an independently deliverable piece of functionality providing access to its service through interfaces

This definition, while informal, stands out 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. Second, it says a component provides some useful functionality that has been collected together to satisfy some need. 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, in contrast to accessing the internal implementation details of the component. Based on this, a component can be seen as a convenient way to package

object implementations, and to make them available for assembly into a larger software system.

From this perspective, a component is a collection of one or more objects implementations within the context of a component model. This component model defines a set of rules that must be followed by the component to make those object implementations accessible to others. Furthermore, it describes a set of standard services that can be assumed by assemblers of component-based systems, such those for naming of components and their operations, security of access to those operations, transaction management, and so on. Some of the most widely known component models are Enterprise JavaBeans (EJB) with standards from Sun Microsystems, and COM+ a standard from Microsoft.

The above definition of a component does not place any requirements on how the components are implemented. As a result, valid components could include previous developed components from an earlier projects, or components acquired to third-party developers. In any case, it involves a process of selection and finally extraction of the candidate component from a repository, or as it is usually called a *Component Library*. It is also possible to create new components by wrapping legacy code or legacy data.

When selecting a component from a library, or when developing a new component, it is necessary to know a precise definition of its behavior in order to be able to integrate the component in a system. To assure that the software built by assembling components behave as it is specified, we must prove the behavioral equivalence between the specification and the concrete implementation. Bisimulation will be used for this task.

3 Bisimulation

The behavior of a process can be described using states and transitions. A transition models one of the possible actions of the process, that can be applied to one state and produce a new state. Let P be a process, A a set of actions, S the set of all possible states of P, and $\alpha \in A$ an action that may be applied to P when the process is in state s. We will denote as $s \xrightarrow{\alpha} s'$ that after this application the system evolves to state s'.

Let P_1 and P_2 be processes with set of states S_1 and S_2 respectively, so that $s_1, s_1' \in S_1$ and $s_2, s_2' \in S_2$, and suppose all members of A are equally-labelled transitions applied to both. A bisimulation between P_1 and P_2 is a binary relation $B \subseteq S1 \times S2 \bullet (s1, s2) \in B$ **iff**

$$s1 \xrightarrow{\alpha} s1' \Rightarrow \exists s2' \bullet s2 \xrightarrow{\alpha} s2' \wedge (s1', s2') \in B \qquad \wedge$$
$$s2 \xrightarrow{\alpha} s2' \Rightarrow \exists s1' \bullet s1 \xrightarrow{\alpha} s1' \wedge (s1', s2') \in B$$

This means that if two processes are bisimilar and we have two states of them that belong to the bisimulation, then applying action α to one of the processes means that it is possible to apply the same action to the second process, and the resulting states will be again in the bisimulation relation. Intuitively, a bisimulation is a relation between the states of two dynamic systems representing that the two systems cannot

```
scheme AM(P..) = class
  type
    T = P.T,
    State = ..
  value
    init: State,
    attb: T × State → T,
    proc: T × State → State,
    other: T × State → State × T
  axiom
    ..
end
```

Fig. 1. Schematic class definition in *RSL*.

be distinguished by interacting with them. It's been a central concept in concurrency theory, but it was also considered on the grounds of logic [Hennessy and Milner, 1985], category theory [Joyal et al., 1996], games [Nielsen and Clausen, 1994] and co-algebras [Jacobs and Rutten, 1997]. Within this approach we consider bisimulation as a formalism to compare and study the behavior of a system built up by components.

Components will be specified in *RSL*, where each specification is defined as a set of class definitions. Each class consists of its internal state, together with operations for reading and writing this state. The state is fully encapsulated and cannot be constructed or examined directly.

To study the behavior of the process, we can make abstractions about the data structure of the module, or about the algorithms, but in any case we need to specify the possible observations outside the module. In this sense we can view the module as a black box. For doing so, we consider the internal data structure of the module as an abstract data type that we are going to call *State*. In order to observe properties of the state, we need functions that allow us to observe and to modify it. The former are functions that take the state as an argument and produces a result of some type. These functions are called *observers*. The latter are *procedure* functions that change the state of a process, taking arguments of some type together with the current state, and producing a new state as a result. It is also possible to combine these two types of functions and have a function that take some parameters and the state as arguments, and produce some result and a new state.

In figure 1 we show a schematic class in *RSL* with examples of all the above mentioned kind of functions. This class will be used in the following section in order to present the definitions of bisimulation variants. Modules thus defined emphasize only the essential aspects. It is allowed that each operation takes its own arguments, different from the others, just because with T we mean a product of types or just no argument. In the same way, the results may be different. Also, we do not take care, at this level, about the set of internal operations of the module.

In CBD we build our system by assembling components. In this case, the state of the system can be modelled as the Cartesian product of the states of all the compounded

components. Also, observer and procedure functions of the system will be defined using observer and procedure functions of the basic components. The key question is if the composite system will behave as it was specified. At this level, we consider the implementation of the system as a concrete module. Further, we will show how this concrete module is a composition of individual components. In order to prove that both have an equivalent behavior, we define a bisimulation relation between an abstract module representing the specification of the system, and a concrete module representing its implementation.

Let AM be an abstract module with the form shown in figure 1, and CM a concrete module that represents an implementation of AM. Suppose that AM and CM are parameterized by other module such as P, containing all necessary type definitions. A and C are objects that instantiate modules AM and CM, with $A.State$ and $C.State$ representing their internal states respectively. Then we can define the bisimulation relation in RSL as:

```
object
  P, ...
  A: AM(P), C: CM(P) ...
type
  Relation = (A.State × C.State)-set
value
  bisimulation: Relation → Bool
  bisimulation(rel) ≡
    (∀ sa : A.State, sc : C.State • (sa,sc) ∈ rel ⇒
      (∀ t: P.T •
        A.attb(t,sa) = C.attb(t,sc) ∧
        (A.proc(t,sa), C.proc(t,sc)) ∈ rel ∧
        let (sa′,ta)=A.other(t,sa), (sc′,tc)=C.other(t,sc) in
            ta=tc ∧ (sa′,sc′) ∈ rel end))
```

This definition needs to be customized for comparing individual classes, according to their signatures: the number of operations, the number and types of arguments, types of results, etc.

In order to assert that two classes L and R are behaviorally equivalent we state

```
value
  Relation: A.State × C.State-set
theorem
  bisimulation(bis) ∧ (L.init,R.init) ∈ bis
```

Therefore, we demonstrate that there exists a concrete relation between their states which is a bisimulation, and that contains the pair of initial states. Thus,bisimulation provides a basic formalization for a behavioural equivalence between two classes.

4 Application Example

We will illustrate the application of bisimulation in CBD by analyzing the example of a ring of nodes that need to send messages between them. The ring is able to handle

```
scheme PARAMETERS =
  class
    value
      nodes : Nat • nodes > 0
    type
       Node = {| n : Nat • n > 0 ∧ n ≤ nodes |},
       Message
    value
      receiver : Message → Node,
      No_Message : Message
  end
```

Fig. 2. *PARAMETERS* module specification.

several messages at a time. The component uses a module *PARAMETERS*, shown in figure 2, that contains the definition of all specific types and values needed. A value *nodes* is defined such that it specifies the number of nodes in the ring. Types $Node$ and $Message$ are included to represent the current node in the ring and the kind of messages handled. We need a function *receiver* to receive a message; we suppose this value is encoded in the message module. There is also a value $No\_Message$ to represent that there is no messages in the ring.

In order to write a specification for this module, we leave open as many as possible alternative development routes. Thus, we write an abstract specification using the abstract type *State*, the values representing the procedures and the observers, and a list of axioms relating the observers with the generators. In Figure 3 we show the abstract specification of the ring on the left, which corresponds to the concrete specification on the right. The ring provides producer *init* for defining the initial state of the ring; *next* which allows the ring to set the position of the following node; *send* to send one message; and *receive* for receiving a message in a node. A node can receive a message if it is the current node of the state, and if the receiver of the message is equal to that node. If the ring has a message, and the receiver of that message is the current node of the state, after applying function *receive*, the state of the ring is empty. The component has three observers: *current* which returns the actual node of the state, *empty* that returns a boolean value indicating whether there is a message in the state, and *message* that returns the message that has the higher priority.

For these specifications, we define a bisimulation between the abstract and the concrete specifications as follows:

```
  object
    P:PARAMETERS,
    A:RING(P),
    C:RING1(P)
  value
    bisimulation: Relation → Bool
    bisimulation(rel) ≡
      (∀ as : A.State, cs : C.State •
        (as, cs) ∈ rel ⇒
          (A.current(as) = C.current(cs)) ∧
          (A.empty(as) = C.empty(cs)) ∧
```

```
PARAMETERS
scheme RING (P:PARAMETERS) =
 class
  type State
  value
   /∗ generators or procedures ∗/
   init : State,
   next : State → State,
   send : P.Message × State ~→ State,
   receive : State → State,
   /∗ observers or attributes ∗/
   current : State → P.Node,
   message : State ~→ P.Message,
   empty : State → Bool
  axiom
   current(init) = 1,
   empty(init),
   (∀ s : State •
      current(next(s))=(current(s) + 1)/P.nodes ∧
      empty(next(s)) = empty(s) ∧
      ∼empty(s) ⇒ message(next(s))= message(s)),
   (∀ s : State, m : P.Message •
       empty(s) ⇒ current(send(m, s)) = current(s) ∧
         ∼empty(send(m, s)) ∧ message(send(m, s)) = m),
   (∀ s : State • current(receive(s)) = current(s)),
   (∀ s : State • current(s) = P.receiver(message(s))
       ⇒ empty(receive(s))),
   (∀ s : State • current(s) ≠ P.receiver(message(s))
       ⇒ empty(receive(s)) = empty(s)),
   (∀ s : State •
       ∼empty(s) ∧ (current(s) ≠ P.receiver(message(s)))
       ⇒ message(receive(s)) = message(s))
end
```

```
PARAMETERS
scheme RING1(P : PARAMETERS) =
  class
    type State = P.Node × P.Message*
    value
      init : State = (1, ⟨⟩),
      next : State → State
      next(n, l) ≡ ((n + 1) \ P.nodes, l),
      send : P.Message × State → State
      send(m, (n,l)) ≡ (n, l ⌢ ⟨m⟩),
      receive : State → State
      receive(n, l) ≡
          if ∼empty(n,l) ∧
              current(n,l) = P.receiver(hd(l))
              then (n, tl l)
              else s
          end
        end,
      current : State → P.Node
      current(n, l) ≡ n,
      empty : State → Bool
      empty(n, l) ≡ l = ⟨⟩,
      message : State → P.Message
      message (n, l) ≡ hd l
  end
```

Fig. 3. Abstract and a Concrete Specifications of a Ring of Nodes.

```
(A.message(as) = C.message(cs)) ∧
(∀ m:P.Message •
   (A.send(m,as),C.send(m,cs)) ∈ rel) ∧
   (A.receive(as),C.receive(cs)) ∈ rel)))
```

This definition follows from applying the general schema of the previous section to the classes shown in Figure 3.

The traditional concept of bisimulation introduced so far for comparing classes is too rigid. We assumed that two classes have the same sets of functions, that corresponding functions have the same signatures, that they are defined for all values of their arguments, and so on. In the following section, we introduce variations of this definition in order to make it more flexible.

5 Adapting Bisimulation to CBD

First we extend the bisimulation concept in order to handle partial functions. By a partial function we mean a relation that fails to be a function because it is not aplicable for certain arguments in the domain. Figure 4 shows a schematic specification of a module with partial functions. To express these constraints we add pre-conditions

```
scheme AM(P..) = class
  type
    T = P.T,
    State = ..
  value
    init: State,
    attb: T × State → T,
    proc: T × State ~→ State,
    other: T × State ~→ State × T,
    pre_proc,pre_other: T × State → Bool
  axiom
    ..
end
```

Fig. 4. Schematic class specification with partial functions.

to procedure definitions, so as to specify the legitimate values for their arguments. In the above specification, functions *pre_proc* and *pre_other* are the preconditions of procedures *proc* and *other* respectively.

In the context of specifications, partial functions will be used only for defining procedure operations. We will assume with no restriction that all observer operations are total functions, since it is easy to translate an observer definition to a new one that satisfies this constraint. This can be done by introducing new sorts for these observer operations, or by strengthening pre and post-conditions on every procedure that uses the observer. Since pre and post-conditions are functions returning a boolean value, they should be considered observer operations, and like all these operations they should be total functions.

In order to define bisimulation between instances of classes like AM we need to take into account such pre-conditions. First, they should behave like observer operations by returning the same values on any pair of related states; this is guaranteed from the property of observers if we only allow preconditions to access the state via such operations. Second, we should only consider applications of procedures on states, and for such arguments that their preconditions are satisfied. So suppose we have two objects A and C that instantiates class expressions like AM. Then, the new definition of bisimulation is:

```
  object
    P.., A: AM(P), C: AM(P)
  value
    bisimulation: (A.State × C.State)-set → Bool
    bisimulation(rel) ≡
      (∀ as : A.State, cs : C.State • (as,cs) ∈ rel ⇒
        (∀ t : P.T •
          A.attb(t,as) = C.attb(t,cs) ∧
          A.pre_proc(t,as) = C.pre_proc(t,cs) ∧
          A.pre_other(t,as) = C.pre_other(t,cs) ∧
          A.pre_proc(t,as) ⇒ (A.proc(t,as),C.proc(t,cs)) ∈ rel ∧
          A.pre_other(t,as) ⇒ let (as′,ta)=A.other(t,as), (cs′,tc)=C.other(t,cs)
                              in ta = tc ∧ (as′,cs′) ∈ rel end ))
```

```
full : State → Bool
full (n,l) ≡ len l = P.messages,
send : P.Message × State ~→ State
send(m, (n,l)) ≡ (n, l ⌢ ⟨m⟩)
    pre ∼full(n,l)
```

Fig. 5. Precondition for a partial function in the ring example.

```
object
  P:PARAMETERS,
  A:RING(P),
  C:RING1(P)
value
  bisimulation: Relation → Bool
  bisimulation(rel) ≡
    (∀ as : A.State, cs : C.State •
      (as, cs) ∈ rel ⇒
        (A.current(as) = C.current(cs)) ∧
        (A.empty(as) = C.empty(cs)) ∧
        (A.message(as) = C.message(cs)) ∧
        (A.full(as) = C.full(cs)) ∧
        (∀ m:P.Message •
          (∼A.full(as) ⇒ (A.send(m,as),C.send(m,cs)) ∈ rel ∧
          (A.receive(as),C.receive(cs)) ∈ rel)))
```

Fig. 6. Bisimulation in the ring example with partial functions.

Consider, for example, that in the specification of the ring in figure 3 we add a new value *messages* to determine the maximum number of messages that the ring can hold, and a new observer $full$ to evaluate whether the number of messages in the ring is equal to this parameter. In this case we should define function *send* as a partial function using the precondition shown in figure 5.

The complete definition of bisimulation between an abstract and a concrete specification of this module is shown in Figure 6.

The definitions and the examples presented so far showed how to compare classes based on their external operations. This approach is very restrictive for CBD, where components usually interact among them. In most cases, these interactions do not require the participation of their environment and are only represented by internal operations. Now we will adapt the bisimulation definition in order to handle this kind of dependencies.

Consider class expressions LS and RS as shown in Figure 7. We have the usual functions *attb*, *proc* and *other*, together with their preconditions. For simplicity, we only introduce one internal operation *int* into RS. This means that it implements a more concrete class which is constructed from several interacting components. Operation *int* is a partial function without arguments, only operating on the state, with pre-condition *pre_int*.

Let L and R be instances of classes LS and RS respectively. In order to define the corresponding bisimulation, we first introduce a function *int_star* as follows

```
scheme LS(P..) = class
  type
    T = P.T,
    State = ..
  value
    init: State,
    attb: T × State → T,
    proc: T × State ~→ State,
    other: T × State ~→ State × T,
    pre_proc,pre_other: T × State → Bool,
  axiom
    ..
end
```

```
scheme RS(P..) = class
  type
    T = P.T,
    State = ..
  value
    init: State,
    attb: T × State → T,
    proc: T × State ~→ State,
    other: T × State ~→ State × T,
    pre_proc,pre_other: T × State → Bool,
    int: State ~→ State,
    pre_int: State → Bool
  axiom
    ..
end
```

Fig. 7. Schematic class definitions with internal operations.

```
int_star: R.State × R.State → Bool
int_star(s1,s2) ≡
  s1 = s2 ∨ pre_int(s1) ∧ int(s1)=s2 ∨
  (∃ s3 : R.State • int_star(s1,s3) ∧ int_star(s3,s2))
```

This function is the reflexive, transitive closure of *int*, describing all pairs of states that can be reached by a sequence of its invocations.

As before, observations returned by observer operations must be equal on the pairs of related states. In this case, pre-conditions may not be attribute functions, but the become valid by the application of internal functions. Given pair of states of L and R, if the pre-condition of *proc* is true in L then we require that there exists a sequence of applications of *int* such that the pre-condition is also true in R and the application of f results in a state which is again in the relation. Similar for g. We call this relation simulation, because it considers internal functions in only one class.

```
simulation: (L.State × R.State)-set → Bool
simulation(rel) ≡
  (∀ l : L.State, r : R.State, t:T • (l,r) ∈ rel ⇒
    L.attb(t,l) = R.attb(t,r) ∧
    L.pre_proc(t,l) ⇒
      (∃ r′ : R.State • int_star(r,r′) ∧ R.pre_proc(r′) ∧
            (L.proc(t,l),R.proc(t,r)) ∈ rel) ∧
    L.pre_other(t,l) ⇒
      (∃ r′ : R.State • int_star(r,r′) ∧ R.pre_other(r′)
        let (l′,rl)=L.other(t,l), (r′′,rr)=R.other(t,r′) in
          rl=rr ∧ (l′,r′′) ∈ rel end))
```

Consider a class *BUFFER*, that allows to input and output messages, similarly as the ring. *BUFFER* also uses the module *PARAMETERS*. Consider also the class *COMP_BUFFER* which has two components *B1* and *B2*, that represent individual buffers. These components have as parameters module *PARAMETERS* and modules *L1* and *L2* respectively, which are used to define the bound of each buffer. *COMP_BUFFER* has functions *input* and *output* and the internal function *move* which moves a message from *B1* into *B2*. Definitions of *input* and *output* invoke directly the

```
scheme BUFFER(..) = class
  type
    State = P.Message*
  value
    isempty : State → Bool
    isempty(s) ≡ s = ⟨⟩,
    isfull : State → Bool
    isfull (s) ≡ len s = P.messages,
    input: P.Message × State ~→ State
    input(m, s) ≡
      s ⌢ ⟨m⟩
      pre ∼isfull(s),
    output: State ~→ State × P.Message
    output(s) ≡
      (tl s, hd s)
      pre ∼isempty(s)
  ...
end
```

```
scheme COMP_BUFFER(..) = class
  object
    L1, L2, P
    B1 : BUF(P, L1),
    B2 : BUF(P, L2)
  type
    State = B1.State × B2.State
  value
    input : M.Message × State ~→ State
    input(m,(s1,s2)) ≡
      (s1,B2.input(m,s2))
      pre ∼B2.isfull(s2),
    output : State ~→ State × M.Message
    output(s1,s2) ≡
      let (s1′,m)=B1.output(s1) in ((s1′,s2),m) end
      pre ∼B1.isempty(s1),
    move : State ~→ State
    move(s1,s2) ≡ (s1 ⌢ ⟨hd s2⟩,tl s2)
      pre ∼B1.isfull(s1) ∧ ∼B2.isempty(s2)
  ...
end
```

Fig. 8. Specifications of the $BUFFER$ classes.

corresponding operations on *B1* and *B2*, under the pre-conditions which are no longer observers on this level. The specifications in RSL are shown in Figure 8.

In order to define a bisimulation, we relate the states of the instances B and CB of these two classes, by concatenating the states of the components *B1* and *B2*, and comparing the result with the state of B.

```
value
 simulation: Relation → Bool
 simulation(rel) ≡
   (∀ b:B.State, (b1,b2):CB.State • (b,(b1,b2)) ∈ rel ⇒
     (B.isfull(b) ≡ CB.isfull(b1,b2)) ∧
     (B.isempty(b) ≡ CB.isempty(b1,b2)) ∧
     ∼B.isfull(b)  ⇒
       (∃ (b1′,b2′):R.State •
         move_star((b1,b2),(b1′,b2′)) ∧ ∼CB.B2.isfull(b2′) ∧
         (∀ m:M.Message • (B.input(m,b),CB.input(m,(b1′,b2′))) ∈ rel)) ∧
     ∼B.isempty(b) ⇒
       (∃ (b1′,b2′):R.State •
         move_star((b1,b2),(b1′,b2′)) ∧ ∼CB.B1.isempty(b1′) ∧
         let (b′,mb)=B.output(b), (cb′′,mcb)=CB.output(b1′,b2′)
         in mb = mcb ∧ (b′,cb′′) ∈ rel end))
```

This relation is a simulation between B and CB, meaning that it is possible to replace B with CB preserving the behavior.

6 Conclusions and Future Work

Bisimulation has been a central concept in concurrency theory[Park, 81,Milner, 1980]. In this paper we apply it to give a precise definition of equivalent behaviors between

classes representing specification of components. Classes exhibiting equivalent behaviors are interchangeable with respect to a compound system. This allows us to replace abstract classes representing specification with concrete implemented versions, with a formal proof of its correctness. The use of RAISE automatic tools helps in the formal verification process.

Recent developments in the area of theoretical computer science promotes the emergence of the related fields of coalgebras [Rutten, 1995,Jacobs, 1999] and hidden algebras [Goguen, 1999]. Although these concepts have been studied for several decades, the recognition that they constitute the underlying structure for dynamic systems is recent. These formalisms may be used for the same kind of problems we analyzed, but they are abstract and do not present a concrete supporting implementation. The main difference with our approach is that we customize the concept of bisimulation using a formal specification language that allows proof generations.

We plan to continue our research on several directions. One extension is to evaluate bisimulation taking into account possible deadlocks originated by internal functions; this point is not considered in the presented definition. Another extension is to apply signature refinement together with bisimulation in order to be able to be able to replace components even when the signatures of the operations are not exactly equal. In this case, it is possible to use as a basis the research in signature matching [Zaremski and Wing, 1997]. It is also possible to complement this work with concepts of refinement using an algebraic approach [Astesiano and Kreowski, 1999], in order to build components that will be later integrated.

References

[Astesiano and Kreowski, 1999] Astesiano, E. and Kreowski, H.-J. (1999). *Algebraic Foundations of System Specification*. Springer.

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

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

[Davis, 1993] Davis, A. M. (1993). *Software Requirements, Objects, Functions & States*. Prentice Hall International.

[Goguen, 1999] Goguen, J. (1999). Hidden algebra for software engineering. *Combinatorics, Computation and Logic*, 21:35–59.

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

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

[Hennessy and Milner, 1985] Hennessy, M. and Milner, R. (1985). Algebraic laws for nondeterminism and concurrency. *Journal of the ACM*, 32(1):137–161.

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

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

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

[Marco, 1979] Marco, T. D. (1979). *Structured Analysis and System Specification*. Yourdon Inc.

[Meyer, 1989] Meyer (1989). Artículo que está en la computer de componentes. *Journal of ACM*, 36(4):887–911.

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

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

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

[Rutten, 1995] Rutten, J. (1995). A calculus of transition systems (towards universal coalgebra). In *Modal Logic and Process Algebra. A Bisimulation Perspective*, pages 231–256. CSLI, Center for the Study of Lanaguage and Information.

[Szyperski, 1997] Szyperski, C. (1997). *Component Software Beyond Object-Oriented Programming*. Addison Wesley.

[Yourdon, 1989] Yourdon, E. (1989). *Modern Structured Analysis*. Yourdon Inc.

[Zaremski and Wing, 1997] Zaremski, A. and Wing, J. (1997). Specification matching of software components. *ACM Transactions on Software Engineering and Methodology*, 6(4).