Universidade Federal de Pernambuco

Centro de Informática

Safe and Constructive Design with UML Components : Extended Report.

Authors
Flávia Falcão
Lucas Lima
Augusto Sampaio

Abstract

Component based software development (CBSD) is an established paradigm to build systems from reusable and loosely coupled units. However, it is still a challenge to ensure, in a scalable way, that desired properties hold for component integration. We present a component based model for UML, including a metamodel, well-formedness conditions and a formal semantics via translation into BRIC. We use (our previous work on) BRIC as an underlying (and totally hidden) component development framework so that our approach benefits from all the formal infrastructure developed for BRIC using CSP. Component composition, specified via UML structural diagrams, ensures, by construction, adherence to classical concurrent properties: our focus is on the preservation of deadlock freedom. Partial automated support is developed as a plug-in to the Astah modelling tool. We illustrate two case studies of our approach.

Contents

List of Figures

Chapter 1

Introduction

Component based software development (CBSD) is a widely disseminated paradigm to build software systems by integrating independent and potentially reusable units called components. One of the motivations for this paradigm is replacing conventional programming with the systematic composition and configuration of components [1].

In some contexts, particularly when there is some criticality involved, a reliable architecture becomes a demand. This is expected to be designed with the goal of verifying the integration of its components in a rigorous and scalable way. However, *a posteriori* verification, can be costly, and is often infeasible.

There are several approaches to CBSD in the literature. For example, in Reo [2], a concurrent system consists of a set of components which are glued together by a circuit that enables flow of data between components. Components can perform I/O operations on the boundary nodes of the circuit to which they are connected. There are formal semantics for Reo, based on coalgebras [3] and automata. Another example is [4], which presents component-based refinement that focuses on the separation of interface and functional contracts, supporting different levels of abstraction. The approach in [5] is based on a semantic model encompassing composition of heterogeneous components; the behaviour of a component is described as an automaton or Petri net extended by data and functions given in C++. In [6], the authors introduce a framework for assessing component properties, like completeness and consistency of requirement specifications, using Z [7] and State-charts [8], and an approach to verifying reliability using stochastic modelling formalisms.

In previous work we have proposed a formal component model, together with a rule-based composition strategy, called BRIC [1], [9]. BRIC has the process algebra CSP [10] as an underlying semantic model. Given that the argument components are deadlock free, each composition rule ensures that he resulting (composed) component preserves deadlock freedom. By using some metadata and communication patterns, it has been shown that the formal and mechanised verification of component integration, using the FDR tool [11], can scale. In spite of the promising results, in order to use BRIC a developer needs to have a considerable knowledge of CSP and of model checking techniques.

Our aim here is to foster a formal CBSD model for UML [12], motivated by the fact that UML is a widely used notation in industry, and amenable to mechanized analysis. We benefit from the overall formal infrastructure built around BRIC, but this is totally hidden from the developer.

While UML is well-suited for modelling software systems in general, it lacks support for modelling components in the sense of a CBSD approach. The usual design notation to represent a component is a *subsystem*. This is a *package* stereotype with an explicit interface and a set of encapsulated elements (including classes, interfaces and other subsystems). Nevertheless, an appropriate component notion must also include a dynamic behaviour (that can be defined by a state machine) and, considering components as independent units, ports for message passing communication should also be a component design feature. The syntactic (metamodel) definition of a component notion in UML is the first contribution of this work.

In general, UML design elements and diagrams can be used in a very flexible way. However, to tailor the design to a CBSD approach, besides defining a component metamodel, we need additional (context sensitive) conditions to ensure the well-formedness of component systems. Particularly, we define how components can be composed to give rise to more elaborate components. This is our second contribution.

Finally, as a third contribution, we define a formal semantics for the proposed component model by translation into BRIC. Components, instances and connections are translated into CSP, and deadlock freedom verifications are conducted in FDR, using the BRIC composition rules. If the verification fails, the problem is traced back to the UML component level, and the problematic composition is exhibited to the developer. Partial automated support is developed as a plug-in to the Astah modelling tool [13].

Chapter 2

The BRIC Component Model

BRIC formalises concepts of interfaces, dynamic behaviour, component contracts, and communication protocols with focus on the interaction points of black box components and their runtime behaviour. CSP, as the underlying formal notation, allows modelling system components in terms of synchronous processes that interact through message-passing communication. Process algebraic operators allow specifying elaborate concurrency and distributed process networks. CSP offers rich semantic models that support a wide range of process verifications, and comparisons.

A component contract is defined in terms of a component behaviour (CSP process), its ports (CSP channels) and their respective types (interfaces). (Component Contract) A component contract *Ctr* comprises an observational behaviour B, a set of communication channels C, a set of interfaces I, and a total function $R : C \to I$ between channels and interfaces:

$$Ctr : \langle B, R, I, C \rangle$$

Such that, B is an I/O process, that is: if an event $c.x$ is in the alphabet of B (αB), c is either an input or an output channel of B; B has infinite traces (but finite states); and B is divergence free, input deterministic and strong output decisive. Also, $dom\, R = C$ and $ran\, R = I$, and, finally, let $c \in C$, $outputs(c, B) = \{out.x : R(c) \bullet c.out.x\}$ and $inputs(c, B) = \{in.x : R(c) \bullet c.in.x\}$.

In the above definition, being input deterministic means that if a set of input events in B is offered by the environment, none of them are refused by B. A process is output decisive when all choices (if any) among output events on a given channel in B are internal. The process, however, must offer at least one output on that channel. The notations *dom* and *ran* stand for the domain and range of a relation. The functions $output(c, B)$ and $input(c, B)$ yield all output and input events on channel c in process B, respectively.

In the Dining Philosophers problem, one can model a philosopher and a fork as components. As an example, the behaviour of a fork is described in UML as a state machine *stm_fork*, Figure 3.3. Initially, the fork is available for both philosophers (*available* state); however, two philosophers cannot hold a same fork simultaneously. This is represented in

the state machine by two states, $busy1$ and $busy2$, capturing the interactions with the two philosophers that share the fork.

This is given a semantics in CSP for the purpose of formal verification, as explained in detail in Section 3. The resulting process *Fork* is parametrised by its id, so that several instances for distinct identifiers can be created. It is defined as process *stm_fork* that captures the behaviour the UML state machine.

$$Fork(id) = stm\_fork(id)$$
$$stm\_fork(id) = available(id)$$

The behaviour of the state machine itself is that of its initial state, which, in this case, is captured by the process *available* that offers two alternative behaviours.

$$\begin{aligned} available(id) \quad = \quad & (port\_fork\_right.id.picksup\_I \rightarrow \\ & port\_fork\_right.id.picksup\_O \rightarrow busy1(id)) \\ \Box \\ & (port\_fork\_left.id.picksup\_I \rightarrow \\ & port\_fork\_left.id.picksup\_O \rightarrow busy2(id)) \end{aligned}$$

This choice is denoted an external choice in CSP ($\Box$). It allows the environment to choose between two processes by communicating an initial event, which resolves the choice. If the first choice of available is taken, the philosopher on the right holds the fork, and similarly for the one on the left. Each of these choices is defined as a sequence of events defined using the CSP prefix ($\rightarrow$) operator. For instance, in the first case, the process performances the event $port\_fork\_right.id.picksup\_I$, and then the event $port\_fork\_right.id.picksup\_O$ which the former represents the intention to pick the fork, and the latter indicates that it has been performed; finally, it behaves as the process $busy1$. The processes below complete the definition of $FORK$.

$$\begin{aligned} busy1(id) \quad = \quad & port\_fork\_right.id.putsdown\_I \rightarrow \\ & port\_fork\_right.id.putsdown\_O \rightarrow available(id) \\ busy2(id) \quad = \quad & port\_fork\_left.id.putsdown\_I \rightarrow \\ & port\_fork\_left.id.putsdown\_O \rightarrow available(id) \end{aligned}$$

The process $busy1$ engages in two events in sequence, capturing the release of a $FORK$, and then behaving again as available. The event $putsdown\_I$ indicates an input operation. Similarly, $putdown\_O$ is used as an output operation. The process $busy2$ is analogous, dealing with the second choice. The process $FORK(id)$ is an example of an I/O process that, as explained previously, has infinite traces, is divergence free, input deterministic and strong output decisive.

Apart from the notation for prefix and external choice, as illustrated in the previous example, CSP offers some basic processes and a rich repertoire of process operators. The processes *SKIP* and *STOP* represent successful termination and deadlock, respectively. The sharing parallel composition ($P1 \underset{cs}{||} P2$) synchronises P1 and P2 on the events in the synchronisation set cs; events that are not in cs occur independently. A particular case

is when the processes are composed in interleaving, denoted $P1 \,|||\, P2$, in which case $P1$ and $P2$ run independently. For a more detailed introduction to CSP see, for example, [10].

BRIC provides four composition rules: interleave, communication, feedback and reflexive compositions. Each of these compositions constructs a new component, which includes the original ones. In other words, each composition results in a unique, and elaborate, component. The structures of three of these composition rules are illustrated in

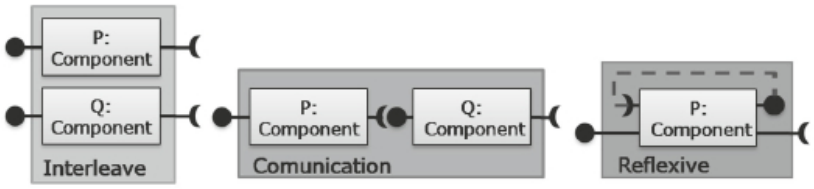

Figure 2.1: Compositions rules

2.1. The interleave and communication composition rules are binary compositions (they assemble channels of two distinct components). The feedback and reflexive composition rules are unary compositions (they assemble distinct channels of a component). These unary compositions rules have the same structure; they are distinguished by the conditions they impose.

Chapter 3

Component model, design and verification in UML

Although BRIC provides a sound and systematic development strategy, it is not appealing for practical use, as it requires deep knowledge of CSP. This was the main motivation for our UML based approach. First, in Section 3.1, we define a component model in UML, including the relevant well-formedness conditions. Then, in Section 3.1.1, we present the approach to create and compose component instances. Finally, in Section 3.1.2, we define a formal semantics for the proposed component model and composition by translation into BRIC.

3.1 Component metamodel and well-formedness

In Figure 3.1 we define a metamodel that formally captures the structure of the component model we propose. This metamodel extends constructs from a subset of UML that are identified as grey filled boxes. The unfilled boxes are the new elements introduced; these are defined as stereotypes of standard UML design elements. Next, we explain each element of our component metamodel.

A component is a UML *Subsystem*. A component must be either a *BasicComponent* or a *HierarchicalComponent*. A *BasicComponent* is not defined in terms of other components. It has one *BasicComponentClass* that describes the behaviour of the component and its ports. A *BasicComponentClass* is a UML *EncapsulatedClassifier* element, which, apart from attribute and methods, includes ports. This is the core class of a component metamodel. Its behaviour is defined by a state machine that should be referenced in *STM_REF*; this represents a reference to a State Machine that defines the behaviour of the component. In the model, it is represented by a UML *Comment*, also known as a *Note*. The content of this comment must be the State Machine name. The ports can be defined either in a class diagram or in a composite structure diagram. For the latter case, the *BasicComponentClass* must be linked to a *Structure_REF* comment.

In the Dining Philosophers, *FORK* is an example of a *BasicComponent*. In Figure 3.2, it is defined as a *Subsystem* stereotyped *BasicComponent*. It has a *BasicComponentClass*

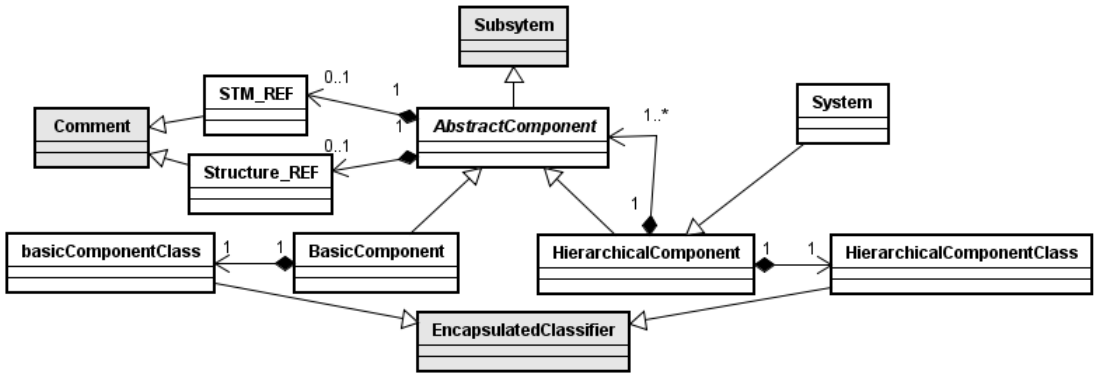

Figure 3.1: The component Metamodel.

with two ports, *right* and *left*, both realising the *interface_phil_fork* interface. Also, it has a comment stereotyped *STM_REF* with the name of the state machine of this component.

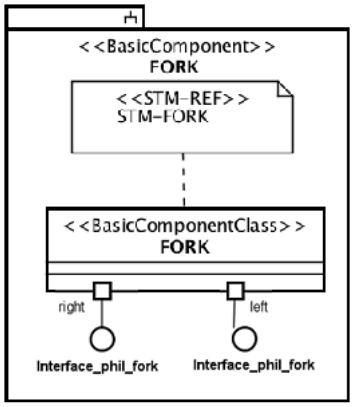

Figure 3.2: Class diagram of the Fork component

Figure 3.3 shows the state machine *STM_FORK*, which represents the reactive behaviour of the *FORK* component. It cyclically offers the possibility of picking up the fork through its left or right ports, and then waits for the fork to be put down via the same port.

In the same way we have the Figure 3.4 that represents the Philosopher component as a *BasicComponent*, called *PHIL*. And the Figure 3.5 shows the state machine *STM_PHIL*, which represents the behaviour of the *PHIL* component.

A *HierarchicalComponent* is defined by the composition of other components. It can have a state machine to define its behaviour, which must have its name specified in a *STM_REF* comment. This component must have a *HierarchicalComponentClass*, which owns a collection of other component classes. The connections between them should be

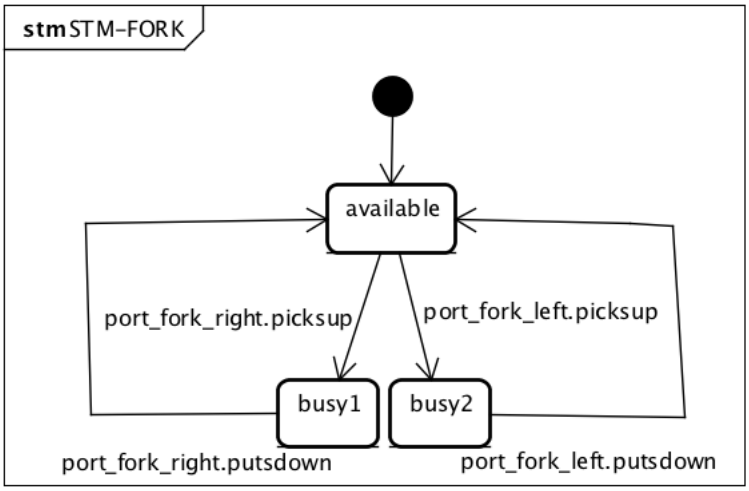

Figure 3.3: State Machine Diagram for the Fork Component

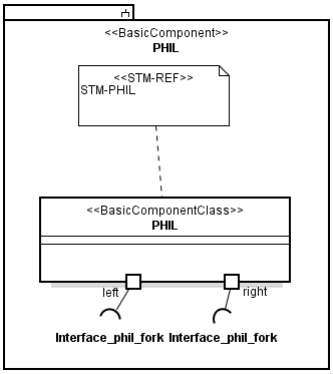

Figure 3.4: Class diagram of the Philosopher component

expressed in the Composite Structure Diagram referred by the *Structure_REF* comment. A *HierarchicalComponentClass* is a UML *EncapsulatedClassifier* element, hence, it may have ports to interact with other components. Similar to a *STM REF*, *Structure_REF* is represented by a UML comment element. The content of this comment is the name of the Composite Structure Diagram that models the structure of the component. Finally, a *System* is a specialisation of a *HierarchicalComponent* and it can be seen as the root component from where the entire system is specified.

The Dining Philosophers problem is modelled as a *System* element and, therefore, as a *HierarchicalComponent*; see Figure 3.6. It has a *HierarchicalComponentClass* that is related to one or more *FORK* and one or more *PHIL* components, using a composition relationship. Also, it has a linked comment specifying the composite structure diagram (*STR-DINING-PHIL*) that details how the parts are connected. The approach to compose component instances is described in the next section.

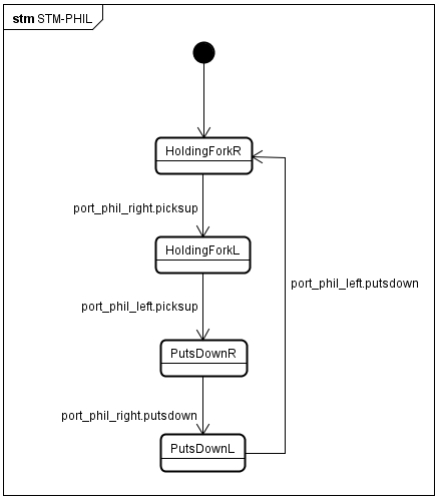

Figure 3.5: State Machine Diagram for the Philosopher Component

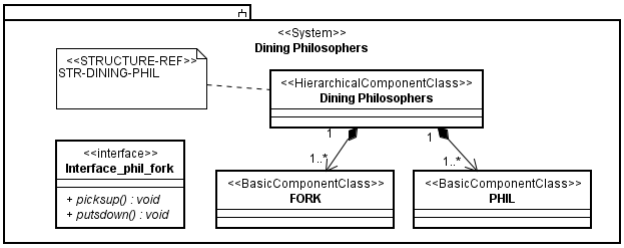

Figure 3.6: Class diagram of the Dining Philosophers.

In addition to the metamodel, we need to define some well-formedness conditions to characterise meaninful models that can be assigned a formal semantics. Furthermore, a precise chacaterisation of a meaningful model can be seen as a modelling style to guide practitioners during the design of systems. The well-formedness conditions are as follows.
System element. There must be exactly one *System*, which is the root component. This is a special type of *HierarchicalComponent* that is specified by a class diagram and a composite structure diagram. The former must have a subsystem stereotyped *System* which has composition relationships between its *HierarchicalComponentClass* (root) and the other component classes. The latter describes the internal structure of its *HierarchicalComponentClass*, that is, how the component instances are connected.
Basic Component. This kind of component is specified by a subsystem stereotyped *BasicComponent* that has one class stereotyped *BasicComponentClass* whose behaviour must be described by a State Machine. The name of the *BasicComponentClass* must be the same as the one for the component. A *BasicComponent* may have an associated structure to describe the ports of the *BasicComponentClass*.
Hierarchical Component. This kind of component is specified by a subsystem stereo-

typed *HierarchicalComponent* that has one class stereotyped *HierarchicalComponentClass*. Similar to the *BasicComponentClass*, the name of the *HierarchicalComponentClass* should be the same as the one for the component. This class must be the head of a composition relationship with other component classes to express the ownership of other components. The *HierarchicalComponentClass* must have its structure described by a composite structure diagram where the connections between the owned component classes are specified.
Multiplicities. Multiplicities with the * character are not allowed in the composite structure diagram because we are dealing with instances. This is important to make the formal analysis feasible. Also, all parts in a composition relationship must appear in the associated composite structure diagram in numbers compatible with their multiplicities.
Binding Structure and Behaviour to Component. UML Comments (or Notes) are used to bind a state machine or a composite structure to a component. To bind a state machine to a component class the associated comment must be stereotyped *STM-REF* and the content of the note must be the name of the state machine diagram. Likewise, in order to bind a composite structure, the comment must be linked to the component class and be stereotyped *Structure_REF*. The content must be the name of the composite structure diagram.
Component Services. The contract of a component must be modelled using ports. Each component class must have ports exposing the required and provided services. Ports must realise Required and/or Provided Interfaces that describe the operations that a component needs or perform.
Port Multiplicity. In case there is a connector between two ports where at least one of them has multiplicity greater than one, the connector must be labelled to indicate the port being connected. The label must follow the pattern *"port1_name"*$<->$*"port2_name[i]"*, where *port1_name* is the name of a port that has multiplicity one, *port2_name[i]* is the name of a port that has multiplicity greater than one and i is the index of the port of the connection which ranges from one to the number of the multiplicity.

In the next section we detail how component instances can be composed at the UML level and the relationship with the BRIC forms of composition.

3.1.1 Composition of component instances

The proposed approach offers two types of component composition: *Interleave* and *Communication*. In the next subsection we map these forms of composition to the four kinds provided by BRIC. We anticipate, however, that, in a UML-based design, where the formalities are hidden, it is unnecessary (and convenient not) to distinguish between communication, feedback and reflexive compositions; these were explained in Section **??**.

Composition of component instances is described using a *Hierarchical Component* element. In this section we describe how to compose component instances based on the metamodel detailed in Section 3.1.

The simplest form of composition is *Interleave composition*. This is achieved by instantiating components in the composite structure diagram of a hierarchical component, see Figure 3.7. Each instance has a type: a component previously defined. For example, in Figure 3.8 we show two instances of *FORK* and two of *PHIL* in a hierarchical component.

Before introducing a connection between *phil*1 and *fork*1, the four instances were in interleaving, as the communication of events through the ports can happen without any interference from each other. Therefore, when component instances are created, they are, by default, in interleaving.

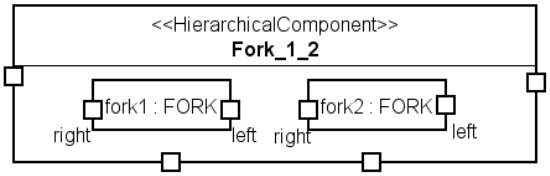

Figure 3.7: Interleave Composition UML

Communication composition is performed through the connection of ports from two different components. The same interface must be provided by one component and required by the other one. Figure 3.8 illustrates in (1) a communication between *fork*1 and a *phil*1.

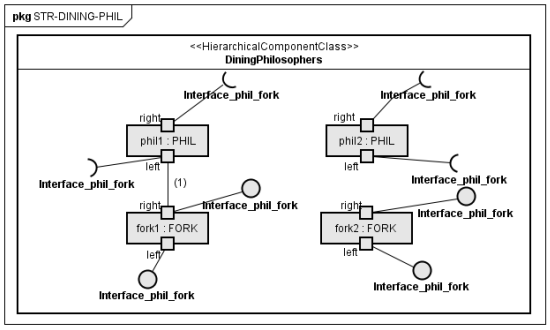

Figure 3.8: Inter-Component Composition UML.

Even when starting with deadlock free components, communication composition can lead to a deadlock if some conditions are not obeyed by the components being connected. It is necessary to verify protocol compatibility of the channels that are to be connected. Broadly, there must always be an output event to be performed, and at least one of the processes must have all enabled outputs accepted by the other process. In our approach, this is verified by translation into BRIC, and using the related verification techniques automated by the FDR tool, completely hidden from the user.

As an illustration, Figure 3.9 shows some additional communication compositions in the Dining Philosophers example. While the first three compositions preserved deadlock freedom, the fourth one, labelled (4), introduces a deadlock. This is evidenced by the red line connection, and is displayed to the user by the tool interface, as detailed in the next section.

The reason for this well-known deadlock is the symmetry of the design of the philosophers and forks: it allows all the philosophers (two in our example) to pick up, say, the

left fork, and then prevents any of the philosophers to pick up the right fork; as a result, the philosophers will starve.

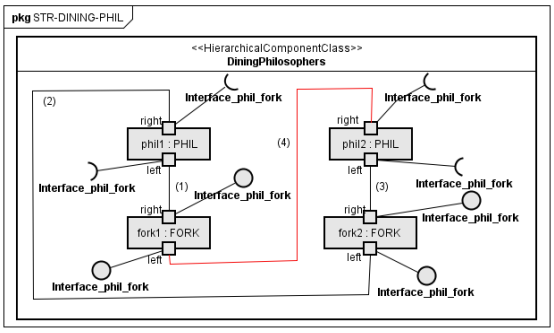

Figure 3.9: Component with deadlock.

A possible solution is to break the symmetry and design one of the philosophers to pick up the forks in a different order than the other ones. This fix makes the system deadlock free and the red line is turned into black.

An interesting feature of this step by step composition is that, when a deadlock is found, the developer is warned of the particular connection that is causing the problem. Also, with the separation of concerns we have adopted in our approach, all the semantic details of the formal verifications are totally hidden from the user, who can concentrate on the more appealing graphical UML notation. As already mentioned, one of the distinguishing features of this work is to verify the properties in background while the UML model is being created to help the user construct a deadlock free model.

3.1.2 Semantics via translation into BRIC

In order to perform a mechanised compositional verification during the model construction, we translate, on demand, the UML models to BRIC, which itself uses CSP as the underlying formal notation. We then use tool support for CSP to automatically check properties and trace back any results to the UML level. In this section we give an intuition on how the UML models are represented as CSP processes according to the BRIC component model.

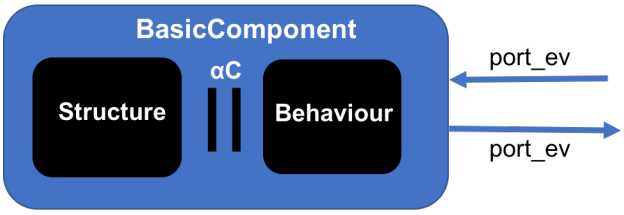

Figure 3.10: Illustration of a *BasicComponent* in CSP.

Figure 3.10 illustrates a *BasicComponent* in CSP. It is translated to a CSP process that composes in parallel two other processes, one for the structural part and another related to its behaviour. The former defines a memory for accessing the attributes of the component, which are defined in the UML *BasicComponentClass*. The latter results from the translation of the component State Machine. Both processes synchronise on the set of events αC, which has events for reading and setting the value of each attribute. UML operations, signals and ports are translated to CSP channels that communicate the related events. Components may expose their services to other components through ports. For instance, the incoming and outgoing arrows shown in Figure 3.10 communicate these events.

Considering again our running example, the Dining Philosophers, in Section **??** we presented the CSP process for the *FORK*, which is a *BasicComponent*. Only the translation of the State Machine depicted in Figure 3.3 was presented; its structural part is simply the *SKIP* process, as the component *Fork* has no attributes. The case study presented in the next section considers components with attributes.

The State Machine of the process *Fork* is translated to a CSP process where each state is a process, as shown in Section **??**. The main process is *stm_fork* from which we can reach the *available* process, and later, the *busy*1 or the *busy*2 process. Triggers between states are represented by channels; in this case, the trigger *port_fork_right* is a channel that represent the communication through the port named *right* from the *FORK* component. Events are communicated through this channel whose type is a pair: *id.operation*: id (the component identifier) and operation (*putsdown_I*, *putsdown_O*, *picksup_I*, *picksup_O*).

Events that represent operations are derived from the interfaces realised by the component. Each operation from the interface produces two datatypes, both named after the operation, but, the first, suffixed by $\_I$, indicates that this type encodes the operation call and the input parameters; the second, suffixed by $\_O$, indicates that this type encodes the reply to the call together with the output parameters. The *FORK* component provides one interface that has two operations: *picksup* and *putsdown*. Then they are translated to one CSP datatype:

$$datatype \quad operation = picksup\_I \mid picksup\_O \mid putsdown\_I \mid putsdown\_O$$

A *HierarchicalComponent* is specified by the parallelism of its internal components. For instance, consider a *HierarchicalComponent* that has two connected internal components, namely *C1* and *C2*. Wherever two components are connected in UML, such a connection is represented in CSP by the parallel composition of the component processes and a *Buffer* process that orchestrates the communication between the components. Communication in CSP is synchronous while message passing in UML is asynchronous: two events are used to represent the sending and the receiving of a message. Thus, the *Buffer* process simply defines the order in which the events happen through the ports of the components.

The synchronisation alphabet of a component process and the buffer is defined by the events *sent to* and *received from* the ports for that particular connection. For instance, if

component *C1* requires a service provided by *C2*, which is represented by the connection between their ports, then $\alpha C1$ has the events of the port of *C1* used in this connection, and $\alpha C2$ has the events of the port of *C2*. The *Buffer* process simply guarantees that the first event comes from the port of *C1* followed by the event related to the port of component *C2*. Finally, a *HierarchicalComponent* can also receive communication from external entities through its ports. These events can be relayed to one of its internal components.

As in the case of basic components, component compositions are also translated to CSP. For instance, the interleave composition of two forks involves no communication and, therefore, no intermediate buffer. This composition is translated to the following process:

$$FORK_{1\_2} = FORK(1) \;|||\; FORK(2)$$

This is checked for deadlock freedom using the BRIC rule for interleaving: it will be deadlock free if one of the components is deadlock free. In this case, both $FORK(1)$ and $FORK(2)$ are deadlock free.

When a connection between two components happens, at the UML level, this entails a communication composition. In BRIC, this can be mapped to communication, feedback or a reflexive composition. As already explained, communication composition in BRIC is used to connect channels of distinct components, whereas feedback and reflexive compositions link two channels of the same component.

For simplicity, we map the instances in a UML composite structure diagram to a single component, and then decide between applying feedback or reflexive composition for deadlock analysis. Then feedback composition is tried first, which requires that the channels being connected are decoupled, meaning that their connection does not establish a cyclic topology [9]. If it fails, then the reflexive composition rule can be applied, as, despite being more expensive, it can handle cyclic networks. The result is then traced back to the UML model.

Figure 3.8 shows a feedback composition of a communication between *phil*1 and *fork*1 in the process that interleaves two forks and two philosophers. The communication uses the port named *left* from *phil*1 with port *right* from *fork*1.

The Feedback composition represents the simple unary composition case, where two channels of the same component are assembled but do not introduce a new cycle [9]. The process *inter_fork_1_2_phil_1_2* contains all forks and philosophers with no communication. This whole process is now considered a new component. This process interleaves the process for the two forks and two philosophers. For simplicity, we omitted the definition of $FORK_{1\_2}\_PHIL1$, but it simply composes the processes $FORK_{1\_2}$ and $PHIL(1)$ in interleaving.

$$inter\_fork\_1\_2\_phil\_1\_2 = \quad (FORK_{1\_2}\_PHIL1 \;|||\; PHIL_2)$$

When we connect two ports of this component, it is considered a Feedback composition, which generates a new component whose process is *feed_inter_fork_1_2_phil_1_2*. This new component is the parallel composition between the process of forks and philosophers

(in interleaving) and the buffer, synchronizing on the channels related to the connected ports:

$$\begin{aligned} feed\_inter\_fork\_1\_2\_phil\_1\_2 = \quad & (inter\_fork\_1\_2\_phil\_1\_2) \\ & \underset{\{|port\_fork\_left.1, port\_phil\_right.1|\}}{||} \\ & BFIO(port\_fork\_left.1, \\ & port\_phil\_right.1) \end{aligned}$$

The subsequent connections, as presented in Figure 3.9, are all translated to the application of feedback composition, except for the last one, which creates a cycle in the process network; this is translated into a reflexive composition. All these compositions are checked using FDR and, for the symmetric version of philosophers and forks, a deadlock is identified in the final composition, as already explained in the previous subsection.

The full translation to CSP specification from UML component models and how the BRIC rules are applied can be found in Appendix A.

Chapter 4

Tool support and case study

To support the proposed CBSD approach, we envision the implementation of a tool with the following features: implementation of the component metamodel and the well-formdness conditions presented in the previous section; editing facilities for model elements and diagrams; creation of component instances and composition of instances by connecting their channels; translation of component models into BRIC; fction of the composition conditions in background (using the FDR tool); and traceability of the verification counterexamples back to the UML component model.

We are currently developing a *plug-in* in the Astah modelling environment [13] to support the above features. Astah has been chosen due to the following reasons: its extension capabilities facilitates the creation of plug-ins; models can be created using the several UML elements and diagrams, which allows us to reuse the notation we to define our component model, and extend our approach to other model elements in the future; and it has a large community of active users. Also, Astah *plug-ins* allow an easy integration with other tools. In our case, we need to integrate with FDR for the purpose of mechanised verification.

Creating models using Astah is considerably intuitive for UML practitioners. With the plug-in, while the user creates a model, this is incrementally translated into CSP, according to the BRIC metamodel; the BRIC composition rules are used to check deadlock freedom preservation using FDR in background. Given that a deadlock is identified, the the user is notified.

Currently, we have a simple prototype of the *plug-in*. Editing facilities are borrowed from Astah, but adherence to the metamodel and well-formedness presented in Section 3 is not yet enforced. However, assuming the developer constructs an adherent model, the prototype automatically generates the CSP from the state machines, run FDR in background, and presents a deadlock trace, when a problem is found.

Apart from the Dining Philosopher that we used as a running example, to validate our strategy we developed another case study: a Ring Buffer. It represents a reactive bounded buffer which is composed by a ring of storage cells with a controller and a cache. Each cell is able to store one value. The controller is responsible to intermediate the communication between the environment and the ring, receiving input requests and

sending values to be stored inside the cells.

The model of the Ring Buffer system in UML is shown in Figure 4.1. It is a *HierarchicalComponent* composed by at least one *Cell* and exactly one *Controller*. We omit the design of these basic components.

In order to allow communication among the controller component and the cells, a common interface is realized by them: *INTERFACE_CONTROL_CELL*. Similarly, *INTERFACE_ENV* is the interface of the *Controller* with the environment.

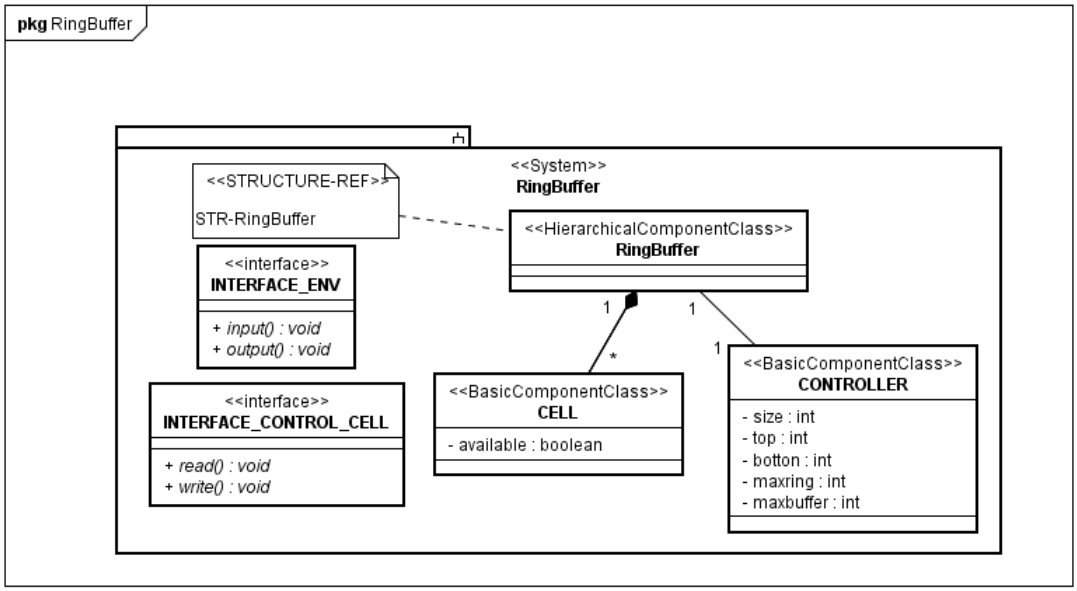

Figure 4.1: Ring Buffer System.

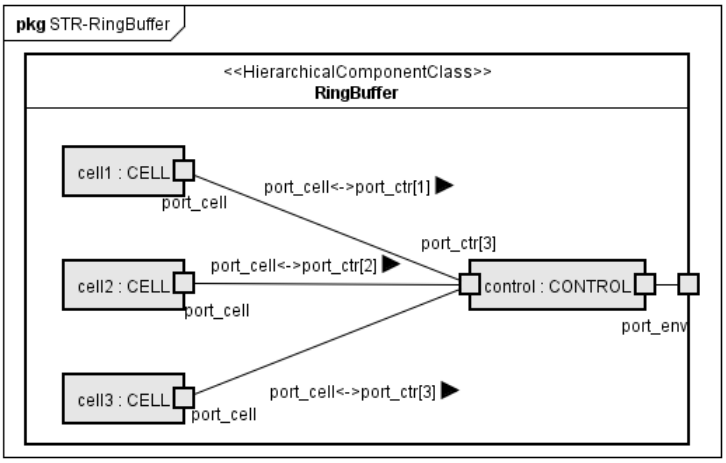

Figure 4.2: Ring Buffer Composition.

The connections are captured by the composite structure diagram *STR-RingBuffer*; see Figure 4.2. We consider a configuration with three cells. The *Controller* has one conjugate port (with arity three), represented by *port_ctr*[3], which is indexed, from 1 to 3, to establish the connections with the three cells. To define which index of *port_ctr* is connected with a port from a *Cell* component, a label is used, for instance: *port_cell* `<->` *port_ctr*[1]. The values to be stored to (and recovered from) the Ring Buffer are communicated by *port_env* that interfaces with the environment and realises *INTERFACE_ENV*.

As detailed for the Dining Philosophers, after each connection is performed, BRIC rules are applied to the automatically genarated CSP model, and verified using the FDR tool. In the table

Chapter 5

Conclusion

We propose a UML component model, with associated well-formedness conditions, that supports an incremental design and ensures the presertavion of desired properties; we have focused on deadlock freedom. We also define a formal semantics for the proposed component model by translation into BRIC. Components, instances and connections are translated into CSP processes, and deadlock verifications are conducted using the FDR tool, using the BRIC composition rules.

We have also implemented part of the approach as a prototype in the form of a *plug-in* to the Astah modeling tool. Using the prototype, the CSP notation and the formal verification is hidden from the user. If the verification fails, the problem is traced back to the UML component level, and the problematic composition is exhibited to the developer who does not need to have CSP knowledge.

To illustrate the overall approach, we developed two case studies (the classical Dining Philosophers and a Ring Buffer) that exemplify the modeling of basic and hierarchical components, with associated state machine and composite structure diagrams with the connection of component instances. We have also described how a UML model, adherent to the proposed metamodel and the well-formedness conditions, is translated to CSP and and the BRIC composition rules.

There are several approaches to defining component models and verification strategies, based on a variaty of formalisms. For instance, in Reo [2], a concurrent system consists of a set of components which are glued together by a circuit that enables flow of data between components. Its formal semantics are based on coalgebras and automata. Another example is rCOS [4], which has a formal semantics based on an extension of the Unifying Theories of Programming (UTP) and automatically generates CSP processes to verify the compatibility between sequence diagrams and the state machine diagram of a contract. However, to our knowledge, there is no work that provides a UML component metamodel and well-formedness conditions that constructively ensures the preservation of deadlock freedom, including a traceability between the underlying formal analysis and the UML model.

We use as a basis for our translation from UML to CSP the work presented in [14], which presents a formal semantics for a comprehensive subset of SysML [15] via a

mapping into CML [16], a formalism that combines CSP and VDM [17]. The work proposes guidelines that assign some design roles to be played by each of the considered elements in an integrated model. It focuses on state machine, activity, sequence, block definition (class) and internal block (composite structure) diagrams. However, the purpose of [14] is not on component-based design nor on ensuring property preservation by construction.

Despite the promising results and the emphasised contributions, our approach has some limitations. The Dining Philosophers and the Ring Buffer models, while suitable to illustrate a compositional approach, are not realistic examples in the context of CBSD. We intend to explore more elaborate industrial examples. Concerning automation, the prototype that was developed needs to be significantly improved to support all the features listed in Section 4. Particularly, we need to implement adherence to the metamodel and the related well-formedness conditions. As another future direction we plan to adapt the approach proposed in [18] for the construction of heterogeneous collections of components that are defined as patterns using generic (rather than concrete) instances. This allows to parametrise a composite structure diagram by the number of instances involved in a system configuration, rather than being forced to statically determining a particular configuration.

Bibliography

[1] M. V. M. Oliveira, P. Antonino, R. Ramos, A. Sampaio, A. Mota, and A. W. Roscoe, "Rigorous development of component-based systems using component metadata and patterns," *Formal Aspects of Computing*, vol. 28, no. 6, Nov. 2016, ISSN: 1433-299X.

[2] F. Arbab, "Reo: A channel-based coordination model for component composition," *Mathematical. Structures in Comp. Sci.*, vol. 14, no. 3, pp. 329–366, Jun. 2004, ISSN: 0960-1295.

[3] B. Jacobs and J. Rutten, "An introduction to (co) algebra and (co) induction," *EATCS Bulletin 62*, 1997.

[4] Z. Chen, Z. Liu, A. P. Ravn, V. Stolz, and N. Zhan, "Refinement and verification in component-based model-driven design," *Sci. Comput. Program.*, vol. 74, no. 4, Feb. 2009, ISSN: 0167-6423.

[5] B. Bonakdarpour, M. Bozga, M. Jaber, J. Quilbeuf, and J. Sifakis, "A framework for automated distributed implementation of component-based models," *Distributed Computing*, vol. 25, no. 5, Oct. 2012, ISSN: 1432-0452.

[6] H. Y. Kim, K. Jerath, and F. Sheldon, "Assessment of high integrity software components for completeness, consistency, fault-tolerance, and reliability," in *Component-Based Software Quality: Methods and Techniques*, A. Cechich, M. Piattini, and A. Vallecillo, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, ISBN: 978-3-540-45064-1.

[7] J. Woodcock and J. Davies, *Using z: Specification, refinement, and proof*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1996, ISBN: 0-13-948472-8.

[8] D. Harel, "Statecharts: A visual formalism for complex systems," *Sci. Comput. Program.*, vol. 8, no. 3, pp. 231–274, Jun. 1987, ISSN: 0167-6423.

[9] R. Ramos, A. Sampaio, and A. Mota, "Systematic development of trustworthy component systems," in *FM 2009: Formal Methods*, D. R. Cavalcanti Anaand Dams, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, ISBN: 978-3-642-05089-3.

[10] A. W. Roscoe, *The theory and practice of concurrency*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1997, ISBN: 0136744095.

[11] T. Gibson-Robinson, P. Armstrong, A. Boulgakov, and A. Roscoe, "Fdr3: A modern refinement checker for csp," English, in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, E. Abraham and K. Havelund, Eds., vol. 8413, Springer Berlin Heidelberg, 2014, ISBN: 978-3-642-54861-1.

[12] Object Management Group (OMG), *Meta-Object Facility (MOF) Specification, Version 2.5.1*, OMG Document Number formal/2016-11-01 (`http://www.omg.org/spec/MOF/2.5.1`), 2016.

[13] "Change vision, inc., . astah professional. last accessed 2018," in. [Online]. Available: `URL:%20http://astah.net/editions/professional`.

[14] L. Lima, A. Miyazawa, A. Cavalcanti, M. Cornélio, J. Iyoda, A. Sampaio, R. Hains, A. Larkham, and V. Lewis, "An integrated semantics for reasoning about sysml design models using refinement," *Softw. Syst. Model.*, vol. 16, no. 3, Jul. 2017, ISSN: 1619-1366.

[15] Object Management Group (OMG), *OMG System Modeling Language (OMG SysML), Version 1.5*, OMG Document Number formal/17-05-01 (`https://www.omg.org/spec/SysML/1.5/`), 2017.

[16] J. Woodcock, A. Cavalcanti, J. Fitzgerald, P. Larsen, A. Miyazawa, and S. Perry, "Features of cml: A formal modelling language for systems of systems," in *2012 7th International Conference on System of Systems Engineering (SoSE)*, Jul. 2012.

[17] J. Fitzgerald and P. G. Larsen, *Modelling systems: Practical tools and techniques in software development*. New York, NY, USA: Cambridge University Press, 2009, ISBN: 0521899117.

[18] A. Cavalcanti, A. Miyazawa, A. Sampaio, W. Li, P. Ribeiro, and J. Timmis, "Modelling and verification for swarm robotics," in *Integrated Formal Methods*, To appear., 2018.

[19] L. Bichler, A. Radermacher, and A. Schurr, "Evaluating uml extensions for modeling real-time systems," in *Proceedings of the Seventh IEEE International Workshop on Object-Oriented Real-Time Dependable Systems. (WORDS 2002)*, 2002.

[20] J. A. Street and R. G. Pettit, "The impact of uml 2.0 on existing uml 1.4 models," in *Proceedings of the 8th International Conference on Model Driven Engineering Languages and Systems*, ser. MoDELS'05, Montego Bay, Jamaica: Springer-Verlag, 2005, ISBN: 3-540-29010-9, 978-3-540-29010-0.

[21] L. Kharb, "Component-based development methodology: The concept and process," Jan. 2007.

[22] C. Szyperski, *Component software : Beyond object-oriented programming*. Jan. 2002, ISBN: 0-201-745572-0.

[23] J. Jürjens, E. B. Fernandez, R. B. France, B. Rumpe, and C. Heitmeyer, "Critical systems development using modeling languages (csduml'04): Current developments and future challenges (report on the third international workshop)," in *UML Modeling Languages and Applications*, N. Jardim Nunes, B. Selic, A. Rodrigues da Silva, and A. Toval Alvarez, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, ISBN: 978-3-540-31797-5.

[24] B. Scattergood, *The semantics and implementation of machine-readable csp*. 1998, ISBN: 0000 0001 3554 8226.

[25] D. Urting, Y. Berbers, S. Van Baelen, T. Holvoet, Y. Vandewoude, and P. Rigole, "A tool for component based design of embedded software," in *Proceedings of the Fortieth International Conference on Tools Pacific: Objects for Internet, Mobile and Embedded Applications*, ser. CRPIT '02, Sydney, Australia: Australian Computer Society, Inc., 2002, ISBN: 0-909925-88-7.

[26] B. Selic, "Using uml for modeling complex real-time systems," in *Languages, Compilers, and Tools for Embedded Systems*, F. Mueller and A. Bestavros, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, ISBN: 978-3-540-49673-1.

Appendix A

Complete UML component model - Dining Philosopher

The Dining philosophers is the most famous example of deadlocked system. It represents a round dining table where philosophers are going to eat. The philosophers are seated around the table and each pair of neighboring philosophers have a fork between them. In order to eat, a philosopher must take two forks. In an example where every philosopher takes the fork on his right side before the fork in his left side, there is a possibility of an eventual deadlock: when every philosopher has taken one fork and they are waiting for the other fork to be free.

A fork and a philosopher are describe as a component model in UML, *Fork* and *Phil* respectively. And in our example a Dining Philosopher Component is composed by two *FORK* and two *PHIL*.

A.1 Fork Basic Component

In this section is illustrated the UML model of *Fork* component. It is modeled as a *BasicComponent*.

A.1.1 Class Diagram

The Class Diagram in the Figure A.1 illustrates the *BasicComponent* called *FORK*. It is composed by a *STM_REF* which is a reference to a State Machine Diagram, *STM_FORK*; and a *BasicComponentClass*, *FORK*. In this view is possible to present the ports of a component and which interfaces it may to provide or require. The *Fork* component has two ports: *right* and *left* that provide the interface *interface_phil_fork*.

A.1.2 State Machine Diagram

The behaviour of a component is described by its state machine. In the Figure A.2, initially, the fork is available for both philosophers (*available* state); however, two philosophers

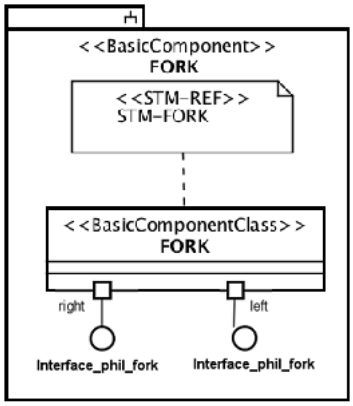

Figure A.1: Class diagram of the Fork component

cannot hold a same fork simultaneously. This is represented in the state machine by two states, *busy*1 and *busy*2, capturing the interactions with the two philosophers that share the fork.

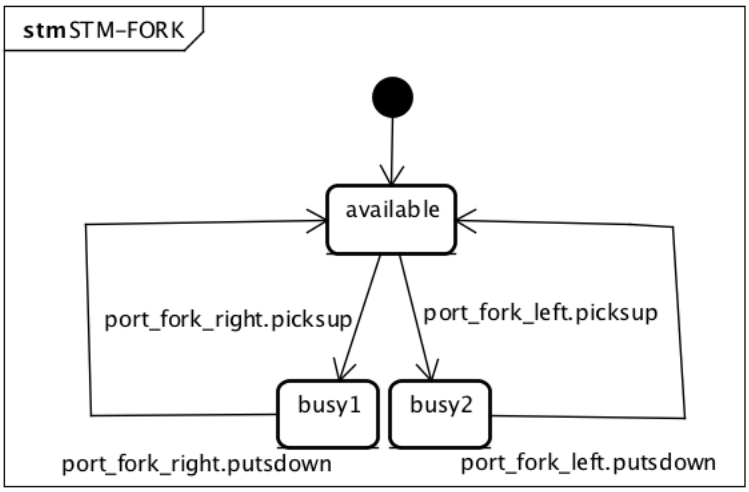

Figure A.2: State Machine Diagram for the Fork Component

A.2 Philosopher Basic Component

In this section is illustrated the UML model of *PHIL* component. It is modeled as a *BasicComponent*.

A.2.1 Class Diagram

The Class Diagram in the Figure A.3 illustrates the *BasicComponent* called *PHIL*. It is composed by a *STM_REF* which is a reference to a State Machine Diagram, *STM_PHIL*; and a *BasicComponentClass*, *PHIL*. The *BasicComponentClass* has two ports: *right* and *left* which require the interface, *interface_phil_fork*.

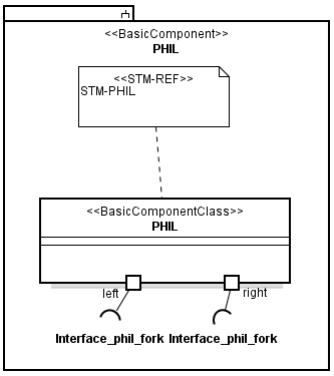

Figure A.3: Class diagram of the PHIL component

A.2.2 State Machine Diagram

The behaviour of a component is described by its state machine. One philosopher must pick up two forks: the right one (*HoldingForkR* state) and the left one(*HoldingForkL* state) before eating and put them down afterwards, these are represented by *PutsDownR* and *PutsDownL* states. After the philosopher is able to start a new cycle and picks up a fork. The Figure A.4.

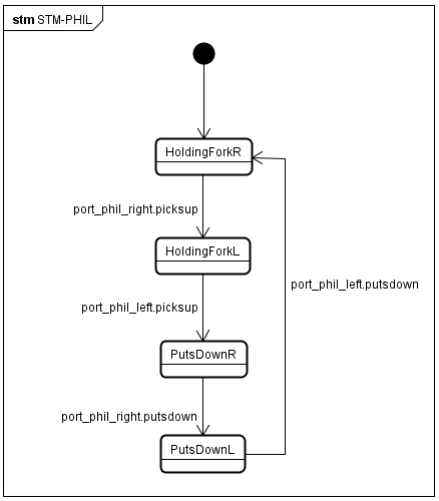

Figure A.4: State Machine Diagram for the PHIL Component

A.3 Dining Philosopher Hierarchical Component

A.3.1 Class Diagram

The Dining Philosophers problem is modelled as a *System* element and, therefore, as a *HierarchicalComponent*; see Figure A.5. It has a *HierarchicalComponentClass* that is related to one or more *FORK* and one or more *PHIL* components, using a composition relationship. It presents the interface that must be required or provided by components to allow the communication, which is called *interface_phil_fork*. This interface has two operations, *picksup()* to represent the act of picking a fork up, and *putdown()* to represent the act of putting a fork down.

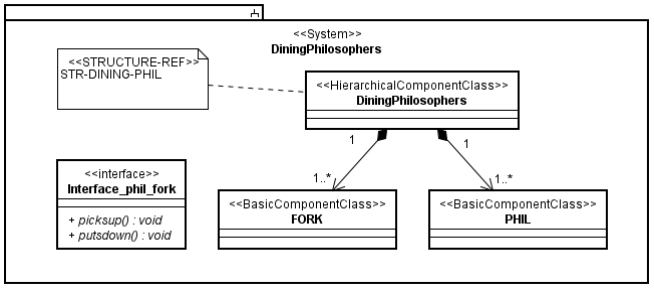

Figure A.5: Dining Philosopher Hierarchical Component Model

A.3.2 Composite Structured Diagram

Also, the *HierarchicalComponentClass* has a linked comment specifying the composite structure diagram (*STR-DINING-PHIL*) that details how the parts are connected, shown in Figure A.6. This represents a deadlock free composition, the solution involves replacing one of the above right-handed philosophers by a left-handed one. This asymmetric system is deadlock free, as can be proved using Checking.

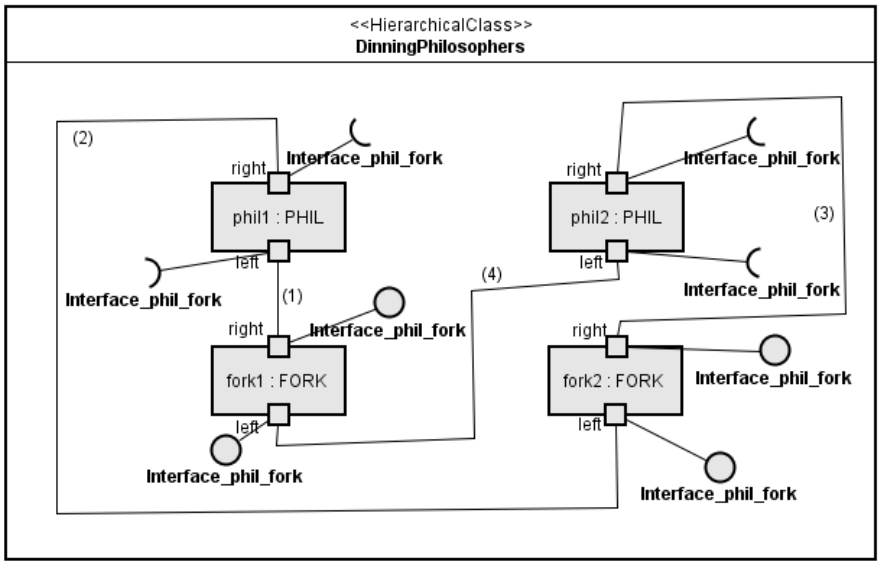

Figure A.6: Component deadlock free

Appendix B

Complete UML component Model - Ring Buffer

a

The Ring Buffer example represents a reactive bounded buffer which is composed by a ring of storage cells with a controller and a cache. Each cell has an identification and is able to store one value. The controller is the responsible to intermediate the communication between the environment and the ring, receiving input and output requests and sending values to be stored inside the cells.

We assume that the values stored in the buffer are natural numbers. The size buffer is a number of cells plus one (the cache) Furthermore, storage cells are identified by natural numbers that range from 1 up to the size of the buffer decremented by one. The controller has four state components: the cache that stores the head of the buffer, when the buffer is non-empty; the size of the list stored in the buffer; and two index bottom and top, to delimit the relevant values.

B.1 Controller Component

The Controller Component is responsible to coordinate how is the storage in cells and cache. It is represented by a *BasicComponent*.

B.1.1 Controller Basic Component

The Figure B.1 illustrates a *BasicComponent Controller*. It has one *BasicComponentClass* with attributes that represents states: *size*, the current size of the buffer; *top*, the index of writable cell and *bot*, the index of the readable cell.

B.1.2 Controller State Machine Diagram

We now describe the Controller's behavioural actions. If the buffer has not reached its maximum size, it is available to get the new input. In the case the buffer is empty, an

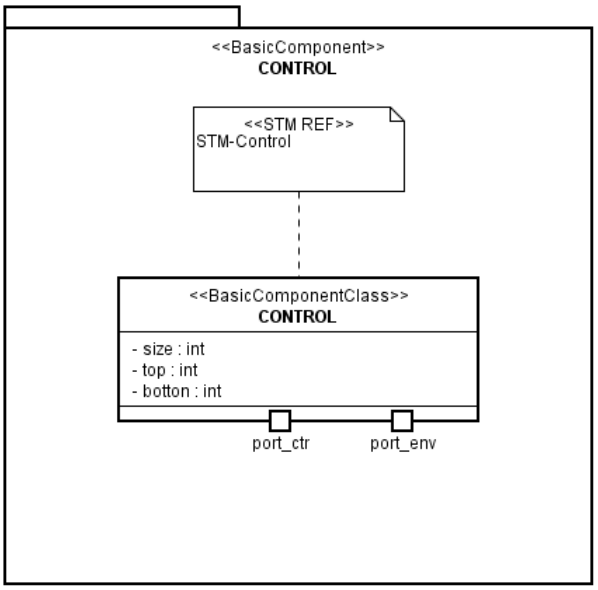

Figure B.1: Controller Class Diagram.

input is cached. The ring indexes do not change and the buffer now contains a single item. If the buffer is not empty, the Controller sends the input value to the ring along with the position top in which the input is to be stored. In this case, the cache is not changed, but the indexes and the size of the ring are updated, they are incremented by one.

Concerning output, which is only enabled if the buffer is not empty, the value in the cache is always the one which is communicated. If the buffer has a single element, communicating this element and updating the size are the only relevant actions. Nevertheless, if there are elements stored in any storage cell, the value x at position bot must be recovered. In this case, the cache is updated with this value and bot is incremented. The following action captures the necessary case analysis for output. See Figure B.2.

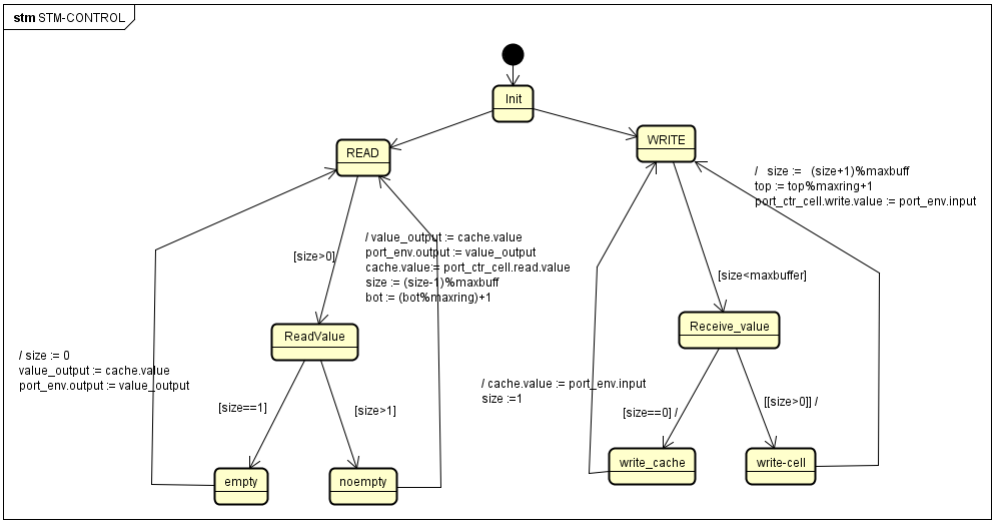

Figure B.2: Controller State Machine Diagram.

B.2 Cell

Each individual cell storage a number. It is modeled as a *BasicComponent*

B.2.1 Cell Basic Component

A *BasicComponent Cell* has a *BasicComponetClass* which contains the port definition : *port_ctr_cell*. This port allow the communication with the component *Controller*. The *STM_REF* indicates the behaviour of *Cell* component. The port *port_cell* allows communication, it required the interface *interface_control_cell* , see Figure B.3

The state machine shown in the Figure B.4 represents the behaviour of a cell. Initially the cell is available to be written, it is identified by the attribute *available*, and the stage is represented by state *available*. When it have been written, the attribute *available* is changed and performs the state *busy*. In this state the cell could be read and become available again.

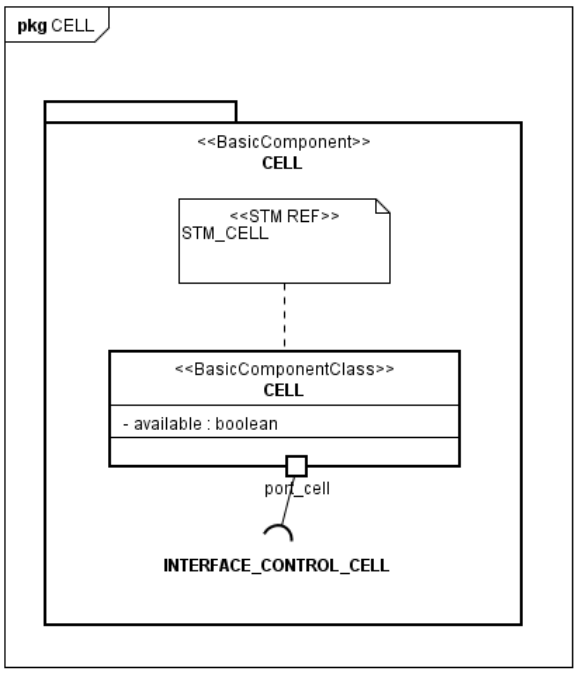

Figure B.3: Cell Class Diagram.

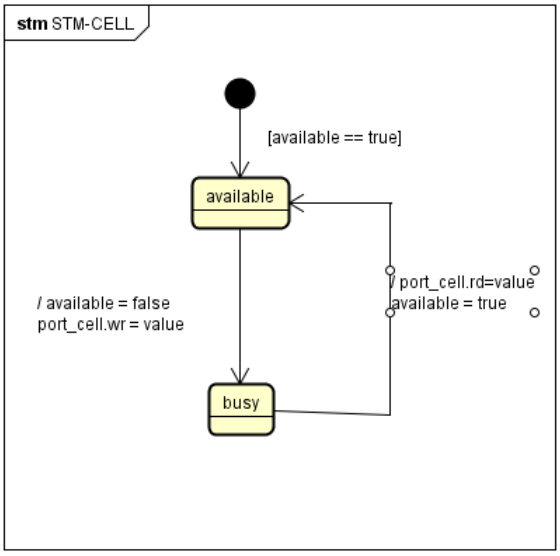

Figure B.4: State Machine Diagram of Cell Component.

B.3 Ring Buffer Class Diagram

The *Ring Buffer* is modelled as a *System* element and, therefore, as a *Hierarchical-Component*; see Figure B.5. It has a *HierarchicalComponentClass* that is related to one *Controller* and one or more *Cell* components, using a composition relationship. In order to allow communication among the controller component and the cells, a common interface is realized by them: INTERFACE CONTROL CELL. Similarly, INTERFACE ENV is the interface of the Controller with the environment.

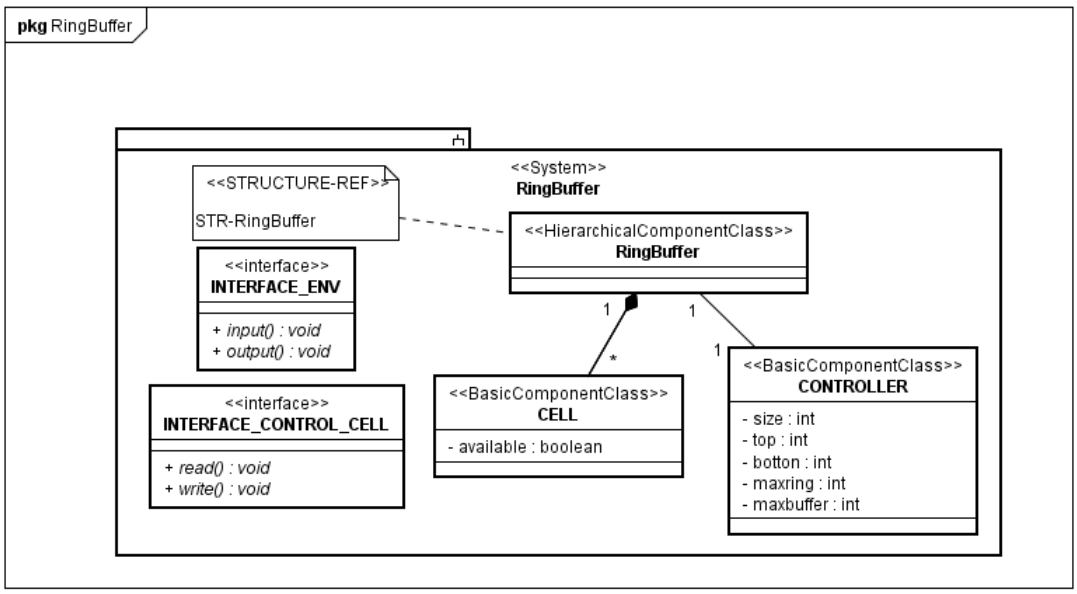

Figure B.5: Ring Buffer System.

B.4 Ring Buffer Composition

The connections are captured by the composite structure diagram STR-RingBuffer; see Figure B.6. We consider a configuration with three cells. The Controller has one conjugate port (with multiplicity three), represented by $port\_ctr[3]$, which is indexed, from 1 to 3, to establish the connections with the three cells. To define which index of port ctr is connected with a port from a Cell component, a label is used, for instance: $port\_cell <-> port\_ctr[1]$. The values to be stored to (and recovered from) the Ring Buffer are communicated by *port_env* (that realises the interface $INTERFACE\_ ENV$) allow exchange with the environment.

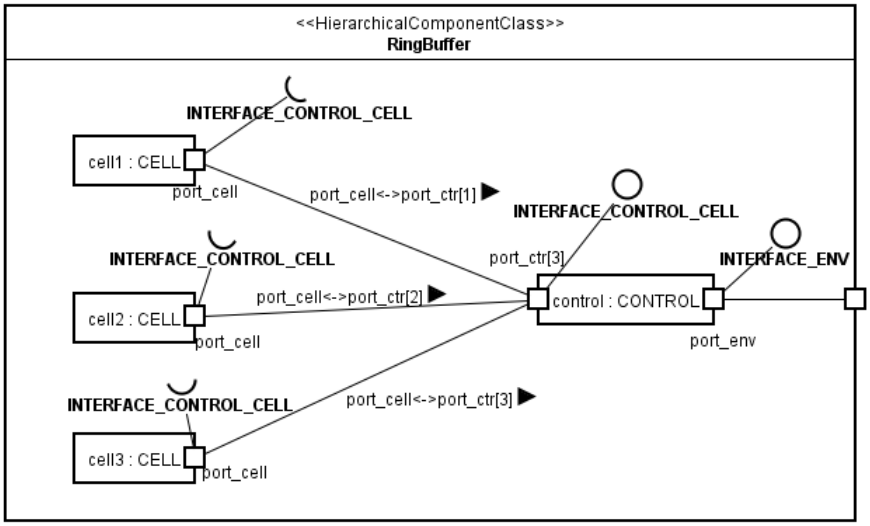

Figure B.6: Ring Buffer Composition.

Appendix C

Complete CSP Translation Model Dining Philosopher

The CSP Specification generated by our translation from UML model need to be made finite in order to be analysed by a model checker.we use the FDR tool [11], which is a CSP refinement checker, to perform our analyses.

C.1 Dining Philosophers

In this section we illustrate the translation from UML Components do CSP. State machine Diagram stm_phil that describe behaviour of one philosopher, Figure A.4 It is translated to CSP in following process. Each state of a state machine is translate to a process. The parameter *id* identifies uniquely the component.

```
stm_phil(id) = HoldingForkR (id);
               HoldingForkL (id);
               PuttingDownR (id);
               PuttingDownL(id);
               stm_phil(id)

HoldingForkL(id) = port_phil_left.id.pickups_I ->
        port_phil_left.id.pickups_O  -> SKIP

HoldingForkR(id) = port_phil_right.id.pickups_I ->
                   port_phil_right.id.pickups_O  -> SKIP

PuttingDownL(id) = port_phil_left.id.putdown_I ->
                   port_phil_left.id.putdown_O -> SKIP

PuttingDownR(id) =  port_phil_right.id.putdown_I ->
                    port_phil_right.id.putdown_O -> SKIP
```

The *Phil* component does not contain attributes in its description, then its is defined as its state machine process:

```
Phil(id) = stm_phil(id)
```

In the same way the State Machine Diagram stm_fork that describes the behaviour of one fork component, Figure A.2 is describe as the following process:

```
available(id) =
    (port_fork_right.id.picksup_I ->
    port_fork_right.id.picksup_O -> busy1(id))
    []
    (port_fork_left.id.picksup_I ->
    port_fork_left.id.picksup_O -> busy2(id))

busy1(id) =
      port_fork_right.id.putsdown_I ->
      port_fork_right.id.putsdown_O -> available(id)

busy2(id) =
     port_fork_left.id.putsdown_I ->
     port_fork_left.id.putsdown_O -> available(id)
```

Component *Fork* does not have attributes as the component *Phil*, then the component is defined like the state machine translation:

```
Fork(id) = stm_fork(id)
```

In order to allow the communication a protocol process is defined:

```
--phil protocol
PROT_PHIL(ch) =    ch.pickups_I-> ch.pickups_O-> ch.putdown_I->
    ch.putdown_O  -> PROT_PHIL(ch)

-fork protocol
PROT_FORK(ch) =    ch.pickups_I-> ch.pickups_O ->ch.putdown_I  ->
    ch.putdown_O -> PROT_FORK(ch)
```

The asynchronous communications are achieved by the use of infinite buffers as intermediate elements of the composition. A buffer is a process with one input channel and one output channel, both having the same type. The buffer copies information from its input channel to its output channel, without losing information and preserving the order. It never refuses inputs, besides it also never refuses outputs when it is non-empty. It is presented below the definition of the buffer,

```

BUFFER_FORK1_FORK2_COMM_PHIL1_PHIL2(ci,co, n) =
  let
    B(<>) = [] x : outputsC_FORK1_FORK2_COMM_PHIL1_PHIL2(ci) @ ci.x ->
    B(<x>)
    B(s)  = (co!head(s) -> B(tail(s)))
                []
             (#s<n & [] x : outputsC_FORK1_FORK2_COMM_PHIL1_PHIL2(ci) @
    ci.x ->              B(s^<x>))
  within B(<>)
```

In this buffer the information that can be communicated are defined by the type of *ci* and *co*. This buffer gathers these information from the function

outputsC_FORK1_FORK2_COMM_PHIL1_PHIL2, which receives a channel ci as parameters and returns all the values x, such that the event ci.x is an output of the component on the channel ci.

```
-- communication
FORK1_FORK2_COMM_PHIL1_PHIL2 =
      (FORK1 ||| FORK2 ||| PHIL1 ||| PHIL2)
        [| {| port_fork_left.1,
              port_phil_right.1|} |]
BFIO_FORK1_FORK2_COMM_PHIL1_PHIL2(
    port_fork_left.1, port_phil_right.1)
```

When two components are connected in UML, such a connection is represented in CSP by the parallel composition of the component processes and a *Buffer* process that orchestrates the communication between the components. Communication in CSP is synchronous while message passing in UML is asynchronous: two events are used to represent the sending and the receiving of a message. Thus, the *Buffer* process simply defines the order in which the events happen through the ports of the components. The synchronisation alphabet of a component process and the buffer is defined by the events *sent to* and *received from* the ports for that particular connection.

Appendix D

Complete CSP Translation Model Ring Buffer

The CSP Specification generated by our translation from UML model need to be made finite in order to be analysed by a model checker we use the FDR tool [11], which is a CSP refinement checker, to perform our analyses.

The translation to CSP from the UML Model of a Controller which is illustrated in Figure B.2. This state machine offers two operations: read and write.

```
stm_ctr(id) =  read(id) [] write (id)
```

A write operation could be describe as a process:

```
write(id) =  internal.1 -> receive_value(id)
receive_value(id) = port_ctr_env_input.id?input -> (
                          write_cache(id,input)
                          []
                          write_cell(id,input))
write_cache(id,input)= internal.2 -> set_size.id!1 -> set_cache.id!input
    -> stm_ctr(id)
write_cell(id,input) = internal.3 ->  get_size.id?size ->
    set_size.id!((size+1)%maxbuff)  ->  get_top.id?top ->
    port_ctr_cell.id.wr_I!input->
                               set_top.id!(top%maxring+1)   ->
    port_ctr_cell.id.wr_O -> stm_ctr(id))
```

As well the read operation is describe as a following process:

```
read(id) = internal.4 -> read_value(id)
read_value(id) =  get_cache.id?vl_output  -> (
                       empty(id,vl_output )
                       []
                       no_empty(id,vl_output ))

empty(id,vl_output )=internal.5 -> set_size.id!0 ->
    port_ctr_env_output.id!vl_output ->  stm_ctr(id)
no_empty(id,vl_output )=
              internal.6 -> get_bot.id?bot   ->
    (port_ctr_env_output.id!vl_output -> port_ctr_cell.id.rd_I ->
```

```
10              port_ctr_cell.id.rd_O?vl -> set_cache.id?vl ->
    get_size.id?size -> set_size.id!((size-1)%maxbuff )->
11                  set_bot.id!(bot%maxring)+1 -> stm_ctr(id) )
```

The following process represents the structural part of the *Controller* component. It allows to access and modify the value of one attribute. The channel *internal* simulate a internal memory; for instance when the channel *internal.5* is synchronized with the process *read*, the attribute size has 1 as value.

```
1  ctr_state(id, cache, size, top , bot) =
2       get_cache.id!cache -> ctr_state(id, cache, size, top, bot)
3       []
4       set_cache.id?v -> ctr_state(id, v, size, top, bot)
5       []
6       get_size.id!size -> ctr_state(id, cache, size, top, bot)
7       []
8       set_size.id?v -> ctr_state(id, cache, v, top, bot)
9       []
10      get_top.id!top -> ctr_state(id, cache, size, top, bot)
11      []
12      set_top.id?v -> ctr_state(id, cache, size, v, bot)
13      []
14      get_bot.id!bot -> ctr_state(id, cache, size, top, bot)
15      []
16      set_bot.id?v -> ctr_state(id, cache, size, top, v)
17      []
18      size < maxbuff & internal.1 -> ctr_state(id, cache, size, top, bot)
19      []
20      size == 0 & internal.2 -> ctr_state(id, cache, size, top, bot)
21      []
22      size >0 &  internal.3 -> ctr_state(id, cache, size, top, bot)
23      []
24      size >0  & internal.4 -> ctr_state(id, cache, size, top, bot)
25      []
26      size ==1 & internal.5 -> ctr_state(id, cache, size, top, bot)
27      []
28      size >1 & internal.6 -> ctr_state(id, cache, size, top, bot)
```

Then the *Controller* component is a CSP process that composes in parallel two other processes, one for the structural part and another related to its behaviour. The former defines a memory for accessing the attributes of the component. The latter results from the translation of the component State Machine. Both processes synchronise on the set of events α, which has events for reading and setting the value of each attribute.

```
1  controller(id) =
2      stm_ctr(id)
3         [|{|get_cache.id,set_cache.id,get_size.id,set_size.id,
4         get_top.id,set_top.id,get_bot.id, set_bot.id,internal|}|]
5      ctr_state(id,0,0,1,1)
6      \ {|get_cache,set_cache, get_size, set_size,
7           get_top, set_top, get_bot, set_bot, internal|}
```

In the same way the UML Component *Cell* is translated. First the State Machine as following:

```
stm_cell(id) = available(id);  stm_cell(id)
available(id) =
            get_available.id?available -> (
              if ( available == 1) then
                 (port_cell.id.wr_I?x:vl-> set_val.id!x ->
                  port_cell.id.wr_O -> set_available.id!0 ->
                      SKIP)
             else
                (port_cell.id.rd_I -> get_val.id?val ->
                port_cell.id.rd_O!val->  set_available.id!1 ->
                     SKIP))
```

After the process that is responsible to structural part.

```
cell_state(id,val,available) =
      get_val.id!val -> cell_state(id,val,available)
      []
      set_val.id?v -> cell_state(id,v,available)
      []
      set_available.id?v -> cell_state(id,val,v)
      []
      get_available.id!available ->cell_state(id,val,available)

```

Then *Cell* component is also a CSP process that composes in parallel the behaviour part (State Machine process) and structural part (cell state process).

```

 cell(id) =
          (stm_cell(id)
     [|{|get_val.id,set_val.id,set_available.id,get_available.id|}|]
          cell_state(id,0,1))
      \ {|get_val,set_val,get_available,set_available|}


```

Similarity to Dining Philosopher *HierarchicalComponent*, it is necessary to define protocol processes.

```
PROT_CELL(e) =  |~| v2:vl @ e.wr_I?v2 -> e.wr_O -> PROT_CELL(e)
                []
                |~| v2:vl @ e.rd_I -> e.rd_O!v2 -> PROT_CELL(e)

PROT_CTR(e) =  |~| v2:vl @ e.wr_I?v2-> e.wr_O -> PROT_CTR(e)
                  []
                |~| v2:vl @ e.rd_I -> e.rd_O!v2 -> PROT_CTR(e)

```

And the asynchronous communications are achieved by the use of infinite buffers as intermediate elements of the composition:

```
 BUFFER_CELL1_CELL2_CELL3_CTR_COM(ci,co, n) =
  let
    B(<>) = [] x : outputsC_CELL1_CELL2_CELL3_CTR_COM(ci) @ ci.x -> B(<x>)
    B(s)  =    (co!head(s) -> B(tail(s)))
```

```
                [] (#s<n & [] x : outputsC_CELL1_CELL2_CELL3_CTR_COM(ci) @
    ci.x -> B(s^<x>))
  within B(<>)

```

Then it is shown a translation to CSP from the connection between the port *port_cell* from *Cell* component (id 2) with the $port_ctr_cell$ (index 1) from *Controler* Component, see Figure B.6.

```
CELL1_CELL2_CELL3_CTR_COM =
             CELL1_CELL2_CELL3_CTR
               [|{|port_cell.2,
                   port_ctr_cell.1|}|]
            BFIO_CELL1_CELL2_CELL3_CTR_COM
            (port_cell.2,port_ctr_cell.1)

```