UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO CIÊNCIA DA COMPUTAÇÃO

Flávia Mérylyn Carneiro Falcão

Safe and Constructive Design with UML Components

Recife

2022

Flávia Mérylyn Carneiro Falcão

Safe and Constructive Design with UML Components

Trabalho apresentado ao Programa de Pós-graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, como requisito parcial para obtenção do grau de Doutor em Ciência da Computação.

Área de Concentração: Engenharia de Software e Linguagens de Programação

Orientador (a): Augusto Cezar Alves Sampaio

Coorientador (a): Lucas Albertins Lima

Recife

2022

Catalogação na fonte
Bibliotecária Nataly Soares Leite Moro, CRB4-1722

F178s Falcão, Flávia Mérylyn Carneiro
Safe and constructive design with UML components / Flávia Mérylyn Carneiro Falcão. – 2022.
126 f.: il., fig., tab.

Orientador: Augusto Cezar Alves Sampaio.
Tese (Doutorado) – Universidade Federal de Pernambuco. CIn, Ciência da Computação, Recife, 2022.
Inclui referências e apêndice.

1. Engenharia de software e linguagens de programação. 2. CSP. 3. Verificação compositional. 4. UML. 5. Análise de deadlock I. Sampaio, Augusto Cezar Alves (orientador). II. Título

005.1 CDD (23. ed.) UFPE - CCEN 2022 – 93

Flávia Mérylyn Carneiro Falcão

“Safe and Constructive Design with UML Components”

Tese de Doutorado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Doutor em Ciência da Computação. Área de Concentração: Engenharia de Software e Linguagens de Programação.

Aprovado em: 29/03/2022.

_____

Orientador: Prof. Dr. Augusto Cezar Alves Sampaio

BANCA EXAMINADORA

_____

Prof. Dr. Alexandre Cabral Mota
Centro de Informática / UFPE

_____

Prof. Dr. Juliano Manabu Iyoda
Centro de Informática / UFPE

_____

Prof. Dr. Márcio Lopes Cornélio
Centro de Informática / UFPE

_____

Profa. Dra. Ana Cristina Vieira de Melo
Instituto de Matemática e Estatística / USP

_____

Prof. Dr. Marcel Vinicius Medeiros Oliveira
Departamento de Informática e Matemática Aplicada / UFRN

A minha mãe.

AGRADECIMENTOS

Primeiramente gostaria de agradecer a Deus por manter minha fé inabalável durante toda a jornada.

Agradeço a minha família, em especial a minha mãe que desde sempre dá o melhor de si para os seus filhos. E vibra com cada realização minha.

Aos meus orientadores Augusto Sampaio e Lucas Lima pela enorme paciência, compreensão, orientação, revisões e incentivo que fizeram desse trabalho possível. Aos colegas Sidney Nogueira e Dalay Pereira que compartilharam de seus conhecimentos.

A todos meus amigos que sempre me incentivaram e me apoiaram.

Por fim, a todos que direta ou indiretamente colaboraram para a realização deste trabalho.

"Ninguém educa ninguém, ninguém educa a si mesmo, os homens se educam entre si, mediatizados pelo mundo. Ninguém liberta ninguém. As pessoas se libertam em comunhão. Ninguém nasce feito, é experimentando-nos no mundo que nós nos fazemos." (FREIRE, 1970, p.33)

"Nobody said it was easy." (COLDPLAY, 2002)

ABSTRACT

Model-based engineering emerged as an approach to tackle the complexity of current systems development. In particular, compositional strategies assume that systems can be built from reusable and loosely coupled units. However, it is still a challenge to ensure that desired properties hold for component integration. BRIC provides an approach for developing component-based systems which guarantee deadlock freedom. Then, we present a component based model for UML, including a metamodel, well-formedness conditions and formal semantics via translation into BRIC; the presentation of the semantics is given by a set of rules that cover all the metamodel elements and map them to their respective BRIC denotations. We use 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 (Communicating Sequential Processes). Component composition, specified via UML structural diagrams, ensures adherence to classical concurrent properties: our focus is on the preservation of deadlock freedom. Automated support is developed as a plug-in to the Astah modelling tool. The verification is carried out using FDR (a model checker for CSP), but, this is transparent to the user. A distinguishing feature of our approach is its support for traceability. For instance, when FDR uncovers a deadlock, a sequence diagram is constructed from the deadlock trace and presented to the user at the modelling level. We illustrate our overall approach with a running example and two additional case studies. We also emphasise the contributions of the proposed component model and modelling strategy via a comparison with other approaches in the literature.

Keywords: CSP; component; compositional verification; UML; deadlock analysis.

RESUMO

A Engenharia de Software baseada em modelos surgiu como uma abordagem para lidar com a complexidade do desenvolvimento de sistemas atuais. Em particular, as estratégias de composição assumem que os sistemas podem ser construídos a partir de unidades reutilizáveis e fracamente acopladas. No entanto, ainda é um desafio garantir que propriedades desejadas sejam válidas para a integração de componentes. BRIC provê uma abordagem para desenvolvimento baseado em componentes que garante a ausência de *deadlock*. Então, apresentamos um modelo baseado em componentes para UML, incluindo um metamodelo, condições de boa formação e semântica formal via tradução para BRIC; a apresentação da semântica é dada por um conjunto de regras que abrangem todos os elementos do metamodelo e os mapeiam para suas respectivas denotações BRIC. Usamos BRIC como um framework de desenvolvimento de componentes subjacente (e totalmente oculto) para que nossa abordagem se beneficie de toda a infraestrutura formal desenvolvida para BRIC usando CSP (Communicating Sequential Processes). A composição do componente especificada por meio de diagramas estruturais UML, garante a aderência às propriedades concorrentes clássicas: nosso foco é a preservação da ausência de *deadlock*. O suporte automatizado é desenvolvido como um *plug-in* para a ferramenta de modelagem Astah. A verificação é realizada usando FDR (um verificador de modelos para CSP), mas isso é transparente para o usuário. Um diferencial de nossa abordagem é o suporte à rastreabilidade. Por exemplo, quando o FDR descobre um *deadlock*, um diagrama de sequência é construído a partir do *trace* de deadlock e apresentado ao usuário como um modelo UML. Ilustramos a aplicabilidade da nossa abordagem com um exemplo apresentado de forma recorrente no texto e dois estudos de caso adicionais. Destacamos também as contribuições do modelo de componentes proposto e da estratégia de modelagem por meio de uma comparação com outras abordagens da literatura.

Palavras-chaves: CSP; componente; verificação compositional; UML; análise de deadlock.

LISTA DE FIGURAS

LISTA DE CÓDIGOS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUCTION

Modelling is central to all activities that lead up to the deployment of well-designed software. Models are built to communicate the desired structure and behaviour of the system; to visualise and to control the system's architecture; and to provide a better understanding of the system, often exposing opportunities for simplification and reuse (BOOCH; RUMBAUGH; JACOBSON, 2005). When reusable units are independent and well defined, they can be called components.

Component-based software development (CBSD) is a widely disseminated paradigm to build software systems by integrating independent and potentially reusable components. One of the motivations for this paradigm is replacing conventional programming with the systematic composition and configuration of components (OLIVEIRA et al., 2016).

In order to ensure the success of component-based software development, it is essential to assure the correct behaviour of the components. Such trustworthiness is even more important in critical applications.

In some contexts, particularly when there is some critical aspect involved, a reliable architecture becomes a demand. The architecture 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.

Therefore, a systematic approach, both to create new components from existing ones, and to ensure that each composition preserves the desired properties, seems a promising direction to follow.

Formal verification can greatly increase the understanding of a system by revealing inconsistencies, ambiguities, and incompletenesses that might otherwise go undetected (CLARKE; WING, 1996). Particularly, *model checking* is a well-established approach for verification that relies on building a finite model of a system and checking that the desired properties hold in that model, like, for example, deadlock and livelock freedom.

There are several approaches to CBSD in the literature that include a formal method as the main outline. For instance, in (CHEN et al., 2009) the authors present component-based refinement that focuses on the separation of interface and functional contracts, supporting different levels of abstraction. The approach in (BONAKDARPOUR et al., 2012; BASU; BOZGA; SIFAKIS, 2006) 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 (HORVÁTH et al., 2020), the authors proposed a cloud-based, end-to-end verification workflow for SysML (Object Management Group (OMG), 2017a) State Machines and reachability properties using an intermediate language and different model checkers; formal aspects are hidden from the engineers. Model checking is fully automated via translations, and traceability is provided through back annotations of the resulting trace.

As far as we are aware, none of the existing approaches provide an integrated framework that include: a formal presentation of a component model, encompassing well-formedness conditions and a semantics systematically presented as a set of rules; support for stepwise design and compositional verification of classical properties such as deadlock freedom, and traceability from the counterexample verification to the component model. For instance, the approaches in (CHEN et al., 2009; BONAKDARPOUR et al., 2012; BASU; BOZGA; SIFAKIS, 2006) embody a component model, but do not provide traceability, nor compositional reasoning. The work in (HORVÁTH et al., 2020) proposes a component model with support for traceability, but verification is not compositional.

In (RAMOS; SAMPAIO; MOTA, 2009; OLIVEIRA et al., 2016) the authors proposed a formal component model, together with a rule-based composition strategy, called BRIC. BRIC has the process algebra Communicating Sequential Processes CSP (ROSCOE, 1997) as an underlying semantic model. Given that the argument components are deadlock free, each composition rule ensures that the resulting (composed) component preserves deadlock freedom. Despite the promising results, a developer needs to have considerable knowledge of CSP and model checking techniques to use BRIC.

UML (Object Management Group (OMG), 2016) is well-suited for modelling software systems in general; however, it lacks support for a CBSD approach. Components in UML are assumed to be concrete and executable artefacts. Another design element 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.

We propose a formal CBSD model for UML, motivated by the fact that UML is a widely used notation in industry and amenable to mechanised analysis. We consider the benefits from the overall formal infrastructure built around BRIC as a semantic model for the proposed UML

component model.

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 the component systems. We proposed a set of well-formedness conditions, in order to define how components can be composed to give rise to more elaborate components. The well-formedness conditions are presented as rules expressed using OCL (Object Constraint Language) (Object Management Group (OMG), 2014).

As another contribution, we define a compositional semantics for the proposed component model by translation into BRIC. We provide a complete formalisation as a set of rigorously defined rules that cover all the model elements.

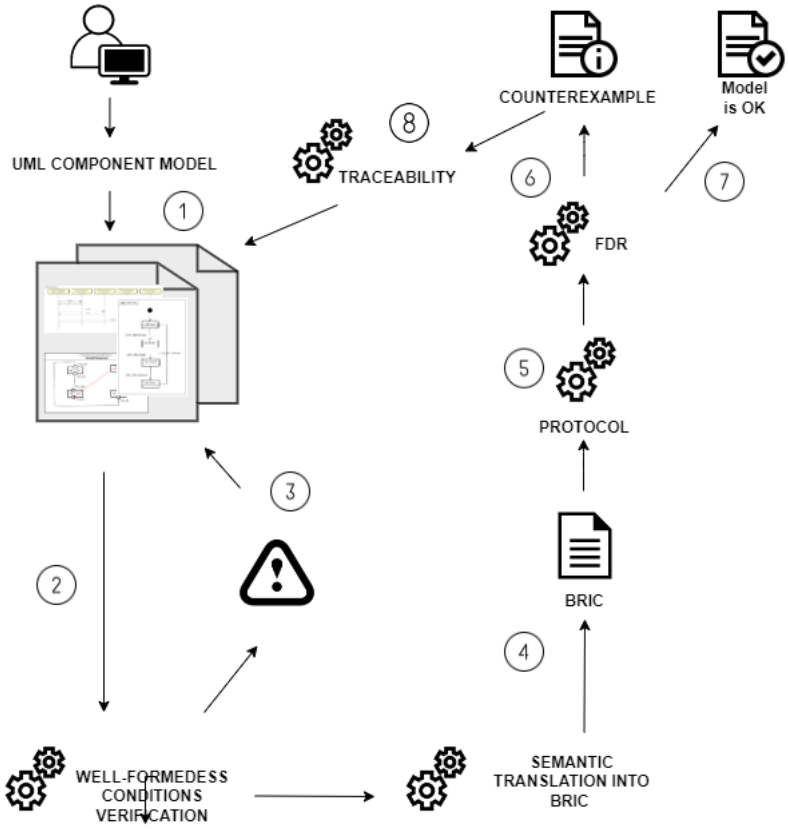
Components, instances and connections are translated into BRIC denotations (contracts and CSP processes), and verification of several properties is conducted using FDR according to the BRIC composition rules. If the verification fails, we provide a traceability mechanism from the CSP counterexample to a UML state machine or sequence diagram. The former is applied to the early verification of the assumed properties of BRIC contracts, as detailed in Section 2.3, and the latter for deadlock issues.

As yet another contribution, we present a strategy for the automatic generation of protocol implementation (a projection of the behaviour of a component over a given set of channels), which is an important element to ensure that the BRIC compositions indeed preserves deadlock freedom.

As our final contribution, all these are automated as a plug-in for the Astah modelling tool (CHANGE, 2020). Astah provides a solution to support the modelling of UML diagrams. We use the Astah Java API to access model information and to create new diagrams. The plug-in uses the FDR tool in background, for checking the side conditions of the composition rules at the CSP semantic level. Astah is a modelling tool that runs on top of the JVM (Java Virtual Machine). Our plug-in is built on top of its UML version. Astah can be extended by the addition of plug-ins to add new features.

Figure 1 illustrates the process of the verification steps of our approach. In step 1, system engineers build components according to the UML component model, whose syntax is precisely defined as a metamodel in UML. Step 2 corresponds to the verification of the well-formedness conditions. If any well-formedness condition is violated, a warning message is shown indicating where the problem is (Step 3). Step 4 corresponds to the automatic translation of the UML

Figure 1 – The major steps of the proposed approach



Source: Author's ownership.

model into BRIC. Then, in step 5, the communication protocol is generated. This allows the analysis using the FDR model checker, which is shown in Step 6. If an analysis fails, the resulting counterexample is traced back to the component model in the form of UML diagrams in Step 7. Otherwise, the model was correctly specified, step 8.

In summary, the contributions of this thesis are the following:

- A UML component metamodel, using a subset of UML;
- Formal presentation of the well-formedness conditions in OCL;
- Formal semantics of the component model in BRIC, based on a set of rigorously defined semantic rules;
- A strategy for the automatic generation of protocol implementations;
- Traceability via the presentation of verification results in FDR as UML diagrams;
- A tool that includes the fully automatic semantic generation and support for traceability of well-formedness condition violations as well as deadlock scenarios;

- Case studies: dining philosophers, ring buffer and Leadership election protocol.

1.1 STRUCTURE OF THE THESIS

We have organised the presentation of this thesis in the following structure. Chapter 2 is devoted to background knowledge. In this chapter, we introduce the formalism that underlies our approach, the CSP process algebra. We describe its notation and three of its semantic models: traces, failures and failures-divergences. Then, we present the BRIC component model; we describe its set of composition rules, which can be used to develop trustworthy systems, guaranteeing, by construction, the absence of deadlock. Next, we briefly describe Labelled Transition System (LTS) and weak bisimilation. These are relevant to our approach to automatic generation of protocol implementations. We also describe the UML diagrams which are used in our component metamodel.

Chapter 3 presents the proposed UML component model, the well-formedness conditions, and the component (instance) composition approach.

Chapter 4 describes the semantic translation into BRIC together with the strategy for the automation of protocol implementation.

Chapter 5 presents the evaluation of our approach. It is dedicated to tool support and the development of case studies.

Chapter 6 presents related work. We analyse the advantages and limitations of our approach in the context of related approaches.

Chapter 7 summarises our conclusions, emphasising our main contributions. Finally, we discuss some topics for future work.

2 BACKGROUND

This chapter presents relevant background for this work. Firstly we present the formalism that we use in our approach, Communicating Sequential Processes (CSP). Next, we introduce Labelled Transition System (LTS) and weak bisimulation. In addition, we show the BRIC component model, which is an approach for compositional development of asynchronous systems. Finally, we briefly discuss the UML diagrams that are relevant to our work.

2.1 CSP - COMMUNICATING SEQUENTIAL PROCESSES

The process algebra CSP (Communicating Sequential Processes) (HOARE, 1985; ROSCOE, 1997) is a notation used to describe concurrent systems whose processes interact by exchanging messages. It provides a set of semantic models that help one to reason about processes and how they interact. An advantage of CSP is that it offers consolidated semantic models, as well as a formal theory of refinement and verification. In what follows, we describe the CSP constructs that we use in this work, and we briefly present some of its denotational models.

2.1.1 CSP Syntax

In CSP, communication between processes takes place over named channels and their synchronous events. CSP has two basic processes: $SKIP$ and $STOP$. The former does nothing and communicates the special event ✓, which is a visible event that indicates successful termination of a process. The process $STOP$ represents a broken process (deadlock); it is not capable of communicating any event.

Processes are defined in terms of a set of events of an alphabet (Σ). An event is a single, atomic, and instantaneously occurring action that a process might engage in. Given an event a in the alphabet of a process P, the prefixing $a \rightarrow P$ is initially able to perform a, after which it will behave like the process P. A structured event is given by a communication channel that may carry values, and its declaration form is: $channel \quad c : T$. In this declaration, c is the name of the channel, and T is the type of values communicated through it. The set of all events on a channel c is the set $c.T = \{c.x \mid x \in T\}$, which is a subset of Σ. An input communication on c has the form $c?x$, and an output communication has the form $c!e$. The

expression $c!e$ is semantically equivalent to $c.e$, when e is an expression that denotes a single expression.

In concurrent systems, it is useful to distinguish between the cases where control over the resolution of choice resides within a process itself and where control is outside it (SCHNEIDER, 1999). The external choice operator ($\Box$), offers a deterministic choice to the environment. The process $P \Box Q$ combines two processes P and Q, and initially offers both behaviours of P and Q to the environment; the environment chooses which one to perform. Once one of the behaviours is chosen, the process $P \Box Q$ behaves as the chosen process, either P or Q.

The internal choice ($\sqcap$) represents a non-deterministic choice. It also combines two processes but in a non-deterministic way; the choice is internally made by the process. In the process $P \sqcap Q$ the environment has no control over the choice between P and Q. The process internally resolves the choice.

CSP offers parallel operators, possibly allowing process interaction. The interleaving operator represents the composition of two processes in parallel but with no interaction. This means that in $P \,|||\, Q$, P and Q can perform their events independently, without any communication between them. The generalised parallel operator takes two processes, *P* and *Q*, and a set of events, say X, as arguments. The resulting process $P \underset{X}{||} Q$ allows P and Q to proceed independently when performing events outside X, but they must synchronise in the events belonging to X.

CSP provides what is called replicated forms of some of its operators. For example, the replicated external choice $\Box\ x : A \bullet P(x)$ evaluates $P(x)$ for each value of A and composes the resulting processes using external choice. Similarity, the construction $|||\ x : A \bullet P(x)$ evaluates P for each value x of A and interleaves these processes.

Hiding is an operator that is used to hide the events of a process. The process $P \setminus S$ can perform any event that is in the alphabet of P and not in the set of events S. On the other hand, when P performs an event in S, the process $P \setminus S$ makes this event internal, represented as τ (tau).

The renaming operator $P[\![R]\!]$ takes a process P and a renaming relation R that contains a list of pairs $a \leftarrow b$. The process $P[\![a \leftarrow b]\!]$ behaves like the process P, but occurrences of the event a are replaced by occurrences of the event b. For example, given a process $P = a \rightarrow SKIP$, the process $P[\![a \leftarrow b]\!]$ results in the process $b \rightarrow SKIP$ that initially offers the event b and then successfully terminates.

The CSP notation allows expressing recursive behaviour by using the name of the process

in its definition. For instance, $P = a \rightarrow P$ performs a and then behaves as P.

A boolean guard may be associated with a process: given a predicate z , if the condition z is true, the process $z \ \& \ P$ behaves like P; otherwise, it deadlocks.

Processes can be combined in sequence using the sequence operator (;). The process $P; \ Q$ first behaves as P and then, if P successfully terminates (ends by a SKIP), behaves as Q.

As a running example, we consider the classical dining philosophers problem where the philosophers are seated at a round table with a single fork between each pair of philosophers. Each philosopher requires both neighbouring forks to eat, so if all get hungry simultaneously and pick up their left-hand fork, then they deadlock and starve to death. It actually captures one of the major causes of real deadlocks, namely competition for resources (ROSCOE, 1997). In our approach, the fork and philosopher behaviours are written as simple CSP processes. A process that captures the behaviour of a *Fork* is as follows.

$$Fork(id) = STM\_Fork(id)$$

$$STM\_Fork(id) = Available(id)$$

$$Available(id) = (fork\_right.id.picksup\_I \rightarrow fork\_right.id.picksup\_O \rightarrow Busy1(id))$$

$$\Box$$

$$(fork\_left.id.picksup\_I \rightarrow fork\_left.id.picksup\_O \rightarrow Busy2(id))$$

$$Busy1(id) = (fork\_right.id.putsdown\_I \rightarrow fork\_right.id.putsdown\_O \rightarrow Available(id))$$

$$Busy2(id) = (fork\_left.id.putsdown\_I \rightarrow fork\_left.id.putsdown\_O \rightarrow Available(id))$$

The process *Fork* is parametrised by its id so that several instances for distinct identifiers can be created. All events associated to forks are represented by the channels $fork\_right$ and $fork\_left$ that, apart from the id, can communicate the data $picksup\_I$, $picksup\_O$, $putsdown\_I$ and $putsdown\_O$. These values represent the actions offered by forks. Initially, a fork is available for both philosophers; however, two philosophers can not hold the same fork simultaneously.

The events related to picking actions are always followed by the events related to putting a fork down. The external choice in the process $Available$ means that if the first choice is taken, the philosopher, on the right (of the fork) holds the fork, and similarly for the one on the left In the first case, the process performs the event $fork\_left.id.picksup\_I$, and then the event $fork\_left.id.picksup\_O$, where the former represents the intention to pick the fork, and

the latter indicates that it has been performed [1]; finally, it behaves as the process $Busy1$.

The process $Busy1$ engages in two events in sequence, capturing the release of a fork and then behaving again as *Available*. The process $Busy2$ is analogous, dealing with the second choice.

A process that captures the behaviour of a *Phil* is as follows.

$$STM\_Phil(id) = HoldForkR(id)$$

$$HoldForkR(id) = (phil\_right.id.picksup\_I \rightarrow phil\_right.id.picksup\_O \rightarrow HoldForkL(id))$$

$$HoldForkL(id) = (phil\_left.id.picksup\_I \rightarrow phil\_left.id.picksup\_O \rightarrow PutsDownR(id))$$

$$PutsDownR(id) = (phil\_right.id.putsdown\_I \rightarrow phil\_right.id.putsdown\_O \rightarrow PutsDownL(id))$$

$$PutsDownL(id) = (phil\_left.id.putsdown\_I \rightarrow phil\_left.id.putsdown\_O \rightarrow HoldForkR(id))$$

Similar to process *Fork*, process *Phil* is parametrised by *id* that identify the philosopher. In the dining philosophers problem, each philosopher must pick up two forks to avoid starving: the right and the left, the former represented by the channel $phil\_right$ and the latter represented by the channel $phil\_left$ that communicate the events $picksup\_I$ and $picksup\_O$. Before eating and put them down afterwards, these are represented by the events $putsdown\_I$ and $putsdown\_O$ communicated through the channels $phil\_right$ and $phil\_left$. After the philosopher is able to start a new cycle and picks up a fork.

2.1.2 CSP Semantic Models

A process written in CSP may be understood in terms of operational, algebraic and denotational semantics. The operational semantics transforms the processes into Labelled Transition Systems (LTS), which is a directed graph with a label on each edge representing what happens when taking an action: each transition represents a possible event. An algebraic semantics is defined by a set of algebraic laws. A denotational semantics maps processes into some abstract

[1] The use of a pair of events to represent a communication is a consequence of the asynchronous model adopted in BRIC, as explained later.

model that captures different types of behaviours such as determinism, deadlock and livelock-freedom, recording different sorts of information about a process. All denotational models are compositional, since the set of possible behaviours of a process can be calculated in terms of the denotational values of its subcomponents.

In the following subsections we describe the three main denotational models of CSP: traces, stable failures and failures-divergences (ROSCOE, 2010); these models are of particular interest for our work.

2.1.2.1 The Traces Model

The traces model denotes a CSP process according to its traces, which are defined as the set of sequences of events in which the process may engage.

Given a CSP process P, the traces of P are denoted as $traces(P)$, and the symbol $\Sigma^{\checkmark}$ ($\Sigma \cup \{\checkmark\}$) represents the set of all the possible events for processes in the universe under consideration, including the special termination event ✓(tick). Table 1 presents the semantic clauses of the basic process in this model. The complete list for all CSP operators can be found in (ROSCOE, 2010).

Table 1 – Semantic clauses for the traces model

$$
\begin{aligned}
traces(STOP) &= \{\langle\rangle\} \\
traces(SKIP) &= \{\langle\rangle, \langle\checkmark\rangle\} \\
traces(a \rightarrow P) &= \{\langle\rangle\} \cup \{\langle a\rangle^\frown s \mid s \in traces(P)\} \\
traces(P \,\Box\, Q) &= traces(P) \cup traces(Q) \\
traces(P \sqcap Q) &= traces(P) \cup traces(Q) \\
traces(P \,|||\, Q) &= \bigcup\{s \,\Big|\Big|\Big|\, t \mid s \in traces(P) \wedge t \in traces(Q)\} \\
traces(P \,||\, Q) &= \bigcup\{s \,||\, t \mid s \in traces(P) \wedge t \in traces(Q)\} \\
traces(P \overset{X}{\setminus} X) &= \{s \setminus \overset{X}{X} \mid s \in traces(P)\}
\end{aligned}
$$

Source: (ROSCOE, 2010)

According to the Table 1, there is one trace associated with the process $STOP$, which is the empty trace, $\langle\rangle$, since this process never communicates anything. The traces of $SKIP$ are the empty trace and the trace with the singleton termination event ✓. The traces of the prefix process $a \rightarrow P$ are the traces formed of the event a followed by the traces of the process P, in addition to the empty trace. Internal and external choices are not distinguished in the traces model. Both result in the union of the traces of the two operands. The traces of $P \,|||\, Q$ are just

the interleavings of traces of P and Q. The operator $|||$ on traces produces the interleavings of a given pair of traces recursively. The traces of a generalised parallel composition are defined in terms of the operator $\underset{X}{||}$ over traces, which combines these traces in all possible ways but forces them to agree on X. The traces resulting from hiding a set of events $(P \setminus X)$ is given by preserving only those events that are not in X.

The trace set of process *fork1* $=$ *Fork(1)* includes the following sequences.

$$\{\langle\rangle, \langle fork\_right.1.picksup\_I\rangle, \langle fork\_right.1.picksup\_I, fork\_right.1.picksup\_O\rangle,$$
$$\langle fork\_right.1.picksup\_I, fork\_right.1.picksup\_O, fork\_right.1.putsdown\_I\rangle,$$
$$...\}$$

In the traces model for CSP, a process P is a traces refinement of a process Q (written as $Q \sqsubseteq_{\mathrm{T}} P$) if, and only if, Q contains all traces of P. That is: $Q \sqsubseteq_{\mathrm{T}} P \Leftrightarrow (traces(P) \subseteq traces(Q)$. These processes are traces equivalent (written as $P \equiv_{\mathrm{T}} Q$); if $Q \sqsubseteq_{\mathrm{T}} P$ and $P \sqsubseteq_{\mathrm{T}} Q$, i.e., $traces(P) = traces(Q)$.

The traces model describes what a process may do, but not what a process must do. Moreover, this model does not distinguish internal and external choices. For example, the process $P = a \rightarrow STOP \ \Box \ b \rightarrow STOP$ has the same traces of the process $Q = a \rightarrow STOP \sqcap b \rightarrow STOP$. P is not able to refuse neither a nor b, and accepts both of them. On the other hand, the process Q initially refuses either a (and accepts b) or b (and accepts a), but not both. As consequence, the traces model is not expressive enough for detecting the presence or absence of nondeterminism.

2.1.2.2 The Stable Failures Model

The stable failures model gives more information about processes. For instance, it allows us to distinguish between internal and external choices. It also allows us to detect deadlocked processes. The stable failures model defines a process as its set of traces and a set of failures of a process. A failure of a process is a pair (s, X), that describes a set of events X which a process can fail to accept after executing the trace s. The set X is called the refusal set; the process cannot perform any event in the set X.

The notation $failures(P)$ represents the set of all failures of P. Similar to the trace semantics, the clauses in Table 2 determine the failures of the various processes in the stable failures model.

Table 2 – Semantic clauses for the stable failures model

$$
\begin{aligned}
failures(STOP) &= \{(\langle\rangle, X) \mid X \subseteq \Sigma^{\checkmark}\} \\
failures(SKIP) &= \{(\langle\rangle, X) \mid X \subseteq \Sigma\} \cup \{(\langle\checkmark\rangle, X) \mid X \subseteq \Sigma^{\checkmark}\} \\
failures(a \rightarrow P) &= \{(\langle\rangle, X) \mid a \notin X\} \cup \{(\langle a\rangle^\frown s, X) \mid (s, X) \in failures(P)\} \\
failures(P \,\Box\, Q) &= \{(\langle\rangle, X) \mid (\langle\rangle, X) \in failures(P) \cap failures(Q)\} \cup \\
&\quad \{(t, X) \mid (t, X) \in failures(P) \cup failures(Q) \wedge t \neq \langle\rangle\} \cup \\
&\quad \{(\langle\rangle, X) \mid X \subseteq \Sigma \wedge \langle\checkmark\rangle \in traces(P) \cup traces(Q)\} \\
failures(P \sqcap Q) &= failures(P) \cup failures(Q) \\
failures(P \,|||\, Q) &= \bigcup\{(s \,|||\, t, Y \cup Z) \mid Y \backslash \{\checkmark\} = Z \backslash \{\checkmark\} \wedge \\
&\quad (s, Y) \in failures(P) \wedge (t, Z) \in failures(Q)\} \\
failures(P \underset{X}{||} Q) &= \bigcup\{(s \underset{X}{||} t, Y \cup Z) \mid Y \backslash (X \cup \{\checkmark\}) = Z \backslash (X \cup \{\checkmark\}) \wedge \\
&\quad (s, Y) \in failures(P) \wedge (t, Z) \in failures(Q)\} \\
failures(P \setminus X) &= \{(t \setminus X, Y) \mid (t, Y \cup X) \in failures(P)\}
\end{aligned}
$$

Source: (ROSCOE, 2010)

In Table 2 the process $STOP$ initially refuses to communicate anything. The process $SKIP$ initially cannot refuse the termination event $\checkmark$. However, all events are refused after termination. After the empty trace, the prefix process $a \rightarrow P$ cannot refuse the prefixing event a. Furthermore, once the event a has occurred, the rest of the stable failure derives from process P. In an interleaving of two processes $P \,|||\, Q$ an event will be refused by the combination only when it is refused by both processes. [2] Any failure of the parallel process $P \underset{X}{||} Q$ will be a combination of failures of its two argument processes (SCHNEIDER, 1999). The stable states of $P \setminus X$ correspond precisely to states of P that cannot perform any element of X, which is equivalent to saying that they can refuse the whole of X. Stable states are where no transition is chosen nondeterministically; they are those in which there are no choices between external and internal actions.

Differently from the previous model, in the stable-failures model we can distinguish external and internal behaviours. For example, considering the processes P and Q over the alphabet $\{a, b\}$: Let $P = (a \rightarrow STOP) \,\Box\, (b \rightarrow STOP)$ and $Q = (a \rightarrow STOP) \sqcap (b \rightarrow STOP)$. These processes have the same traces. However, the stable failure sets of P and Q are different. Initially, P can neither refuse a nor b, since the choice is made by the environment. On the other hand, the process Q can initially refuse either a or b because the choice is made internally by the process.

[2] $x \backslash y$ means: $\{a \in x \mid a \notin y\}$

$$
\begin{aligned}
failures(P) = \{(\langle\rangle, \{\checkmark\})\} \\
&\cup \{(\langle a\rangle, X) \mid X \subseteq \{a, b, \checkmark\}\} \\
&\cup \{(\langle b\rangle, X) \mid X \subseteq \{a, b, \checkmark\}\}
\end{aligned}
$$

$$
\begin{aligned}
failures(Q) = \{(\langle\rangle, X) \mid X \subseteq \{a, \checkmark\}\} \\
&\cup \{(\langle\rangle, X) \mid X \subseteq \{b, \checkmark\}\} \\
&\cup \{(\langle a\rangle, X) \mid X \subseteq \{a, b, \checkmark\}\} \\
&\cup \{(\langle b\rangle, X) \mid X \subseteq \{a, b, \checkmark\}\}
\end{aligned}
$$

A process P is a stable failures refinement of process Q (written as $Q \sqsubseteq_{\mathrm{F}} P$) if, and only if, Q contains all traces of P and P presents the same or less stable failures than Q; it refuses the same or less communications. That is: $Q \sqsubseteq_{\mathrm{F}} P \Leftrightarrow (traces(P) \subseteq traces(Q) \wedge failures(P) \subseteq failures(Q))$. Two processes P and Q are stable failures-equivalent, $P \equiv_{\mathrm{F}} Q$, if $P \sqsubseteq_{\mathrm{F}} Q$ and $Q \sqsubseteq_{\mathrm{F}} P$, i.e., $traces(P) = traces(Q)$ and $failures(P) = failures(Q)$.

The stable-failures model allows one to capture liveness properties, such as deadlock freedom. Deadlock arises when two processes cannot agree to communicate with each other nor with any other process. The simplest example of a deadlocked process in CSP is the process $STOP$. In this context, a process P is deadlock-free if P after performing any trace t never becomes equivalent to $STOP$. The stable-failures model is effective in contexts where process divergence (infinite execution of internal actions) is not relevant. When divergence is a possibility then this model is not expressive enough, since it completely ignores any divergent behaviour that a process might have.

2.1.2.3 The Failures-Divergences model

The failures-divergences model allows one to detect not only deadlocked but also livelocked (divergent) processes. A divergence (or livelock) occurs when a process can perform only internal events indefinitely. Livelock is even worse than deadlock ($STOP$) because it behaves like an endless loop that may consume unbounded computing resources without achieving anything (HOARE, 1985).

The hiding operator is the most subtle and difficult one to deal with in the failures-divergences model; this is because it turns visible actions into τ's and thus removes stable states and potentially introduces divergences (ROSCOE, 1997). For instance, consider the processes

$P = P$ and $Q = (a \rightarrow Q) \setminus \{a\}$. Q converts the external event a into an internal action τ. Therefore, Q indefinitely performs internal actions, which leads to a divergence. As a consequence, Q and P have the same behaviour in the failures-divergences model. The CSP process DIV represents the livelock phenomenon: it can refuse every event, and it diverges after any trace.

In the failures-divergence model, the processes are represented by two sets of behaviours: the failures and the divergences. So, each process P is modelled by the pair: $(failures_{\perp}(P)$, $divergences(P))$, where:

- $divergences(P)$ is the set of traces s after which a process can diverge.
- $failures_{\perp}(P)$ represents all the stable failures of P extended by all the pairs (s, X) for $s \in divergences(P)$ and $X \subseteq \Sigma$, allowing the process to refuse anything after diverging.

$$failures_{\perp}(P) = failures(P) \cup \{(s, X) \mid s \in divergences(P)\}$$

A process P is divergence-free if, and only if, $divergences(P) = \{\}$. Similar to the previous models, the clauses in Table 3 determine the divergences of the various processes in the failures-divergences model (where Σ^w is the set of infinite traces, and $traces_{\perp}(P) = traces(P) \cup divergences(P)$). As the process $SKIP$ only communicates the event ✓, it has no infinite traces, and, consequently, it cannot diverge. Similarly, the process $STOP$ does not diverge because it performs nothing. The set of traces after which $a \rightarrow P$ can diverge are prefixed by a. The calculation of divergences for external and internal choices are the union of the divergences of the two operands. An interleaved combination, $P \mid\mid\mid Q$, diverges as soon as one of its processes does. Similarly, divergences in a parallel composition, $P \underset{X}{||} Q$, will arise from divergences of either process (P or Q).

Table 3 – Semantic clauses for the failures-divergences model

$$
\begin{aligned}
divergences(STOP) &= \emptyset \\
divergences(SKIP) &= \emptyset \\
divergences(a \rightarrow P) &= \{\langle\rangle\} \cup \{\langle a\rangle^\frown s \mid s \in divergences(P)\} \\
divergences(P \,\Box\, Q) &= divergences(P) \cup divergences(Q) \\
divergences(P \sqcap Q) &= divergences(P) \cup divergences(Q) \\
divergences(P \,|||\, Q) &= \{u \mid \exists\, s : traces_\perp(P), t : traces_\perp(Q) \mid u \in (s \,|||\, t) \cap \Sigma^* \wedge \\
&\qquad (s \in divergences(P) \wedge t \in traces(Q)) \} \\
&\qquad \vee\, (s \in traces(P) \wedge t \in divergences(Q))\} \\
divergences(P \underset{X}{||} Q) &= \{u^\frown v \mid \exists\, s : traces_\perp(P), t : traces_\perp(Q) \mid u \in (s \underset{X}{||} t) \cap \Sigma^* \wedge \\
&\qquad (s \in divergences(P) \vee t \in divergences(Q))\} \\
divergences(P \setminus X) &= \{(s \setminus X)^\frown t \mid s \in divergences(P)\} \quad \cup \\
&\qquad \{(u \setminus X)^\frown t \mid u \in \Sigma^w \wedge \neg(u \setminus X) : \Sigma^w \wedge \\
&\qquad (\forall\, s < u \mid s \in traces_\perp(P))\}
\end{aligned}
$$

Source: (ROSCOE, 2010)

The set of divergence for $P \setminus X$ is the union of the set of divergences of the process P with all sequences that can be performed by P, removing the infinite occurrences of the elements of X. Furthermore, all prefixes of the infinite traces of P, which were introduced after hiding the events of X, must belong to $traces_\perp(P)$.

Similar to the previous models, a process P is a refinement of a process Q (written as $Q \sqsubseteq_{\mathrm{FD}} P$) if, and only if: $failures_\perp(P) \subseteq failures_\perp(Q)$ and $divergences(P) \subseteq divergences(Q)$. These processes are failures-divergences equivalent ($P \equiv_{\mathrm{FD}} Q$) if, and only if, $Q \sqsubseteq_{\mathrm{FD}} P$ and $P \sqsubseteq_{\mathrm{FD}} Q$.

2.2 LTS AND WEAK BISIMULATION

In this section, we define the concepts of an LTS and Weak Bisimulation, which are essential to understand the protocol implementation of BRIC contracts described in Section 4.3.1.

2.2.1 LTS

The operational semantics of a CSP process is given by Labelled Transition Systems (LTS).

An LTS is a directed graph with a label on each edge representing what happens when we take the action which the edge represents. Most LTSs have a distinguished node q_0 that is the

one we are assumed the LTS to start from (ROSCOE, 1997).

Definition 1. *(LTS) A labelled transition system is a 4-tuple* $\langle Q, A, T, q_0 \rangle$ *where* Q *is a set of states;* A *is a set of labels;* T *is the transition relation, which satisfies* $T \subseteq Q \times (A \cup \{\tau\}) \times Q$*, with* $\tau \notin A$*; and* $q_0 \in Q$ *is the initial state.*

The FDR model checker interprets a process by expanding it into a finite LTS.

2.2.2 Weak Bisimulation

There are many bisimulation relations for process algebras that are used to characterise equivalences between nodes of LTSs and used to calculate the reduction of LTS states that represent equivalent processes (GORRIERI; VERSARI, 2015).

Two states in an LTS are bisimulation equivalent if they can simulate each other's transitions.

A weak bisimulation is a relation in which chains of τ actions are compressed into a singular τ, and chains of τ actions before and after a visible action α are absorbed into α, as explained in Definition 2 (GORRIERI; VERSARI, 2015).

Definition 2. *(Weak Bisimulation) For an LTS* $\langle Q, A \cup \{\tau\}, T, q_0 \rangle$*, where* $\tau \notin A$*, a weak bisimulation is a relation* $R \subseteq (Q \times Q)$ *such that if* $(q_1, q_2) \in R$ *then for all* $\alpha \in A$

- $\forall\, q_1'$ *such that* $q_1 \xrightarrow{\alpha} q_1'$*,* $\exists\, q_2'$ *such that* $q_2 \stackrel{\alpha}{\Rightarrow} q_2'$ *and* $(q_1', q_2') \in R$*,*
- $\forall\, q_2'$ *such that* $q_2 \xrightarrow{\alpha} q_2'$*,* $\exists\, q_1'$ *such that* $q_1 \stackrel{\alpha}{\Rightarrow} q_1'$ *and* $(q_1', q_2') \in R$*.*

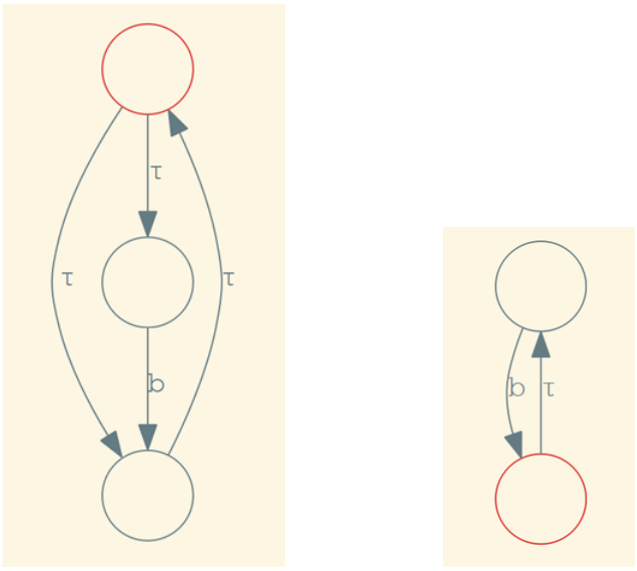
where if $\alpha \neq \tau$*, then* $s \stackrel{\alpha}{\Rightarrow} t$ *means that from* s *one can get to t by doing zero or more* τ *actions, followed by the action* α*, followed by zero or more* τ *actions. On the other hand, if* $\alpha = \tau$*, then* $s \stackrel{\alpha}{\Rightarrow} t$ *means that from s one can reach t by doing zero or more* τ *actions.*

As an example, consider the following CSP process:

$$P = a \rightarrow b \rightarrow a \rightarrow P \;\Box\; a \rightarrow Q$$
$$Q = a \rightarrow P$$
$$T = P \setminus \{a\}$$

Figure 3(a) represents the LTS of the process T that has chains of τ. Otherwise, Figure 3(b) shows its weak bisimulation equivalence with no chains of τ.

Figure 2 – Weak Bisimulation Equivalence



(a) Original LTS

(b) Weak Bisimulation Equivalent LTS

Source: Author's ownership.

2.3 THE BRIC COMPONENT MODEL

In this section, we describe an approach for creating and composing components, BRIC, used in this work. This approach is based on a set of rules that guarantees the absence of deadlock and livelock by construction (RAMOS; SAMPAIO; MOTA, 2009; Conserva Filho et al., 2018). Our focus here is on deadlock freedom. A concurrent system is deadlocked if no component can make any progress, generally because each one is waiting for communication with others.

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 (OLIVEIRA et al., 2016).

In some contexts, particularly when there is some criticality involved, a reliable architecture becomes a demand. The architecture 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.

The reason is that formal verification techniques such as model checking face the well-known state explosion problem. The number of global states of a concurrent system with

multiple processes can be enormous; it is exponential in both the number of processes and the number of components per process (CLARKE et al., 2012).

In order to avoid or minimize this problem, instead of verifying the entire system, other more promising approaches focus on iteratively identifying problems in compositions; BRIC is one of them.

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.

A component contract encapsulates a component in BRIC. It is defined in terms of the component behaviour (represented as a CSP process), its ports (represented as channels) and their respective types.

Definition 3. *(Component Contract) A component contract* $Ctr : \langle B, R, I, C \rangle$ *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 (where each type, interface, is also a set of values), such that* B *is an I/O process (see Definition 4).*

We use B_{ctr}, R_{ctr}, I_{ctr} and C_{ctr} to denote the elements of the contract Ctr: behaviour, relation among channels and interfaces, interfaces and channels, respectively. In our example, the contracts for fork and philosopher are defined as follows:

$$
\begin{aligned}
&FORK : \\
&\quad B_{FORK} = Fork(id), \\
&\quad R_{FORK} = \langle (fork\_right.id, \{picksup\_I, picksup\_O, putsdown\_I, putsdown\_O\}), \\
&\qquad\qquad (fork\_left.id, \{picksup\_I, picksup\_O, putsdown\_I, putsdown\_O\}) \rangle, \\
&\quad C_{FORK} = \{fork\_right.id, fork\_left.id\}, \\
&\quad I_{FORK} = \{picksup\_I, picksup\_O, putsdown\_I, putsdown\_O\}
\end{aligned}
$$

$PHIL$:

$B_{PHIL} = Phil(id)$,

$R_{PHIL} = \langle (phil\_right.id, \{picksup\_I, picksup\_O, putsdown\_I, putsdown\_O\}),$

$(phil\_left.id, \{picksup\_I, picksup\_O, putsdown\_I, putsdown\_O\})\rangle$,

$C_{PHIL} = \{phil\_right.id, phil\_left.id\}$,

$I_{PHIL} = \{picksup\_I, picksup\_O, putsdown\_I, putsdown\_O\}$

The behaviour of these components, given by $Fork(id)$ and $Phil(id)$, are represented by I/O processes.

Definition 4. *(I/O Process) An I/O process is a CSP process P that satisfies the following properties:*

- ***I/O channels*** *Every event in P is either an input or an output (but not both). A channel c is an I/O channel if, for a process P:*

$$inputs(c, P) \cup outputs(c, P) \subseteq \{|\ c\ |\} \wedge$$
$$inputs(c, P) \cap outputs(c, P) = \{\}$$

where $\{|\ c\ |\}$ yields the set of all events on channel c, and inputs(c,P) and outputs(c,P) yield all input and output events on c in process P, respectively.

Let P be a process that only uses I/O channels. The inputs and outputs of P are determined by the following, equally named, functions inputs(P) and outputs(P):

$$inputs(P) = \{c.e \mid c.e \in inputs(c, P)\} \wedge outputs(P) = \{c.e \mid c.e \in outputs(c, P)\}$$

In our example, the definitions of inputs and outputs are as follows.

$inputs(Fork) = \{fork\_right.id.picksup\_I, fork\_left.id.putsdown\_I\}$

$outputs(Fork) = \{fork\_right.id.picksup\_O, fork\_left.id.putsdown\_O\}$

$inputs(Phil) = \{phil\_right.id.picksup\_O, phil\_left.id.putsdown\_O\}$

$outputs(Phil) = \{phil\_right.id.picksup\_I, phil\_left.id.putsdown\_I\}$

No event is both input and output, in the same process. Note that the events of the channels in $inputs(Fork)$ and $outputs(Phil)$ communicate the same set of data. In order

to allow communication, outputs of one are observed as inputs of the other, and vice-versa. In general, inputs and outputs of a process divide the events of a channel c in two sets. However, the process is not obliged to perform all events of c. The directions of the events as inputs or outputs are not explicit in the channel definition but implicit in the process definitions through interfaces that define a component's provided and required services.

- ***Non-terminating*** *P is a non-terminating process but has a finite state space. The processes $Fork$ and $Phil$ satisfy this condition since they are defined as infinite loops.*
- ***Divergence free*** *P has no livelocks. $Fork$ and $Phil$ are divergence-free, that is, there is no infinite traces of internal events.*
- ***Input determinism*** *If a set of input events of P is offered by the environment, none of them is refused by P. The processes $Fork$ and $Phil$ are deterministic processes. Consequently, they are input deterministic processes.*
- ***Strong output decisive*** *All choices (if any) among output events on a given channel of P are internal. The process, however, must offer at least one output on that channel.*

Details about these properties are presented in (RAMOS, 2011).

Normally, a component is defined once and reused multiple times and in multiple different contexts. In BRIC, a context is represented as a set of channels since these channels represent interaction points of the component, and each channel is used to communicate with a single component in the environment. Then, replacing the channels of a component contract by another set means that it supposedly interacts with another environment. This replacement is represented by a bijection of the set of channels of the component contract into a set with new channels.

In the example of the $Fork$ component this bijection is realised by replacing the id parameter. Since the process $Fork$ is parametrised by its id, several instances for distinct identifiers can be created:

$$fork1 = Fork(1)$$
$$fork2 = Fork(2)$$

2.3.1 Communication Protocol

Protocols can represent the entire observable behaviour of the component, or the behaviour associated to an interaction point of the component; this observable behaviour is defined as a projection over a set of channels, see Definition 6.

Communication protocols (Definition 5) are commonly associated with specifications of component behaviours at a specific abstraction level, with an exclusive focus on a portion of the communicated events. They are used in local analyses of component interaction before their composition

Definition 5. *(Communication protocol). We say a CSP process P is a communication protocol if :*

$$\exists\, c_1, c_2 \bullet inputs(P) \subseteq \{|\ c_1\ |\} \land outputs(P) \subseteq \{|\ c_2\ |\}$$

Then a communication protocol is an I/O process that inputs by a unique channel (c_1, for instance) and outputs by a unique channel (c_2, for instance).

Definition 6. *(Projection). Let P be a process, and C a set of communication channels. The projection of P over C (denoted by $P \upharpoonright\ C$) is defined as:*

$$P \upharpoonright\ C = P \setminus (\Sigma \setminus C)$$

Projections restrict the behaviour of a process to a set of events. It behaves as the hiding of all events, except those in C. This restriction, however, might introduce divergence in the protocol implementation, which must be avoided.

Definition 7. *(Protocol implementation). Let P be an I/O process, and C a set of communication channels. The communication protocol, named $ProtIMP(P, C)$ and implemented by P over C, is a protocol that satisfies the following properties;*

$$ProtIMP(P, C) \sqsubseteq_{\mathrm{F}} P \upharpoonright\ C$$

and

$$P \upharpoonright\ C \sqsubseteq_{\mathrm{FD}} ProtIMP(P, C)$$

$ProtIMP(P, C)$ is a process that is related, via refinement, to $P \upharpoonright\ C$. However, the former cannot have divergences. When projecting a process into a set of channels, divergence might

be introduced, due to the use of the hiding operator. Thus, $P \upharpoonright C$ may diverge. In place of the divergences of $P \upharpoonright C$, $ProtIMP(P, C)$ is allowed to exhibit stable failures, as explained in the stable-failures model. Hence, $ProtIMP(P, C)$ may have more failures than $P \upharpoonright C$ (first refinement statement of Definition 7). Moreover, as $ProtIMP(P, C)$ cannot diverge, then, it has less or equal divergences than $P \upharpoonright C$, and the same set concerning $failures_{\perp}$ (unstable failures). That is the reason we need the second refinement of Definition 7.

One of the important contributions of this thesis is that, rather than asking the user to propose a valid protocol implementation from a projection over a set of channels, we compositionally derive $ProtIMP(P, C)$ from $P \upharpoonright C$. This is detailed in Section 4.3.1.

To illustrate why simply projecting a process over a set of channels can lead to divergent behaviour, consider a protocol implementation for the $fork1$ process over the channel $fork\_right.1$ given by:

$$fork1 \upharpoonright \{fork\_right.1\} = fork1 \setminus (\Sigma \setminus \{| fork\_right.1 |\})$$

By expanding this process, we have the following result, where all events are hidden, except the ones related to channel $fork\_right.1$.

$$fork1 = FORK(1) = STM\_FORK(1)$$

$$STM\_FORK(1) = Available(1)$$

$$Available(1) = (fork\_right.1.picksup\_I \rightarrow fork\_right.1.picksup\_O \rightarrow Busy1(1))$$

$$\Box$$

$$Busy2(1)$$

$$Busy1(1) = (fork\_right.1.putsdown\_I \rightarrow fork\_right.1.putsdown\_O \rightarrow Available(1))$$

$$Busy2(1) = Available(1)$$

In this case, the projection is directly obtained by hiding the relevant events, but the resulting process has the following livelock sequence: $STM\_FORK(1)$; $Available(1)$; $Busy2(1)$; $Available(1)$. The attempt to conceal an infinite sequence of consecutive events leads to the same result as an infinite loop or *unguarded recursion*, divergence (HOARE, 1985). Thus, this process cannot be used as a valid protocol implementation.

In the same way if we project $fork1$ over the channel $fork\_left.1$, this also results in a divergent behaviour. However, by excluding such divergence sequences from both projections,

results in the process $PROT\_FORK(ch)$ that represents the protocol related to each channel ch from component $FORK$, and it is divergence free. This process is given by:

$$PROT\_FORK(ch) = ch.picksup\_I \rightarrow ch.picksup\_O \rightarrow ch.putsdown\_I \rightarrow$$
$$ch.putsdown\_O \rightarrow PROT\_FORK(ch)$$

In general, however, it might not be easy to construct a valid protocol manually. As presented in Section 4.3, we conceived an automated strategy to generate valid protocol implementations.

In a composition, protocol implementations of components have to be strongly compatible. Before formalising the verification of this condition, we define an auxiliary notion: the dual protocol of *P*, *DualProt(P)*, is a protocol with the same traces of *P*, but whose inputs are the outputs of *P*, and vice-versa.

Definition 8. *(Dual Protocol) Let P be a deadlock-free communication protocol. The dual protocol of P is defined as a deadlock-free communication protocol DP, such that:*

$$inputs(P) = outputs(DP) \wedge outputs(P) = inputs(DP) \wedge traces(DP) = traces(P)$$

The formal verification of *Strong Compatibility* is characterised by assertions on simple failures refinement: two protocols *P* and *Q* are strongly compatible if $DualProt(P) \sqsubseteq_{\mathrm{F}} Q$ or if $DualProt(P) \sqsubseteq_{\mathrm{F}} Q \parallel CTX(P)$. The context protocol of *P*, Definition 9, $CTX(P)$, represents its possible communications, and is formally defined by a deadlock-free deterministic process with the same traces as those of P.

Definition 9. *(Communication context process). Let P be a deadlock-free communication protocol. The communication context process of P (denoted by CTX_p) is defined as a deadlock free deterministic process, such that $traces(CTX_p) = traces(P)$.*

Similarly to the definition of protocol implementation, there is a communication context process associated to a specific channel.

Definition 10. *(Communication context process implementation). Let P be a communication protocol. The communication context process of P is named $CTX(P)$.*

The context protocol of *P*, $CTX(P)$, represents its possible communications, and is formally defined by a deadlock-free deterministic process with the same traces as those of P.

As an example, consider the verification of a dual protocol of $PROT\_FORK$ in the channel $fork\_left$:

$$DualProt(PROT\_FORK(fork\_left))$$
$$\sqsubseteq_{\mathrm{F}}$$
$$PROT\_PHIL(phil\_right) \underset{\{|fork\_left, phil\_right|\}}{||} CTX(PROT\_FORK(fork\_left))$$

Then, it is possible to verify the compatibility of two protocols $PROT\_FORK$ and $PROT\_PHIL$ by assuring that dual protocol of $PROT\_FORK$ is refined by protocol of $phil$ ($PROT\_PHIL$) in parallel with the context process process of $PROT\_FORK$.

2.3.2 Compositional Development in BRIC

Based on the work presented in (ROSCOE, 2005) on buffer tolerance, BRIC adopts an asynchronous communication model in component interactions. To represent asynchronous communication, buffers are introduced as intermediate elements of the composition. They copy information from one component channel to another, preserving order and without loss. Information is always accepted, independent of the other component being ready or not to input. It also never refuses outputs when it is non-empty. These buffers are not first-class elements, but are implicit to the component model. Buffers are considered infinite.

Component compositions are defined in two modes: a binary operation on two components, and a unary operation over a single component.

In order to specify an asynchronous binary composition in BRIC, we first present an auxiliary function $AsyncComp$ that takes a set of processes S and a bijective function F among distinct sets of channels used by processes within S and yields the assembly of the processes within S, connecting each channel c to its respective channel represented by $F(c)$.

$$AsyncComp(S, F) = (\Big|\Big|\Big|_{P \in S} P) \; ||_{dom\, F \cup ran\, F} \; (|||_{c \in dom\, F} \; BUFF^{\infty}_{IO}(c, F(c)))$$

The function $AsyncComp$ runs the processes from S in interleaving. They are put in parallel with $BUFF^{\infty}_{IO}$ forcing them to synchronise on events of the set $dom\, F \cup ran\, F$. The process $BUFF^{\infty}_{IO}(c, z)$ is an infinite buffer that copies information from c to z, and vice-versa.

Definition 11. *(Asynchronous Binary Composition). Let Ctr_1 and Ctr_2 be two distinct component contracts, and $\langle ic_1, ..., ic_n \rangle$ and $\langle oc_1, ..., oc_n \rangle$ sequences of distinct channels within*

C_{Ctr_1} *and* C_{Ctr_2} [3] *, respectively, with* $C_{Ctr_1} \cap C_{Ctr_2} = \emptyset$*. Then, the asynchronous binary composition of* Ctr_1 *and* Ctr_2 *(namely* $Ctr1_{ic} \asymp_{oc} Ctr_2$*) is given by:*

$$Ctr_{1\langle ic_1,...,ic_n\rangle} \asymp_{\langle oc_1,...,oc_n\rangle} Ctr_2 = \\ (\langle AsyncComp(\{B_{Ctr_1}, B_{Ctr_2}\}, \{ic_i \mapsto oc_i \mid i \in 1..n\}), R_{Ctr_3}, I_{Ctr_3}, C_{Ctr_3}\rangle)$$

where $C_{Ctr_3} = (C_{Ctr_1} \cup C_{Ctr_2} \setminus \{ic_1, ..., ic_n, oc_1, ..., oc_n\}$*,* $R_{Ctr_3} = C_{Ctr_3} \lhd (R_{Ctr_1} \cup R_{Ctr_2})$*, and* $I_{Ctr_3} = ran\, R_{Ctr_3}$*.*

In the definition above, each component has a distinct set of interaction points and their communication is asynchronous, mediated by buffers. The behaviour of the composition is defined by the synchronisation of the components (Ctr_1 and Ctr_2) with a buffer. The communications used in the composition are not offered to the environment in further compositions (C_{Ctr_3}). The operator $\lhd$ stands for domain restriction. It is used to restrict the mapping from channels to interfaces (R_{Ctr_3}) and, furthermore, to restrict the set of interfaces in the composition contract (I_{Ctr_3}).

The other composition mode concerns the assembling of channels of the same component. Unary compositions $Ctr \asymp|_t^s$ are needed when we want to assemble inner channels from two channel lists s and t of a single component Ctr.

Definition 12. *(Asynchronous Unary Composition). Let* Ctr *be a component contract, and* $\langle ic_1, ..., ic_n\rangle$ *and* $\langle oc_1, ..., oc_n\rangle$ *sequences of distinct channels within* C_{Ctr}*, such that* $\{ic_1, ..., ic_n\} \cap \{oc_1, ..., oc_n\} = \emptyset$*. The asynchronous unary composition of* Ctr*, namely* $Ctr \asymp|_{oc}^{ic}$*, is given by:*

$$Ctr \asymp|_{\langle oc_1,...,oc_n\rangle}^{\langle ic_1,...,ic_n\rangle} = (\langle AsyncComp(B_{Ctr}, \{oc_i \mapsto ic_i \mid i \in 1...n\}), R_{Ctr'}, I_{Ctr'}, C_{Ctr'}\rangle)$$

where $C_{Ctr'} = (C_{Ctr} \setminus \{\langle oc_1, ..., oc_n, ic_1, ..., ic_n\rangle\})$*,* $R_{Ctr'} = C_{Ctr'} \lhd R_{Ctr}$ *and* $I_{Ctr'} = ran\, R_{Ctr'}$*.*

It is similar to the definition of binary asynchronous composition. However, the definition above allows us to assemble channels of the same component, instead of two distinct components.

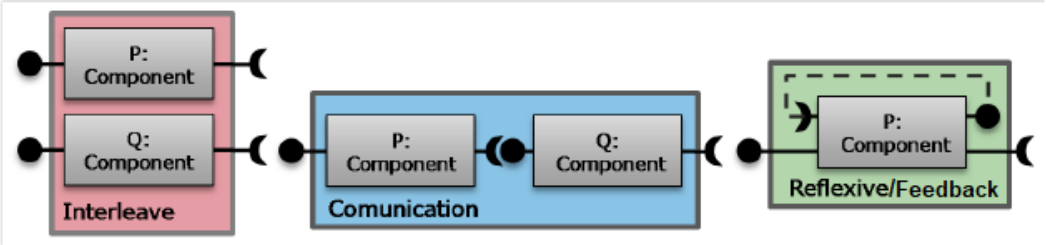
2.3.3 Composition Rules

The constructive design of a BRIC component architectural model is based on composition rules for components. These rules present a systematic strategy to build systems when compo-

[3] The alphabet of events of a component Ctr is given by C_{Ctr}.

nents are able to communicate. These composition rules guarantee composition properties of a system by construction, based on the same properties already established for the constituent components, so that problems are anticipated before all parts are integrated. Our focus here is on the preservation of deadlock freedom. BRIC provides four composition rules: interleaving, communication, feedback and reflexive compositions. Each of these compositions constructs a new component, which includes the original ones.

Figure 3 – Composition Rules



Source: (RAMOS, 2011)

Each rule (see Figure 3) has well-defined side conditions that ensure a sound composition (RAMOS; SAMPAIO; MOTA, 2009). The first composition rule is interleaving, which aggregates two independent components that do not communicate with each other; the components do not share any channels, so no synchronisation is performed. The second rule is based on the traditional way to compose two components by connecting two channels, one from each component. The other two rules provide unary compositions: feedback and reflexive, which enable building systems with cyclic topologies, connecting two channels of the same component. Feedback composition represents the simpler unary composition, where two channels of the same component are assembled but do not introduce a new cycle of ungranted requests, which is a circular dependency among the channels of the component (more details can be seen in (ROSCOE, 1997)). Reflexive composition deals with more complex systems that indeed present cycles of dependencies in the system topology.

Next we explore more about each composition rule.

Definition 13. *(Interleave composition). Let* Ctr_1 *and* Ctr_2 *be two component contracts, such that* Ctr_1 *and* Ctr_2 *have disjoint channels. Then, the interleave composition of* Ctr_1 *and* Ctr_2*, namely* $(Ctr_1[||||]Ctr_2)$ *is given by:*

$$Ctr_1[||||]Ctr_2 = Ctr_{1\langle\rangle} \asymp_{\langle\rangle} Ctr_2$$

This rule uses the binary composition operator, $\asymp$, which provides an asynchronous interaction. This is a particular kind of direct composition that involves no communication, resulting

in a weakly cohesive entity, which performs all events defined in the original entities without any interference from each other. An interleave composition of two deadlock-free component contracts is also a deadlock-free component contract.

For instance, we can build interleave compositions of fork and philosopher contracts. Let Ctr_{fork1}, Ctr_{fork2} and Ctr_{phil1}, Ctr_{phil2} be fork and philosopher contracts, respectively. The interleave composition of fork contracts is given by:

$$Ctr_{fork1\_fork2} = Ctr_{fork1}\,[||||]\,Ctr_{fork2} = Ctr_{fork1\langle\rangle} \asymp_{\langle\rangle} Ctr_{fork2}$$

In the same way, the interleave composition of philosopher contracts is given by:

$$Ctr_{phil1\_phil2} = Ctr_{phil1}\,[||||]\,Ctr_{phil2} = Ctr_{phil1\langle\rangle} \asymp_{\langle\rangle} Ctr_{phil2}$$

The interleave composition of all contracts is given by:

$$Ctr_{fork1\_fork2\_phil1\_phil2} = Ctr_{fork1\_fork2}\,[||||]\,Ctr_{phil1\_phil2} = Ctr_{fork1\_fork2\langle\rangle} \asymp_{\langle\rangle} Ctr_{phil1\_phil2}$$

Definition 14. *(Communication composition). Let Ctr_1 and Ctr_2 be two component contracts, and ic and oc two communication channels such that $ic \in C_{Ctr_1} \wedge oc \in C_{Ctr_2}$ and $ProtIMP(Ctr_1, ic)$ and $ProtIMP(Ctr_2, oc)$ are strongly compatible, where C_{Ctr_1} and C_{Ctr_2} are the set of channels from Ctr_1 and Ctr_2, respectively. Then, the communication composition of Ctr_1 and Ctr_2 namely $Ctr_1[ic \leftrightarrow oc]Ctr_2$, via ic and oc is defined as follows:*

$$Ctr_1[ic \leftrightarrow oc]Ctr_2 = Ctr_{1\langle ic\rangle} \asymp_{\langle oc\rangle} Ctr_2$$

The communication composition of two deadlock-free component contracts is also a deadlock free component contract; according to theorem *Deadlock-free communication composition* described in (RAMOS, 2011)

For instance, we can build compositions of fork and philosopher contracts. Let Ctr_{fork1} be a fork contract and Ctr_{phil1} be a philosopher contract, where the communication occurs via the channels $fork\_left.1$ and $phil\_right.1$. The communication composition of these contracts is given by:

$$Ctr_{comm\_fork1\_phil1} = Ctr_{fork1}[fork\_left.1 \leftrightarrow phil\_right.1]\,Ctr_{phil1} = Ctr_{fork1\langle fork\_left.1\rangle} \asymp_{\langle phil\_right.1\rangle} Ctr_{phil1}$$

The next two composition rules allow assembling two channels of the same component. The third composition rule (which we call feedback composition) deals with pseudo-cyclic

topologies, which are behaviourally equivalent to systems with tree-topologies. It does have some cycles, but none of them introduces deadlocks. However, it cannot express all possible topologies. For this reason, verification on this topology is simpler than in arbitrary complex topologies. The feedback composition is aligned with the incremental nature of our strategy, dealing with one problem at a time. The feedback composition rule requires the two linked channels to be decoupled.

Definition 15. *(Decoupled channels). Two channels of a process are decoupled if communication on one channel does not interfere with communications of the other. For this reason, the communications through the two behave as communications between channels of distinct processes. Formally, the channels within cs are decoupled in P (denoted as cs Decoupled In P) if, and only, if:*

$$P \upharpoonright\ cs \equiv_{\mathrm{F}} \Big|\Big|\Big|_{c \in cs} ProtIMP(P, c)$$

A channel $c1$ is independent of (or decoupled from) a channel $c2$ in a process when any communication of $c2$ does not interfere with the order of events communicated by $c1$, and vice-versa. It means that they are offered to the environment independently. So, a communication between two channels of a same process behaves as communications between channels of distinct processes.

Definition 16. *(Feedback composition). Let Ctr be a component contract, and ic and oc two communication channels, such that $\{ic, oc\} \subseteq C_{Ctr}$ are independent (decoupled) in Ctr, $ProtIMP(Ctr, ic)$ and $ProtIMP(Ctr, oc)$ are strong compatible. Then, the Feedback composition of Ctr, namely $Ctr[oc \hookrightarrow ic]$, hooking oc to ic is defined as follows:*

$$Ctr[oc \hookrightarrow ic] = Ctr \asymp|_{oc}^{ic}$$

When we connect two interfaces of the contract $Ctr_{fork1\_fork2\_phil1\_phil2}$, for instance $fork\_left.1$ and $phil\_right.1$ channels, it is considered a feedback composition, which generates a new component whose contract is $Ctr_{FEED\_fork1\_fork2\_phil1\_phil2}$:

$$Ctr_{FEED\_fork1\_fork2\_phil1\_phil2} = Ctr_{fork1\_fork2\_phil1\_phil2}[fork\_left.1 \hookrightarrow phil\_right.1] = Ctr_{fork1\_fork2\_phil1\_phil2} \asymp|_{fork\_left.1}^{phil\_right.1}$$

The Reflexive composition rule deals with more complex systems that indeed present cycles of dependencies in the topology of the system structure. This rule connects dependent

channels, which may introduce undesirable cycles of dependencies among the communication of events in the system. In order to make a composition using the Reflexive rule, the buffering self injection property must be established.

Definition 17. *(Buffering self-injection compatibility). Let P be a deadlock-free I/O process, and c and z channels. Then* $P_j = P \upharpoonright \{|\ c, z\ |\}$ *is buffering self-injection compatible if, and only if:*

1. $\forall (s, X) : failures(P_j) \mid (s \downarrow O_c = s \downarrow I_z) \wedge (s \downarrow O_z = s \downarrow I_c) \bullet X \cap (O_c \cup O_z) = \emptyset$
2. $\forall (s, X) : failures(P_j) \mid (s \downarrow O_c > s \downarrow I_z \bullet (s \upharpoonright z, X \cup \{|\ c\ |\}) \in failures(P_j \upharpoonright z)$
3. $\forall (s, X) : failures(P_j) \mid (s \downarrow O_z > s \downarrow I_c \bullet (s \upharpoonright c, X \cup \{|\ z\ |\}) \in failures(P_j \upharpoonright c)$

where $O_c = outputs(P, c)$, $O_z = outputs(P, z)$, $I_c = inputs(P, c)$ *and* $I_z = inputs(P, z)$ *and* $s \downarrow X$ *returns the cardinality of events within* X *in the trace* s.

Buffering self-injection compatibility is very similar to the notion of strong compatibility, except for the fact that it does not compare the communication between two simple processes (protocols) but between events of the same process, the behaviour of the contract. All the outputs produced in s or z are consumed by the inputs of the process in a way that none of the outputs can be refused.

Definition 18. *(Reflexive composition) Let* Ctr *be a component contract, and* ic *and* oc *two channels, such that:* $\{ic, oc\} \subseteq C_{Ctr}$ *and* $B_{ctr} \upharpoonright \{|\ ic, oc\ |\}$ *is buffering self-injection compatible, then, the Reflexive composition is defined as:*

$$Ctr[ic \overset{-}{\hookrightarrow} oc] = Ctr \asymp|_{oc}^{ic}$$

The structure of a reflexive composition is similar to a feedback composition. However, it does not impose the restriction that the channels to be connected are decoupled; on the other hand, it requires a deadlock analysis by checking if the process restricted to the channels involved in the composition is buffering self-injection compatible.

The presented composition rules comprise a systematic method to preserve behavioural properties by construction in component compositions, focusing on the preservation of deadlock.

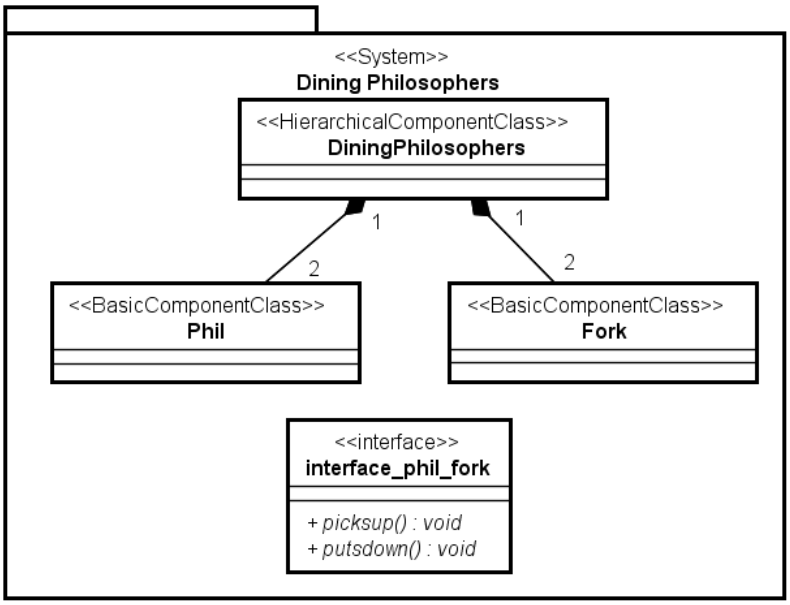
2.4 UML

Our aim here is to foster a formal CBSD model for UML (Unified Model Language) (Object Management Group (OMG), 2016), motivated by the fact that UML is a widely used notation in industry, and amenable to mechanised analysis. We have chosen some elements from UML to describe the behaviour of a component and how it can communicate with other ones. In this section we briefly describe the diagrams that are used in our component metamodel: Class, State Machine and Composite Structure Diagrams.

2.4.1 Class Diagram

A class diagram represents the static view that shows a collection of declarative (static) model elements, such as classes, types, and their contents and relationships. Figure 4 represents the component $Fork$ in UML. Here we use a stereotype to characterize a component; a stereotype extension mechanism defines a new kind of model element based on an existing model element. Figure 4 shows an example of a class diagram that describes a Dining Philosophers component. A *diningPhilosophers* class is composed of *Phil* and *Fork* classes. Multiplicity specifies the number of instances of the component; in this case, there are two instances of each *Phil* and Fork component that realise the same interface: *interface_phil_fork*.

Figure 4 – Class diagram of Dining Philosophers component.



Source: Author's ownership.

2.4.2 State Machine Diagram

A state machine diagram is used to model the dynamic aspects of a system, emphasizing the flow of control from state to state, specifying the sequences of states an object goes through during its lifetime in response to events. A state is a result of previous activities performed by the object and is typically determined by the values of its attributes and links to other objects. A transition is a relationship between two states. It has five elements: source state, trigger, guard condition, action and target state. Except for the source and target states, the other elements are optional. A source state is a state from which a transition is trigerred. An event trigger is a stimulus that can trigger a source state to fire on satisfying guard conditions. Guard conditions are boolean expressions evaluated that affect the behaviour of a state machine by allowing the transition if evaluated to true and disabling it if evaluated to false. In the UML notation, guard conditions are defined in square brackets ([size $==$ 0]). An action is an executable atomic computation that may directly act on the object that owns the state machine and indirectly on other objects visible to the object. A target state is a state that is active after the completion of the transition.

Figure 5 shows the state machine which represents the reactive behaviour of the $Fork$ component. It has three states: $available$, $Busy1$ and $Busy2$. It cyclically offers the possibility of picking up the fork through its left ($fork\_left.picksup$) or right ports ($fork\_right.picksup$), and then waits for the fork to be put down via the same port: $fork\_right.putsdown$ and $fork\_left.putsdown$.

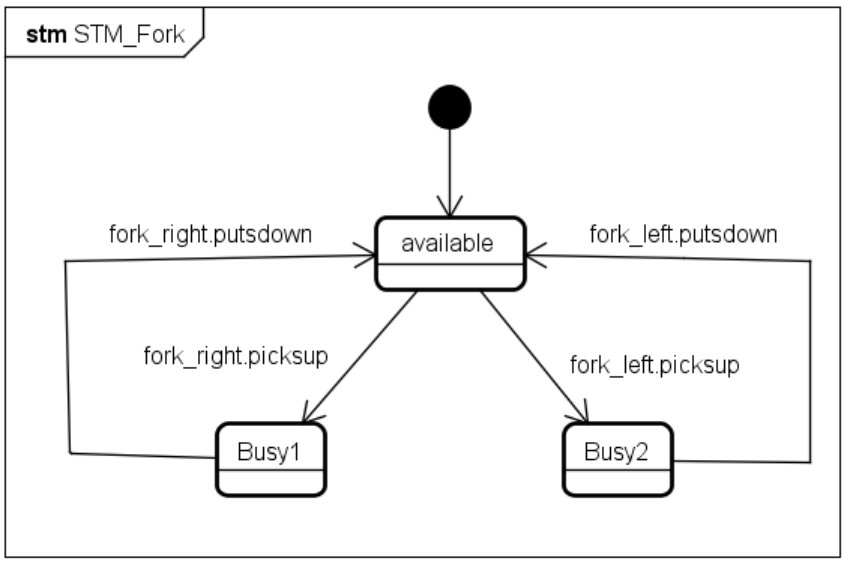
2.4.3 Sequence diagram

In UML, there are four types of diagrams to describe interactions: sequence diagrams, communication diagrams, interaction overview diagrams, and timing diagrams. Among them, the sequence diagram is the most commonly used to describe interaction of system participants.

A sequence diagram describes operational scenarios of a system with an emphasis on order. This is achieved through the use of lifelines. Each participant of the diagram, typically, instances of classes, possesses a lifeline, so that it can represent a message-exchange order.

The sequence diagram in Figure 6 presents a scenario of the dining philosophers example where two philosophers and two forks communicate. A lifeline is represented by a dashed vertical line under each participant.

Figure 5 – State Machine Diagram for the FORK Component.



Source: Author's ownership.

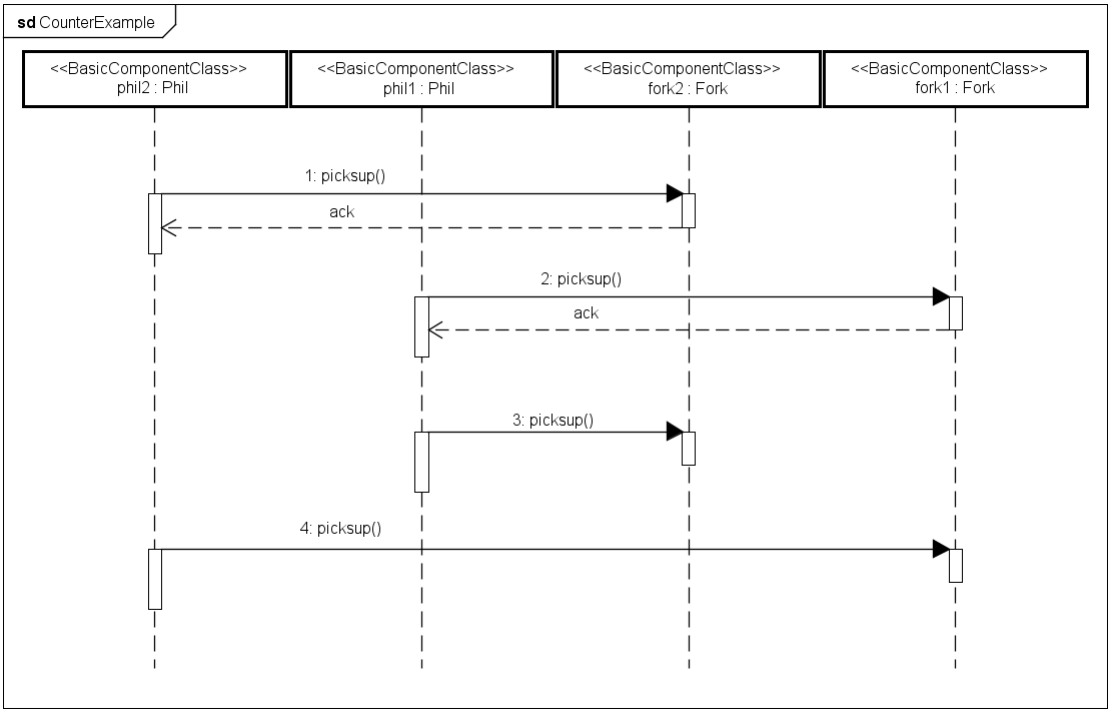
Figure 6 shows the following participants: $phil1$ and $phil2$ which are instances of $Phil$; $fork1$ and $fork2$ instances of $Fork$. Participants communicate via messages. For example, $phil2$ sends message $picksup$ to $fork2$. Messages are sent in sequence along a participant lifeline. So, the first message sent by the $phil2$ goes to $fork2$, the second from $phil1$ to $fork1$, the third from $phil1$ to fork2 and the fourth from $phil2$ to $fork1$.

Messages can be of three types: asynchronous (open arrowhead), synchronous call (closed arrowhead), or reply to a synchronous call (dashed arrow). Each message shown in Figure 6 is either synchronous or reply.

2.4.4 Composite Structure Diagram

A composite structure diagram is a type of static structure diagram that shows the internal structure of a class and the collaborations that this structure makes possible. This diagram can include internal parts, ports through which the parts interact with each other or through which instances of the class interact with the parts and with the outside world, and connectors between parts or ports. A composite structure is a set of interconnected elements that collaborate at runtime to achieve some purpose. Each element has some defined role in the collaboration. In Figure 7 the composite structure diagram shows how a dining philosophers could be composed; the parts of the diagram represent forks and philosophers that can communicate, connected through the ports ($fork\_rigth$, $fork\_left$, $phil\_rigth$, $phil\_left$); each port

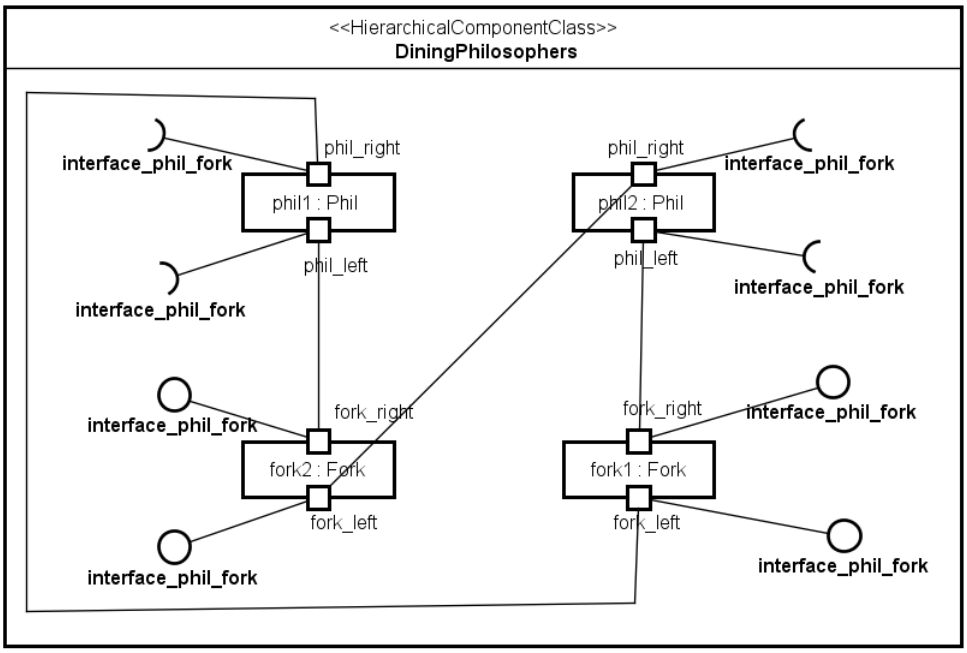
Figure 6 – Sequence Diagram



Source: Author's ownership.

provides or requires one interface, in this case $interface\_phil\_fork$.

Figure 7 – UML Composite Structure Diagram.



Source: Author's ownership.

3 PROPOSED UML COMPONENT MODEL

Although BRIC provides a sound and systematic component 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; this is followed by Section 3.2 that establishes the relevant well-formedness conditions. Then, in Section 3.3, we present the approach to create and compose component instances.

3.1 COMPONENT METAMODEL

Component models define specific representation, interaction, composition, and other standards for software components (HEINEMAN; COUNCILL, 2001). Component models can be defined for different levels of component abstractions. They can be very general, but they need to define at least the (LAU; WANG, 2005):

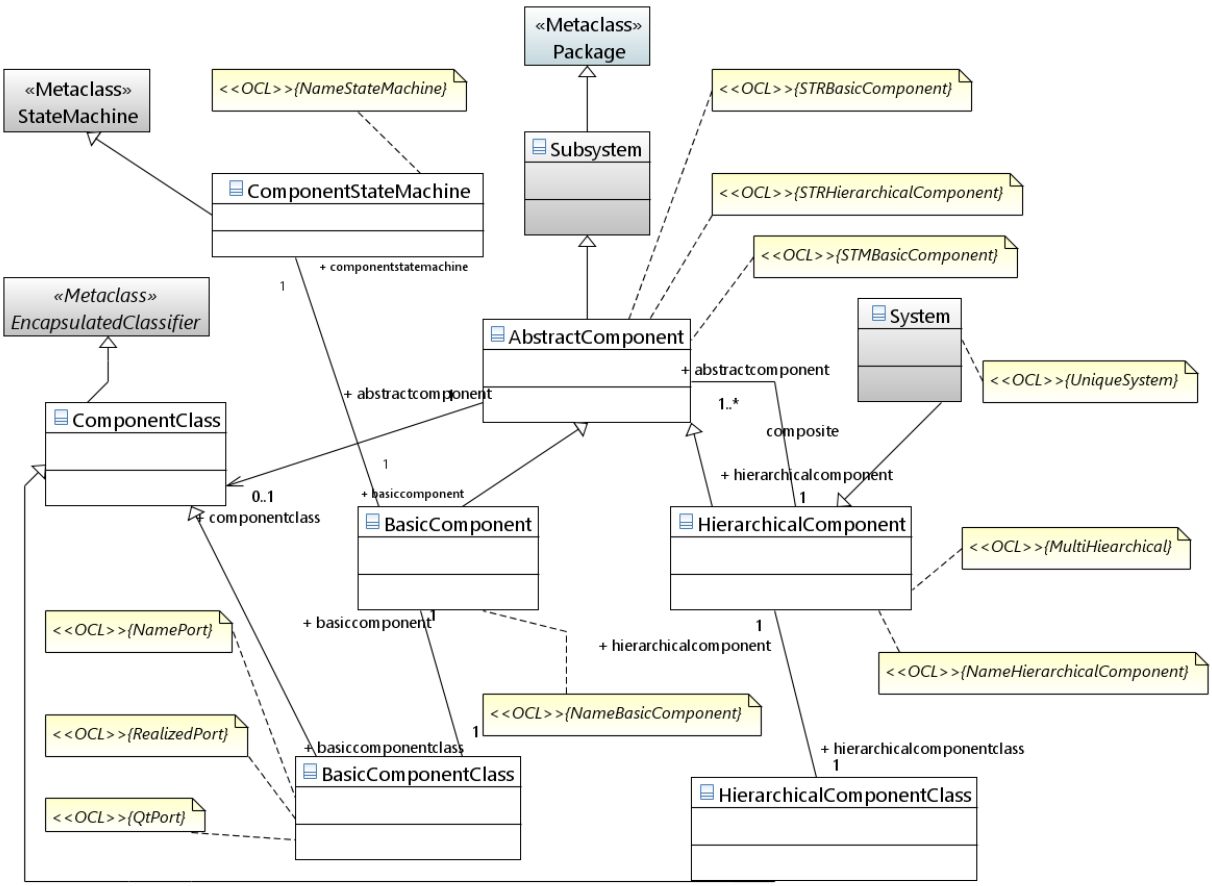
- *Syntax* of components, how they are constructed and represented;
- *Semantics* of components, what components are meant to be;
- *Composition* of components, how they are composed or assembled.

Although UML has a metamodel for components, this is normally used as a way to represent concrete artefacts, typically component implementations. We propose a component metamodel at the design phase, which is closer to the notion of a subsystem in UML, but we define the necessary elements to form a detailed component model, including structural and behavioural aspects, as well as composition rules to produce more elaborate components from basic ones.

In Figure 8, 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 on standard UML design, and are explained in the sequel.

We define a component as an *AbstractComponent*, which inherits from a UML *Subsystem*. A component must be either a *BasicComponent* or a *HierarchicalComponent*. A *BasicComponent* has one *BasicComponentClass* that describes the behaviour of the component, variables, constants and its ports. A *BasicComponentClass* is a UML *EncapsulatedClassifier* element

Figure 8 – The Component Metamodel



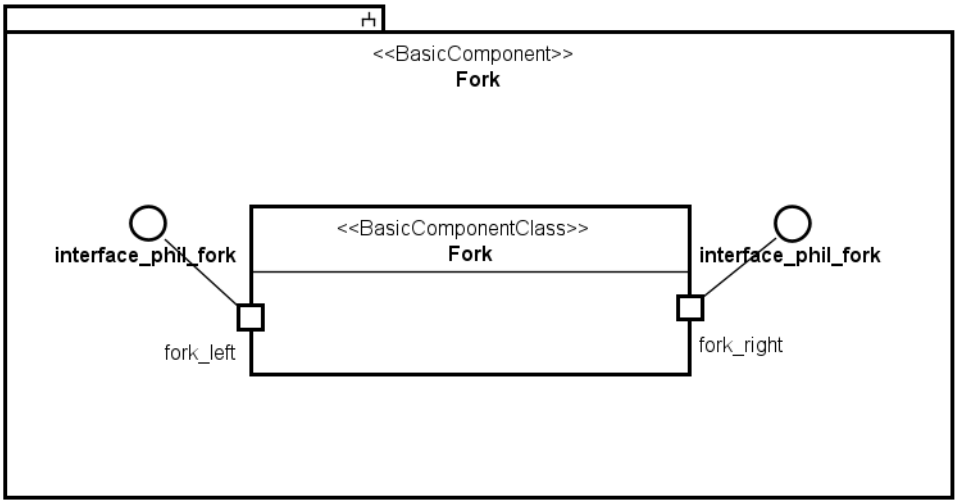
Source: Author's ownership.

that is represented by a *ComponentClass*, which, apart from attributes and operations, includes ports. It is modelled in a composite structure diagram that shows the internal structure of a class and the collaborations that this structure makes possible. The *BasicComponent* is the core class of a component metamodel. Its behaviour is defined by a state machine. In this work, it suffices considering the basic constructors of a state machine: initial pseudostate, choice pseudostate, final pseudostate, simple state and behavioural transition with triggers, guards and actions.

In the Dining Philosophers, *Fork* is an example of a *BasicComponent*. In Figure 9, it is defined as a *Subsystem* stereotyped *BasicComponent*. It has a *BasicComponentClass* with two ports, *fork_right* and *fork_left*, both realising the *interface_phil_fork* interface. Also, each *BasicComponent* has one state machine whose name is formed of the prefix *STM_* and the component's name.

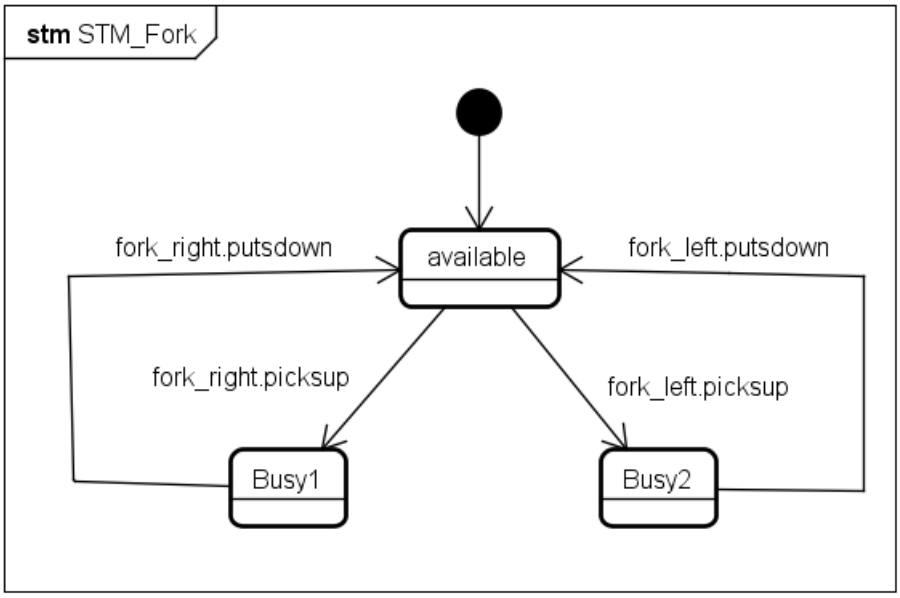
Figure 10 shows the state machine STM_Fork, which captures the reactive behaviour of the $Fork$ component. It cyclically offers the possibility of picking up the fork through its left

Figure 9 – Fork Component



Source: Author's ownership.

Figure 10 – State Machine STM_FORK

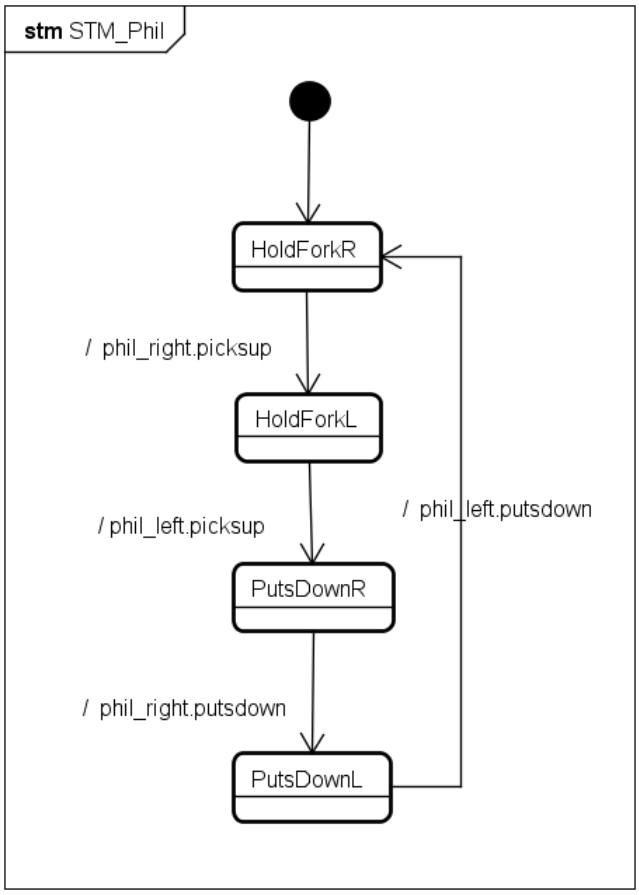


Source: Author's ownership.

or right ports and then waits for the fork to be put down via the same port. Figure 11 shows the behaviour of the *Phil* component given by *STM_Phil*.

A *HierarchicalComponent* is defined by the composition of component instances. This component must have a *HierarchicalComponentClass*, which owns a collection of other component classes; this is a composition relationship between the hierarchical component class and the classes of the other components. The connections between them should be expressed in the a *HierarchicalComponentClass* that is a UML *EncapsulatedClassifier* element, hence, it

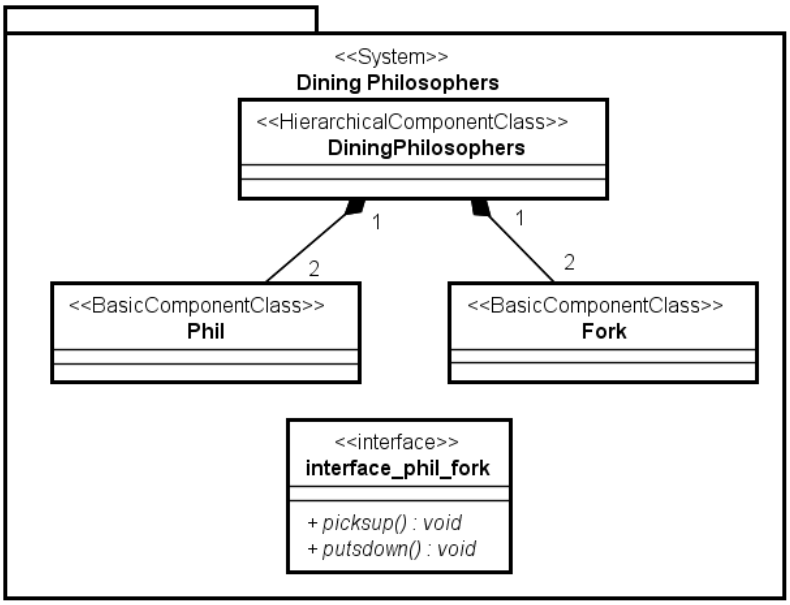
Figure 11 – PHIL State Machine



Source: Author's ownership.

may have ports to interact with other components. Finally, a *System* is a specialisation of a *HierarchicalComponent*, and it can be seen as the root component of the entire system.

Figure 12 – Hierchical Component



Source: Author's ownership.

The Dining Philosophers is modelled as a *System* element and, therefore, as a *HierarchicalComponent*; see Figure 12. It has a *HierarchicalComponentClass* that is related to one or two *Fork* and one or two *Phil* components, using a composition relationship. The ports from Phil and Fork components realize the interface *interface_phil_fork*. This interface defines the operations picksup and putsdown.

3.2 WELL-FORMEDNESS CONDITIONS

In addition to the metamodel, we need to define some well-formedness conditions to characterise meaningful models that can be assigned a formal semantics. Furthermore, a precise characterisation of a meaningful model can be seen as a modelling style to guide practitioners during the design. In Figure 8, these conditions are formally specified in OCL (Object Constraint Language) (Object Management Group (OMG), 2014); their titles appear inside model elements that represent notes, and their definitions are available in Appendix A. The informal explanation is as follows:

Basic Component. This kind of component has one stereotyped class *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*, which can itself have

attributes. In OCL, this is captured by the Constraint 1 that is about the *BasicComponentClass* context where the invariant (inv) *qtPortBC* determines that the number of ports is at least one. In an OCL expression, the reserved word $self$ is used to refer to the contextual instance. In this case, $self$ refers to an instance of *BasicComponentClass*. The expression $OwnedPort$ refers to a set of ports that *BasicComponentClass* owns, and the arrow ($\rightarrow$) is used to acess the size property on set. The invariant *namedPort* determines that the ports must have different names. All ports of the component must realise a provided or required interface to conform to the constraint *realizedPort*. The $forAll$ operation allows specifying a boolean expression, which must hold for all objects in a collection. And the $required()$ and $provided()$ operations refer to a set of interfaces required and provided, respectively, by the type of the port.

Constraint 1 – Basic Component Class

```
context BasicComponentClass

inv qtPortBC:
 self.ownedPort->size()>=1

inv namedPort:
 self.ownedPort->forAll(c1,c2 | c1<>c2
  implies c1.name <> c2.name and
       c1.name <> '' and c2.name <> '')

inv realizedPort:
 self.ownedPort->forAll(c1|
  c1.required()->size()>0 or c1.provided()->size()>0 )

inv conjugatedPort:
  self.ownedPort-> forAll(c1 | c1.isConjugated)

inv interfaceOperations:
  self.ownedPort ->
   forAll (c1| c1.required().ownedOperation ->size()>0  or
       c1.provided().ownedOperation ->size()>0)
```

Source: Author's ownership.

Hierarchical Component. This kind of component has one stereotyped class *HierarchicalComponentClass*. Similar to the *BasicComponentClass*, the name of the *HierarchicalComponentClass* should be the same as the one for the component, conforming to the constraint *NameHierarchicalComponent*. Also, a *HierarchicalComponentClass* may have attributes. This class must be the owner class of a composition relationship with other component classes to

express the ownership of other components. The structure of a *HierarchicalComponentClass* is described by a composite structure diagram where the connections among the owned component instances are specified. A Hierarchical Component must own at least one Basic or Hierarchical component; this is captured by the constraint *MultiHierarchical*.

System element. There must be exactly one *System*, which is the root component. This is a special type of *HierarchicalComponent*. The singleness is determined by the OCL Constraint 2 with the invariant *UniqueSystem* in System context.

Constraint 2 – System

```
1  context System
     inv UniqueSystem:self->size()=1
```

Source: Author's ownership.

Component Instance It is an individual element with its own internal state. Each component instance must be bounded to a type: Basic Component or Hierarchical Component. Component instances are represented by the UML *part* element in the *EncapsuledClassifier* of a hierarchical component. Figure 13 illustrates two instances of the $Fork$ component and two of the $Phil$ component.

Multiplicities. Multiplicities with the * character are not allowed in the composite structure diagram because we are dealing with a bounded number of instances. This is important to make the formal analysis feasible. All parts in a composition relationship must appear in the associated composite structure diagram in numbers compatible with their multiplicities. This is captured by the constraint *multiplicityLimited*, Appendix B.

Ports. A port allows communication between component instances. Each port must realise one interface; components do not realise interfaces directly. An interface can be realised as a provided or required interface. A connection is established between two compatible component ports, that is, ports that realise the same interface: one in a required mode and the other one in a provided mode. This is described in Constraint 1 by invariant *conjugatedPort* that specifies all ports are conjugated. We distinguish ports according to the components they belong to. *BasicComponent* ports are connected at most to another component port. On the other hand, a port from a *HierarchicalComponent* is connected at most to two other ports, one to a port of an inner component instance, and another to a port of an external component instance.

Component Services. The contract of a component must be modelled using ports. Each component class must have ports exposing the required and provided services. Then, Ports

describe the operations that a component needs or performs. This is captured by invariant *interfaceOperations* Constraint 1, the property ownedOperation informs a set of operations of a interface. .

Operations. Components can execute operations that are defined in interfaces that are realised by their ports. An operation can be asynchronous or synchronous. We define asynchronous operations as a signal; it is indicated by a stereotype signal in the operation declaration. In our example, Figure 12, *picksup* and *putsdown* are synchronous operations and do not need an explicit stereotype. A synchronous operation blocks the caller until the operation completes.

Operation Parameter. The parameter modes determine the behaviours of parameters. If an operation has parameters, its parameter modes have to be defined either as *in* (input) or as *out* (output).

State Machine. A State Machine describes the behaviour of the component. Its name needs to be the same as that of the owner component prefixed with the term *STM_*, to conform to the constraint *NameStateMachine*. A state machine must have at least one state and one transition. Transitions must have a valid pair (port/operation) described in their trigger or action fields. It is also possible to describe guards for transitions.

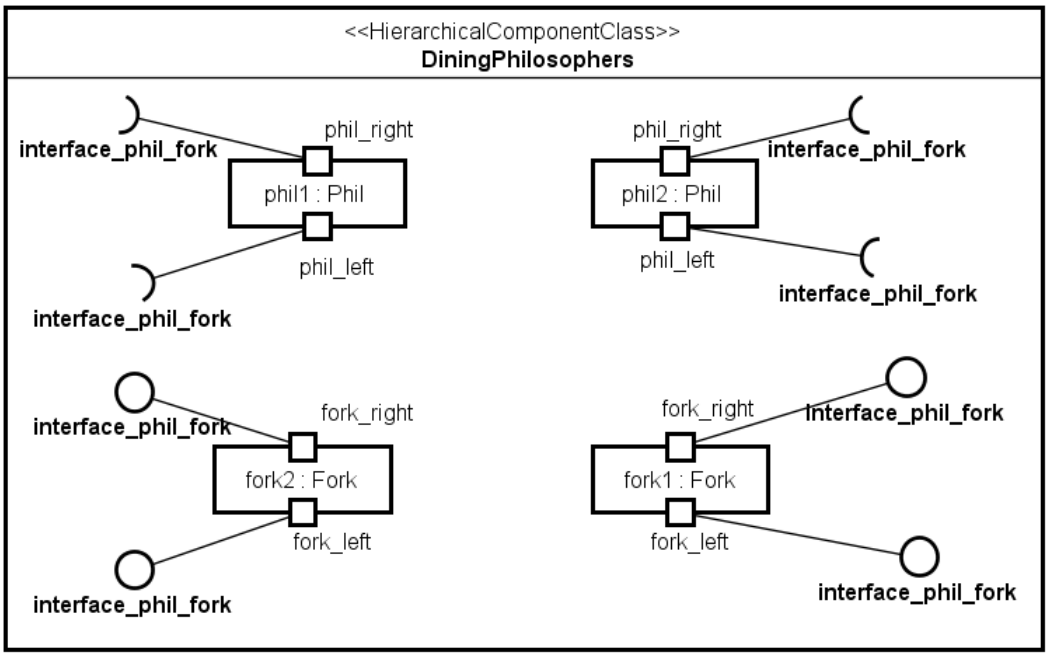
Port Multiplicity. If there is a connector between two ports where at least one of them has multiplicity greater than one, the connector must be labelled to indicate the port being connected. The label must follow the pattern *port1_name[j]↔port2_name[i]*, where *port1_name* and *port2_name* are the name of ports; *j* and *i* are the indices of the port of the connection.

3.3 COMPOSITION OF COMPONENT INSTANCES

The composition of component instances is described using a *Hierarchical Component* element, where it is possible to create instances of the components and make connections between them. The simplest form of composition is *Interleave composition*; there is no communication among the component instances. This composition is achieved by instantiating component instances in the structure of a hierarchical component class. Each instance has a type related to a component previously defined. For example, in Figure 13 we show two instances of $Fork$ and two of $Phil$ in a hierarchical component. When component instances are created, they are, by default, in interleaving.

Communications are performed through the connection of ports from two different components. The same interface must be provided by one component and required by the other

Figure 13 – Interleave

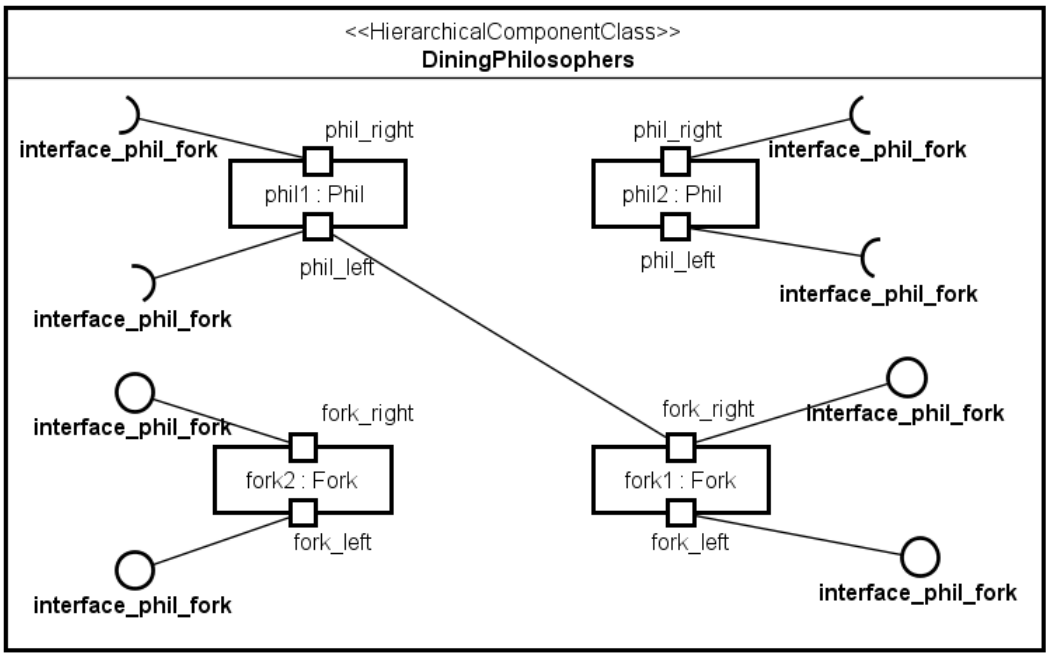


Source: Author's ownership.

one.

Figure 14 illustrates a communication between $fork1$ and $phil1$. This communication happens through the connection from port *phil_left* of *phil1*, that requires the interface *interface_phil_fork*, to port *fork_right* of *fork1* that provides the interface *interface_phil_fork*.

Figure 14 – Communication



Source: Author's ownership.

4 FORMAL SEMANTICS AND COMPOSITIONAL VERIFICATION

In order to perform a mechanised compositional verification during the model construction, we translate the UML Metamodel to BRIC Metamodel , which itself uses CSP as the underlying formal notation.

One question that may arise is whether the defined semantics properly captures the intended behaviour of the component model. This, of course, cannot be proved, unless we consider independent semantics as reference. Unfortunately, there is no complete semantics for UML. However, the problem is minimised in our particular case, since our semantics is inspired by the Foundational UML works (Object Management Group (OMG), 2017b; Object Management Group (OMG), 2019b; Object Management Group (OMG), 2019a).

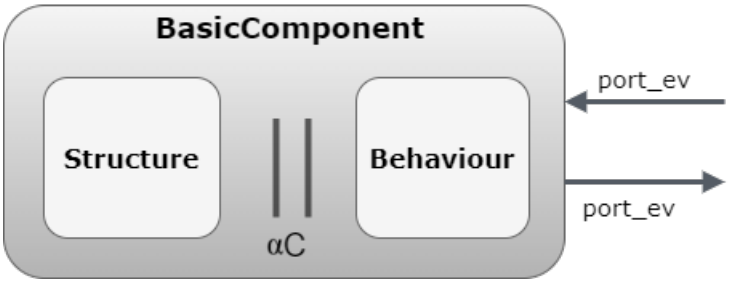
Concerning component interactions, in particular, they are asynchronous: always intermediated by a buffer. Interoperation of asynchronous system is an important issue an real concurrent systems. However, the UML message exchanging (synchronous or asynchronous) between components can be specified using this mechanism. In the context of our formal semantics, all possible interleavings of messages must be taken into account, but operational issues of a specific execution environment, like scheduling which may impact the order or the priority of messages, is not our concern.

First, we give an overview of the behaviour of a component contract BRIC for UML components; next, we present the rules that formalise the formal semantics definition, and then we discuss the verification strategy.

4.1 OVERVIEW

In the previous Chapter, we defined well-formedness conditions for our model; this encodes restrictions in order to define a meaningful model. The objective of providing a formal semantics for our UML Component Model is to define the behaviour of a component, its communication through ports and the interaction between components.

Figure 15 – Illustration of BasicComponent in CSP



Source: Author's ownership.

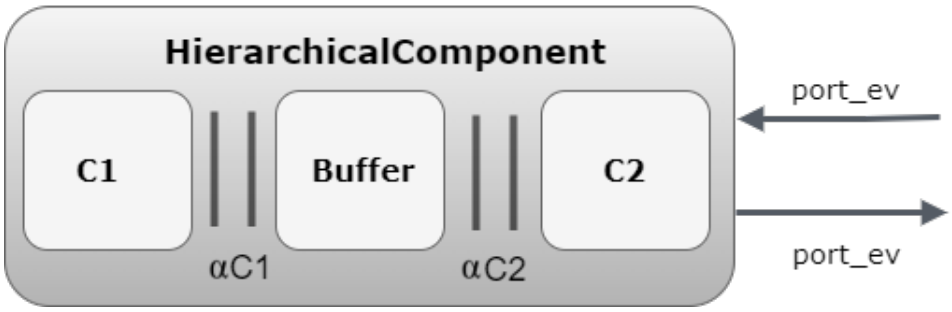
As we have previously explained, our UML Component Model has two types of components: *BasicComponent* and *HierarchicalComponent*. Figure 15 shows the high-level architecture of the behaviour element of BRIC contract, a *BasicComponent* in CSP. The rounded boxes represent CSP processes, and the arrows illustrate communication of events related to a port. It has two processes that are composed in parallel and ports for communication. The process on the left-hand side, *Structure*, represents the attributes of the component, captured by a process that defines a memory for accessing the attributes; these are specified in a UML *BasicComponentClass* by class attributes. The need to represent this memory as a process is that, as a process algebra, CSP processes are stateless. Instead of using CSP process with parameters, we choose the memory process since it creates a local context. A behaviour process uses additional events to query the memory process as to whether a guard is true.

The process on the right-hand side, *Behaviour*, captures the core dynamics of a component. It results from the translation of the component State Machine.

The two processes synchronise on the set αC, which has events for reading, and setting the value of each attribute; this set also has one event associated with each guard in the component state machine. The event occurrence means that the corresponding guard is true. This is a technical detail necessary to impose atomicity in the evaluation of guards that may include global (and shared) variables. More details about this mechanism is presented in Section 4.2.

A *BasicComponent* allows communication with other components or with the environment through ports. In Figure 15, this is represented by the arrows named *port_ev*. The incoming and outgoing arrows represent the input and output communications of a component, respectively. Ports are translated to CSP channels, while operations and signals are translated to CSP events.

Figure 16 – Illustration of HierarchicalComponent in CSP



Source: Author's ownership.

A *HierarchicalComponent* is specified by the parallelism of its internal component instances. As shown in Figure 16, the behaviour element of BRIC contract, consider a *HierarchicalComponent* that has two internal component instances, namely *C1* and *C2*. A *HierarchicalComponent* can be composed of n components; however, the composition is made in pairs. These instances must be either a *BasicComponent* or a *HierarchicalComponent* previously defined. The connections between instances owned by *HierarchicalComponents* are specified in UML using a Composite Structure Diagram, as illustrated in Figure 14. These component instances can communicate between themselves, which is represented by connectors.

Whenever two component instances are connected through their ports in UML, such a connection is represented in CSP by the parallel composition of the components' processes and a *Buffer* process that orchestrates the communication between the components. The Buffer works as an intermediate element of the composition, transferring information from one component to another. Information is always accepted, independent of the other component being ready to input. This *Buffer* is not a first-class element; it is implicit in our component model.

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, in Figure 16, 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*. In this way, in our running example, the dining philosophers, the *Phil* component requires services, picking up and putting down, which are provided by the *Fork* component. The communication occurs through the connection between their ports. For example, in Figure 14 the port *phil_left* from the instance $phil1$ of the component *Phil* is connected to

the port *fork_right* from the instance $fork1$ of the component *Fork*; through these ports the events *picksup* and *putsdown* are communicated and the buffer, which is shared between the instances of the components, ensures that the ordering of the events exchanged between the two instances is preserved.

If there is no connection between the components, the synchronisation sets (αC_1 and αC_2) are empty, and the buffer has no effect. In CSP, the processes that capture the behaviour of C_1 and C_2 are combined in interleaving.

Finally, a *HiearchicalComponent* can also communicate with external entities through its ports: *port_ev* arrows in Figure 16 illustrate this scenario.

4.2 FORMAL SEMANTICS

The concepts introduced in the previous section are formalised in a denotational semantics of our Component Model using BRIC as the semantic domain. For each syntactic element from the metamodel is given a semantic function to map this element into its denotation in the (BRIC) semantic domain. We use double brackets, $[\![\_]\!]$, to identify a semantic function and its argument is a syntactic element from our UML Component Model. Semantic functions can use auxiliary functions.

The definition of the translation rules adopts some conventions: the title of semantic functions starts with the term *Semantics of* and that of auxiliary functions starts with *function*; the header of the rule identifies a function and its parameters; control flow statements are presented in italics font (e.g. *for each*, *if-then-else*); elements of the Component Model are accessed using a dot notation (e.g. *c.StateMachine* refers to the State Machine of the component denoted by *c* and *c.name* accesses the name of a component), and the meta-notation is underlined, and is written in a light-grey font colour. The content in typewriter font refers to a CSP text.

We describe in this section the main rules. All rules and auxiliary functions are described in Appendix B.

Rule 1 gives the semantics of a model by mapping this model to a BRIC Component; the semantic function $[\![\_]\!]_{\mathcal{M}}$ takes a model *M* as argument and, for each *AbstractComponent*, such as *BasicComponent* or *HierarchicalComponent*, in *M*, it invokes the function BricContract that builds a BRIC contract given a UML *AbstractComponent* passed as argument.

Rule 1. Semantics of Component Model

$[\![$M : Model$]\!]_{\mathcal{M}}$: List of BricComponent =

for each c in M.AbstractComponent
  BricContract(c)
end for

The auxiliary function BricContract, Rule 2, yields a tuple with the elements that compose a BRIC contract signature, as explained previously in Section 2.3 (Definition 3); and from this tuple all elements of BRIC are defined. The first element is the behaviour of the component. It is represented by a CSP process that has the same name as the component *(c.name)* and it is parameterised by an `id` to uniquely identify its instances. The range of `id` is defined during the translation. This CSP process is defined by Rule 6.

Rule 2. Function bricContract

bricContract(c : AbstractComponent) : BricSignature =

⟨
  c.name(id),
  relation(c),
  interface(c),
  communicationChannel(c)
⟩
$[\![$c$]\!]_{\mathcal{C}}$

The fourth element of the tuple of a BRIC contract is the set of channels of the component, which is yielded by the auxiliary function communicationChanel(c). This function, Rule 3, fetches all port names from the structured class and concatenates each one with the identifier *.id* and composes them as a set.

This set for the *Fork* component is:

$$\{fork\_right.id, fork\_left.id\}$$

Rule 3. Function communicationChannel

communicationChannel(c : AbstractComponent) : SetPortName =

$$\{p : PortName \mid \exists\, class \in c.StruturedClass \bullet \exists\, port \in class.Ports \bullet p = port.name^\frown \texttt{.id}\}$$

The third element of the contract in Rule 2 is the interface of communications interface(c). Rule 4 describes the set of operations and signals that the component can communicate. These elements are collected from the interfaces that the component ports realise. A port must realise an interface. Interfaces can be realised in either provided or required mode. Interfaces describe operations or signals in our component model. The difference between them in our semantics is that an operation uses synchronous communication. To capture this in CSP we create two events for each operation: an input event and output event, which represent the operation call and its reply, respectively. An operation name is concatenated with the string *_I* for the former case and with *_O* for the latter. A signal uses asynchronous communication and is encoded as a homonymous channel. The union of these two sets composes the interface of communications.

For the *Fork* component, the set of interfaces is represented by:

$$\{picksup\_I, picksup\_O, putsdown\_I, putsdown\_O\}$$

Rule 4. Function interface

interface(c : AbstractComponent) : SetInterfaceCommunication =

$$\{i : interface\_Name \mid \exists\, class \in c.StructuredClasses \bullet \exists\, port \in class.Ports \bullet \exists\, interface \in (port.RequiredInterfaces \cup\ port.ProvidedInterfaces) \bullet \exists\, operation \in interface.Operation \bullet i = operation.name^\frown \_0 \vee i = operation.name^\frown \_I\}$$
$$\cup$$
$$\{i : interface\_Name \mid \exists\, class \in c.StructuredClasses \bullet \exists\, port \in class.Ports \bullet \exists\, interface \in (port.RequiredInterfaces \cup port.ProvidedInterfaces) \bullet \exists\, signal \in interface.Signal \bullet i = signal.name\}$$

The bricContract function, Rule 2, also invokes the function relation(c) that describes the relationship between the channels and the interfaces. It is presented in Rule 5, and it yields a set of pairs *(portName, {interface})* that represents the link between ports (channels) and the interfaces (types). We use *required_provided_interface(p)* as an auxiliary function to return all interfaces that are realised by port *p*. The symbol $\mathbb{P}$ stands for power set.

Rule 5. Function relation

relation(c : AbstractComponent) : relationPortInterface =

$$\{(p : portName, i : \mathbb{P}\, interface) \mid \exists\, class \in c.StructureClass \bullet$$
$$\exists\, port \in class.Port \bullet p = port.name \wedge$$
$$i = required\_provided\_interface(port)\}$$

The relation between communication channels and interfaces from *Fork* is defined by the tuple:

$$\langle (fork\_right.id, \{picksup\_I, picksup\_O, putsdown\_I, putsdown\_O\}),$$
$$(fork\_left.id, \{picksup\_I, picksup\_O, putsdown\_I, putsdown\_O\})\rangle$$

The semantic function $[\![\_]\!]_{\mathcal{C}}$, Rule 6, defines the semantics of a behaviour of a BRIC component. This function takes a component as argument and defines the *CSP Specification* that captures the component behaviour. It verifies if the component is a *BasicComponent* or a *HierarchicalComponent*, and it invokes the corresponding semantic function $[\![\_]\!]_{\mathcal{BC}}$ or $[\![\_]\!]_{\mathcal{HC}}$ to define the behaviour of a component.

Rule 6. Semantics of Component

$[\![$c : AbstractComponent$]\!]_{\mathcal{C}}$: CSPSpecification =

if $isbasicComponent(c)$ *then*
  $[\![$c$]\!]_{\mathcal{BC}}$
else
  $[\![$c$]\!]_{\mathcal{HC}}$

The function $[\![\_]\!]_{\mathcal{BC}}$, presented in Rule 7, considers all elements of a basic component. The first one are events that represent operations of the component and are derived from the interfaces realised by the ports of the component: input and output channels. 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 parameter. To generate this information, inputs and outputs, we use the functions subtype_operationsInput(c) and subtype_operationsOutput(c), respectively.

The ports of the *FORK* component provides one interface that has two operations: *picksup* and *putsdown*. Then they are translated to a CSP_M datatype and two subtypes (one for inputs and another for outputs). [1]

$$\begin{aligned} &datatype \quad fork\_operation = picksup\_I \mid picksup\_O \mid putsdown\_I \mid putsdown\_O \\ &subtype \quad fork\_I = picksup\_I \mid putsdown\_I \\ &subtype \quad fork\_O = picksup\_O \mid putsdown\_O \end{aligned}$$

The function $[\![\_]\!]_{\mathcal{BC}}$ also retrieves the ports of the component which, in CSP, are channels that are generated by the auxiliary function *channel_port(c)*, Rule 8. Each port gives rise to one channel of communication if the port belongs to a *BasicComponent*. Otherwise, if it is part of a *HierarchicalComponent*, two communication channels are associated with its sides: internal and external. The internal channel refers to the connection of internal components to the port. The external channel refers to an external connection to the port or the environment.

Rule 7. Semantics of Basic Component BRIC

$[\![$c : AbstractComponent$]\!]_{\mathcal{BC}}$: CSPSpecification =

```
datatype c.name_operation = subtype_operationsInput(c) | subtype_operationsOutput(c)
subtype c.name_I =  subtype_operationsInput(c)
subtype c.name_O =  subtype_operationsOutput(c)
channel_port(c)
channel_get_var(c)
channel_set_var(c)
memory(c)
[[c.State]]STM
mainProcess(c)
```

The channels that represent ports of the *Fork* component are:

$$\begin{aligned} &channel \quad fork\_right : id\_Fork.fork\_operation \\ &channel \quad fork\_left : id\_Fork.fork\_operation \end{aligned}$$

[1] Complex data types can be defined in CSP_M by the construtors $datatype$ and $subtype$ that specify set of atomic constants. In CSP there is no need of using constructors.

Rule 8. Function Channel Port

channel_port(c : AbstractComponent) :: CSPSpecification =

for each class in c.StructureClass
  for each port in class.port
   if(*port.OwnedByBasicComponent*)
    channel *port.name* : ID_*c.name*.*c.name*_operation
   else
    channel *port.name*_internal : ID_*c.name*.*c.name*_operation
    channel *port.name*_external : ID_*c.name*.*c.name*_operation
  end for
end for

As previously mentioned, as a process algebra, CSP is stateless. Then, in our work, class attributes are represented as a memory process. Therefore *set* and *get* channels are necessary to assign and recover values to and from these attributes. The function channel_set_var(c) creates, for each attribute of a component, a channel used to assign new values to this attribute. There is no attribute in the Dining Philosophers, however, considering a component with an attribute, namely *philosopherName*, this function would yield *channel set_philosopherName: id_Phil.String* where *id_Phil* is the identifier of the component and *String* is the type of *philosopherName*. Similarly, function channel_get_var(c) creates channels to access the values of the attributes. In this case, the channel would be *channel get_philosopherName: id_Phil.String*.

In Rule 9 we have function memory(c) that defines the memory process for a component (*Structure* process illustrated in Figure 15). As already explained, its purpose is to control the access to attributes and internal events of the component. The process for the component memory records local variables (attributes of the component). These variables/attributes are generated by the function varList(c). The function vid calculates a unique identifier for variables formed for component name and attribute name. This function, vid(v), is used to define *set* and *get* channels.

By varList(c)[valueName(v) := x] we denote the list of variables by the name valueName(v) replaced with the value *x*, when the set event is communicated.

Guarded transitions in the process resulting from the semantic definition of a state machine (Rule 11) are also controlled by events of the memory process. This mechanism allows the atomicity for checking a guard that uses attributes of the component, thus, avoiding changes in the value of attributes while a guard is being evaluated.

Rule 9. Function memory

memory(c : AbstractComponent) :: CSPSpecification =

$c.name$_memory (id,varList(c)) =
  (□ v:varList(c) • get_vid(v)id!name(v) → $c.name$_memory (id , varList(c)))
    □
  (□ v:varList(c) • set_vid(v)id?vl → $c.name$_memory (id,varList(c)[name(v) := vl]))
    □
  memoryGuards(c.StateMachine.transitions)

The guards are translated by the function memoryGuards(c). Each iteration of the *for each* construct generates a process that captures the behaviour of a transition. The *sep* clause with the external choice operator (□) means that each pair of such processes is combined by external choice.

For each transition from the state machine, Rule 10 verifies if there is a guard and if there is a trigger. When both exist, a boolean expression is formed of the semantics of a guard, $[\![\_]\!]_{\mathcal{GRD}}$; then a trigger, $[\![\_]\!]_{\mathcal{TRG}}$, is indexed by a number that identifies this statement as unique. This identifier is given by the function generatedIdTrs(). Otherwise, if there is no trigger, an expression is formed of the semantics of the guard, as a prefix (→) and an *internal* event that is indexed by the result from generatedIdTrs(). This memory process enables or disables a particular event *internal.x*, where *x* comes from generatedIdTrs(), depending on whether the guard for the transition with identifier *x* holds or not. This allows the synchronisation with the process that represents the state machine. Since the synchronisation involves the trigger channel that can be constrained by different guards in other transitions, the unique identification avoids ambiguity. On the other hand, when a transition includes only a guard, this guard is associated to a channel named *internal* and, in a similar way to guards and triggers, it has an identifier given by generatedIdTrs().

Rule 10. Function memoryGuards

memoryGuards(c : AbstractComponent) :: CSPSpecification =

for each t in c.StateMachine.transitions sep □
  if(*not empty*(*t.guard*) *and not empty*(*t.trigger*)) *then*
    $[\![$t.guard$]\!]_{\mathcal{GRD}}$ & $[\![$t.trigger$]\!]_{\mathcal{TRG}}$.generatedIdTrs()
      → *c.name*_memory (id, varList(c))
  elseif(*not empty*(*t.guard*) *and empty*(*t.trigger*)) *then*
    $[\![$t.guard$]\!]_{\mathcal{GRD}}$ → internal.generatedIdTrs()
      → *c.name*_memory (id, varList(c))
  end if
end for

As the Dining Philosopher example does not have attributes, we illustrate Rule 10 with the memory process of the Control component from the Ring Buffer case study presented in Section 5.1. It is parameterised by *id* and the other component attributes. We present a fragment of this process. The $get\_size.id!size$ event communicates the current value of the attribute size for this instance identified by id, after which the process recurses preserving the values of all attributes in the memory. The event $set\_size.id?vl$ receives the new value (vl) of the size attribute so that the memory is updated with this value in the subsequent recursive call. Events get and set for the other attributes are similar. When the attribute $size$ is greater than zero and the event $port\_env.id.retrieve\_data\_I$ is performed, and when the $size$ is equal to one, and the event $internal.3$ is engaged, the memory process is called recursively preserving all attributes.

$CONTROL\_memory(id, size, cache, top, bot, vl\_env) =$
  $get\_size.id!size \rightarrow CONTROL\_memory(id, size, cache, top, bot, vl\_env)$
□
...
  $set\_size.id?vl \rightarrow CONTROL\_memory(id, vl, cache, top, bot, vl\_env)$
□
...
  $size > 0 \quad \& \quad port\_env.id.retrieve\_data\_I \rightarrow CONTROL\_memory(id, size, cache, top, bot, vl\_env)$
□
  $size == 1 \quad \& \quad internal.3 \rightarrow CONTROL\_memory(id, size, cache, top, bot, vl\_env)$
...

Rule 11 specifies the semantic function $[\![\_]\!]_{\mathcal{STM}}$ that formalises the core behaviour of the component by translating its state machine to a CSP process that has as signature the string

*STM_* concatenated with the component name, and it is parameterised by the component instance identifier *id*. Each state of the state machine becomes a CSP process also with a component instance identifier *id* as argument and the semantics of a state is given by the function $[\![\_]\!]_{\mathcal{STATE}}$. The first invoked process is the one reachable from the initial pseudostate. It is represented by *stm.FirstState.name(id)*.

Rule 11. Semantics of State Machine

$[\![\text{stm} : \text{StateMachine}]\!]_{\mathcal{STM}} : \text{CSPSpecification} =$

STM_$stm.name$(id) = $stm.FirstState.name$(id)
for each st in stm.State
  $st.name$(id)= $[\![\text{st}]\!]_{\mathcal{STATE}}$
end for

We consider only simple states. Therefore, in the definition of $[\![\_]\!]_{\mathcal{STATE}}$ each transition from the source state is evaluated using $[\![\_]\!]_{\mathcal{TR}}$; the transitions are composed in external choice.

Rule 12. Semantics of State

$[\![\text{st} : \text{State}]\!]_{\mathcal{STATE}} : \text{CSPProcess} =$

$\Box\ tr : \text{transitionFrom(st)} \bullet [\![\text{tr}]\!]_{\mathcal{TR}}$

The semantics of a transition, Rule 13, is given by the evaluation of `str_transition` that is formed of the translation of guards, trigger and action of a transition; it behaves as a process that has the name of the target state of the transition and is parameterised by the component identifier. The auxiliary function getIdTrs(tr) yields a number that indexes the transition. Note that, in Rule 13, if an action is not present, then the process `str_action` behaves like SKIP. The process `str_trigger_action` captures the semantics of a trigger, if it exists, and then behaves as the process that captures the behaviour of `str_action`. Finally, `str_transition` behaves as `str_trigger_action`, but preceded by an internal event that represents the guard evaluation, if there is a guard and no trigger. This internal event synchronises the memory with the event described in Rule 10.

Rule 13. Semantics of Transition

$[\![\text{tr : Transition}]\!]_{\mathcal{TR}}$: CSPProcess =

```
let
  str_action =
    if not empty(tr.action)  then
      [[tr.action]]_ACTION → SKIP
    else
      SKIP
    end if

  str_trigger_action =
    if not empty(tr.trigger)  then
      [[tr.trigger]]_TRIGGER.getIdTrs(tr) → str_action
    else
      str_action
    end if

  str_transition =
    if not empty(tr.guard) and empty(tr.trigger)  then
      internal.getIdTrs(tr) → str_trigger_action
    else
      str_trigger_action
    end if

  within
    str_transition ; tr.target.name(id)
```

Considering again our running example, the Dining Philosophers, we show in Figure 11 (on page 51) the state machine diagram of *Phil*. It is translated to a CSP process where each state is a process. The main process is the first one that can be reached in the machine: *HoldForkR*, where the philosopher picks up the right fork. In this machine, operations are designed as actions without guards. Triggers and actions on transitions are represented by channels; in this case the action *phil_right* is a channel that represents the communication through the port of the same name. Events are communicated through this channel whose type is a pair: $id.operation$: id is the component instance identifier, and $operations$ is the set $\{putsdown\_I, putsdown\_O, picksup\_I, picksup\_O\}$. The evaluation of a transition from the state *HoldForkR* to *HoldForkL* results into a sequence of input and output events:

$$phil\_right.id.picksup\_I \rightarrow phil\_right.id.picksup\_O \rightarrow HoldForkL(id)$$

The following process represents the behaviour of *Phil*, modelled in the state machine

diagram showed in Figure 11. It cyclically picks up a fork in the right port, then in the left one, and finally puts down via the same port.

$$STM\_Phil(id) = HoldForkR(id)$$
$$HoldForkR(id) = (phil\_right.id.picksup\_I \rightarrow phil\_right.id.picksup\_O \rightarrow HoldForkL(id))$$
$$HoldForkL(id) = (phil\_left.id.picksup\_I \rightarrow phil\_left.id.picksup\_O \rightarrow PutsDownR(id))$$
$$PutsDownR(id) = (phil\_right.id.putsdown\_I \rightarrow phil\_right.id.putsdown\_O \rightarrow PutsDownL(id))$$
$$PutsDownL(id) = (phil\_left.id.putsdown\_I \rightarrow phil\_left.id.putsdown\_O \rightarrow HoldForkR(id))$$

The last step of Rule 7 is a call to the function mainProcess(c) (Rule 14), which defines a process that represents the structure and behaviour of a component. This process is defined by STM_*c.name(id)* composed in parallel with memory_*c.name*(id, valueList(c)) where valueList(c) returns the default values of the attributes. The synchronisation set provided by the function setSync(c) includes get and set channels and internal events. In order to keep only the channels that represent ports visible to other components, channels as get, set and internal channels are hidden; this set is obtained by the function setHidden(c).

Rule 14. Function Main Process

mainProcess(c : Component) : CSPProcess =

```
c.name(id) =
  (STM_c.name(id)
       ||
  {|setSync(c)|}
  memory_c.name(id, valueList(c))) \ {| setHidden(c) |}
```

In the Dining Philosophers example, neither *Fork* nor *Phil* components have attributes. Therefore, there is no need for a memory to record state information. They are represented only by their state machine processes. Then, in these cases the Rule 14 is replace by Rule 15 where the main process is formed only by the process that describes the state machine.

$$FORK(id) = STM\_Fork(id)$$
$$PHIL(id) = STM\_Phil(id)$$

Rule 15. Function Main Process
mainProcessNoMemory(c : Component) : CSPProcess =

$c.name$(id) = STM_$c.name$(id)

With the basic component definition it is possible to define the semantics of hierarchical components since this is a collection of component instances.

In Rule 17, the semantic function $[\![\_]\!]_{\mathcal{HC}}$ describes the behaviour of the hierarchical component. It retrieves the instances from the hierarchical component metamodel element, and, initially, composes all of them in interleaving, which is the type of communication when none of the instances are connected, as shown in process *CON(0)*, as described in Rule 16 . In this rule, component instances are composed in pairs in interleaving.

Rule 16. Function Intervealing Component
InterleaveProcess : CSPProcess =

let
interleave(0) = *SKIP*
i = $length(c.instances)$

for each instance in c.instances
interleave(i) = interleave($i-1$) ||| $instance.name$
end for
within
$CON(0)$ = interleave($length(c.instances)$) ||| portProcess(c)

The result of the interleave composition is also composed in interleave with ports from a *HierarchicalComponent*. These ports are described as a process yielded by the function call portProcess(c), which returns the interleaving of all port processes. Each port process uses two channels, as defined in Rule 8 (internal and external). This process relays the communication according to its direction. When an external message arrives at the port, the event from the external port happens followed by the event of the internal port. When the message comes from an internal communication, then the events occur in the opposite direction. Then, for each connection between two component instances, a new process *CON(i)* is defined by composing *CON(i-1)* in parallel with a Buffer process (Rule 18) that orchestrates the message communication order; the synchronisation set is formed of the ports involved in the connection

($i.port[1]$ and $i.port[2]$). [2] Then, it is a representation of a behaviour of unary composition where assemble distinct channels of a same component.

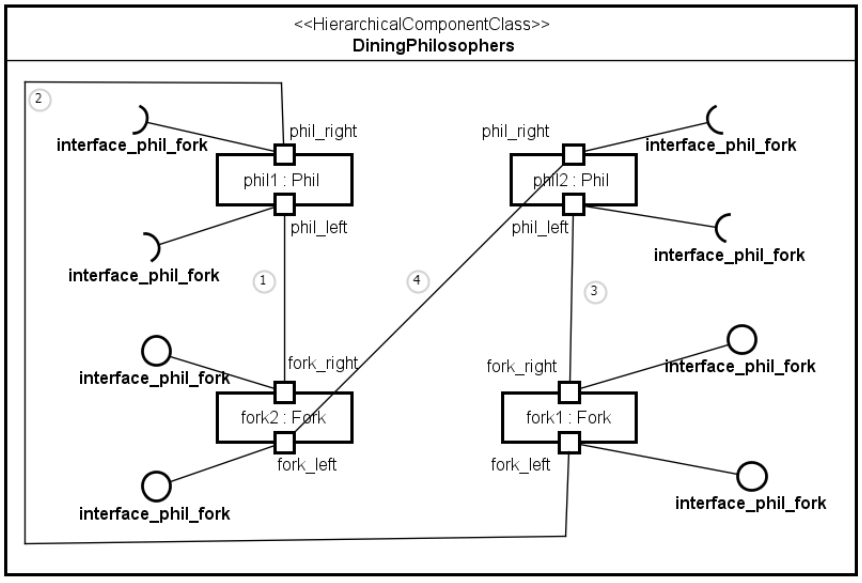
Rule 17. Semantics of Hiearchical Component Behaviour

$[\![$c : AbstractComponent$]\!]_{\mathcal{HC}}$: CSPProcess =

$$
\begin{array}{l}
let \\
\quad \texttt{CON(0)} = \text{InterleaveProcess(c)} \\
\quad for\ each\ i\ in\ c.connections \\
\quad\quad \texttt{CON}(i) = \texttt{CON}(i-1) \underset{\{|i.port[1],i.port[2]|\}}{||} \text{Buffer}(i.port[1],i.port[2]) \\
\quad end\ for \\
within \\
\quad c.name(id) = \texttt{CON}(length(c.connections))
\end{array}
$$

Figure 17 shows the hierarchical component DiningPhilosophers, where each instance of basic components *Fork* and *Phil* (*fork1*, *fork2*, *phil1*, *phil2*) are connected. In our translation it is represented in CSP by the parallel composition of the component processes and a buffer process that orchestrates the communication in each pair of connections between component instances. Although there is no ordering for the connections, we have numbered them in Figure 17 to facilitate the understanding of how we compose the different instances.

Figure 17 – Dining Philosophers - connections



Source: Author's ownership.

[2] The function *length* returns the number of connections.

The function $[\![\_]\!]_{\mathcal{HC}}$ composes all instances from the Dining Philosophers component in interleaving, generating the *CON(0)* process. Then, a process is generated for each connection between component instances. The *CON(1)* process is defined to represent the connection between *phil1* and *fork2*. This new process composes in parallel the previous process *CON(0)* and the *BFIO* process, which is a buffer with two channels, one input (*ci*) and one output (*co*) of the same type (see Rule 18). It copies information from its input channel (*ci*) to its output channel (*co*), without loss or ordering. In our example, the *BFIO* process has as arguments the channels that represent the ports of this particular connection. That is, channels $port\_fork\_right.2$ ($i.port[1]$) and $port\_phil\_left.1$ ($i.port[1]$) form the synchronisation set for connection 1.

$$CON(0) = Fork(1) \;|||\; Fork(2) \;|||\; Phil(1) \;|||\; Phil(2)$$

$$CON(1) = CON(0) \underset{\{|port\_fork\_right.2, port\_phil\_left.1|\}}{||} BFIO(port\_fork\_right.2, port\_phil\_left.1)$$

In a similar way, other connections can be established between instances. For example, a connection 2 between *fork1* and *phil1* produces *CON(2)* that is given by parallel composition of *CON(1)* and a buffer, synchronising on $port\_fork\_left.1$ and $port\_phil\_right.1$.

Rule 18. Function Buffer Process

BFIO(ci : channel, co : channel) : CSPProcess =

Buffer_aux$(ci, co, 1)$ ||| Buffer_aux$(co, ci, 1)$

The intermediary buffer maps outputs from an instance of *FORK* into inputs to an instance of *PHIL*, and vice-versa. These internal buffers perform asynchronous bidirectional communication, which conveys information in both directions (see Rule 19). The process *Buffer* is a buffer of size *n*. The information communicated are defined by type of *ci* and *co*.This buffer gathers these information from the function *outputC_All* with receives a channel *ci* as parameter and returns all events *x*, such that the event *ci.x* is an output of the component on the channel ci. [3]

[3] The length s of a sequence s is the number of elements it contains.

Rule 19. Function Buffer Auxiliary Process

Buffer(ci : channel, co : channel, n : integer) : CSPProcess =

$$
\begin{array}{l}
let \\
\quad B(\langle\rangle) = \Box\ x : outputsC\_All(ci) \bullet ci.x \rightarrow B(\langle x \rangle) \\
\quad B(s) = (co!head(s) \rightarrow B(tail(s))) \\
\qquad \Box \\
\qquad\quad (\#s < n \quad \& \\
\qquad \Box\ x : outputsC\_All(ci) \bullet ci.x \rightarrow \quad B(s^\frown\langle x \rangle)) \\
\quad within \\
\qquad B(\langle\rangle)
\end{array}
$$

Once the semantics is defined, it is possible to translate well-formed UML component models, based in our metamodel, to BRIC components and, afterwards, the composition rules are applied, conduct formal analyses.

4.3 PROTOCOL GENERATION AND VERIFICATION STRATEGY

As the behaviour of a BRIC component is described in CSP, it can be formally verified using the FDR model checker (GIBSON-ROBINSON et al., 2014). BRIC defines a set of assertions that represent the BRIC properties that must be checked of a BRIC model. For instance, it is mandatory to check whether a contract is an I/O Process. Additionally, we described the composition rules that provide a systematic method to preserve deadlock freedom by construction where, for each connection, a set of verifications is performed in a compositional fashion. As we translate the UML components to BRIC, the BRIC-related properties for each component and the connections between them are also checked in a compositional way.

In this section, we describe the mechanisation of the protocol implementation generation that is necessary in the side conditions of the feedback and reflexive BRIC composition rules. Local analyses are made to check the compatibility between communication protocols, verifying whether any communication problem such as deadlock is introduced with the composition.

Then, we recall that performing such analyses is simpler than verifying the complete model and we show how the deadlock verification can be performed and how we trace the results back to the UML model.

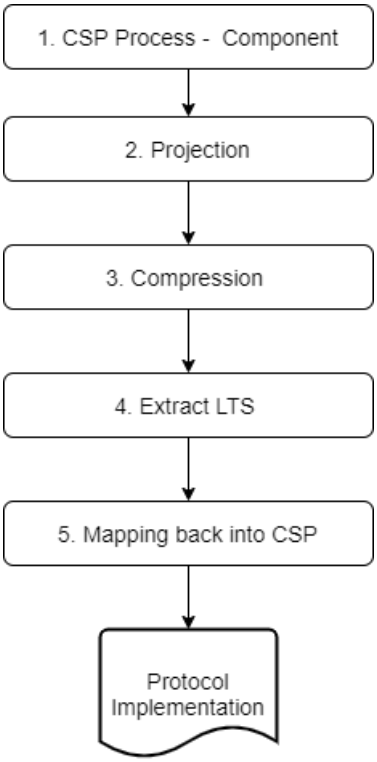
4.3.1 Automatic Protocol Generation

When a component is modelled in UML according to our metamodel and then translated into BRIC, it is necessary to define a protocol implementation to allow communication between component instances.

A protocol implementation is given by the projection of the process over a subset of the communication channels. However, as the projection is expressed by internalising the other channels (those not in the projected set), this may introduce divergence (livelock) due to the usage of the CSP hiding operator, which may create cycles of internal events.

So far, the definition of a protocol implementation is a completely manual task, where the designer must analyse the process of a component and define this process accordingly. In order to facilitate this endeavour and make the methodology less error-prone, we propose an automated strategy for generating a protocol for a component process. We systematise the generation of the protocol as shown in Figure 18.

Figure 18 – Protocol Implementation Steps



Source: Author's ownership.

The first step is obtained by Rule 14 which generates a process P that represents the behaviour of a component. Next, we create a temporary process Q given by the process P

with the projection over the communication channels: $Q = P \setminus (diff(\Sigma, C))$ [4], where C is the set of channels being projected. In other words, we hide all the channels, except those in C, which are of interest to the protocol. This can result in a process with livelock, as already explained. To overcome this issue, we perform normalisation on process Q to remove possible livelocks. First, we obtain the Labelled Transition System (LTS), which gives an operational semantics for this process. The LTS can be automatically obtained from a CSP process using, for example, the FDR tool.

In the third step, we use the weak bisimulation (Definition 2) to compress the result of the projection step.

FDR implements a different notion of weak bisimulation known as *divergence-respecting weak bisimulation*; this differs from that presented in Definition 2 in that it preserves τ self-loops that characterise divergence.

For compressing a process, FDR uses a variation of LTS, called GLTS (Generalised LTS) (BOULGAKOV; GIBSON-ROBINSON; ROSCOE, 2014), in which there are no τ actions in transitions; a τ self-loop is represented as an attribute of a GLTS state. Then, in the fourth step of Figure 18, an LTS is extracted from the GLTS, and these τ self-loop attributes are ignored. In this way, the resulting LTS has no divergences and conforms to the weak bisimulation notion as defined in Definition 2.

The last step is to convert this LTS into a new CSP process. For this, we use an approach defined in (SAMPAIO et al., 2014), where each transition is translated into a CSP process prefixed with the corresponding event, and whose behaviour is given by recursively mapping the transitions of the target state. After this step, we have a valid protocol implementation. The resulting process preserves the refinement expressions of Definition 7. The fact that these assertions are obeyed by the generated protocol implementation follows from the relation between the two variations of bisimulation: Definition 2 and the divergence-respecting weak bisimulation implemented by FDR; for more technical details we refer the reader to (BOULGAKOV; GIBSON-ROBINSON; ROSCOE, 2014).

4.3.2 Verification Strategy

Considering the overall process, the first verification that is perfomed concerns the well-formedness of the component model, expressed in OCL, which is checked before the translation

[4] Function *diff* returns the relative complement of two sets.

into CSP. After the translation, we split the formal verification into two major steps: the first one checks the properties that must be met by a BRIC contract, and the second step verifies whether the connections between the components preserve deadlock freedom after the compositions.

A BRIC contract verification includes a set of assertions that confirms some characteristics of a component as checking if its behaviour conforms to an I/O Process, described in Section 2.3. Table 4 presents some of the mechanised verification of I/O Process characterisation. [5] [6]

Table 4 – Assertions

I/O Process Characterisation	CSP assertion
I/O Channels	assert not Test(inter(inputs(P),outputs(P)) == {})[T= ERROR
Infinite Traces	assert not HideAll(P):[divergence free [FD]]
Divergence Free	assert P:[divergence free [FD]]
Input Determinism	assert LHS_InputDet(P) [F= RHS_InputDet(P)
Strong Output Decisive	assert LHS_OutputDec_A(P) [F= RHS_OutputDec_A(P) assert LHS_OutputDec_B(P,c1) [F= RHS_OutputDec_B(P,c1) assert LHS_OutputDec_B(P,c2) [F= RHS_OutputDec_B(P,c2)

Source: Author's ownership.

When one of these refinements fails, FDR returns a counterexample in terms of a trace of events. We have created a mechanism where every event is traced back to the elements of the state machine at the UML level. Thus, the user can navigate the transitions of the state machine up to the point where the violation occurs. This helps in identifying the cause of the issue. As these properties are checked for each component in isolation, we do not need to worry about interactions with other components at this point. We provide an example of this kind of traceability in Section 5.

Once no violation is identified, and the protocol implementation is automatically generated, it is possible to verify if each connection preserves deadlock freedom.

Although the user submits the entire model to be verified, each component instance is translated independently into BRIC. More importantly, the verification of each composition is carried out by applying a BRIC rule in a compositional way, as already explained. Therefore, deadlock freedom is ensured by construction at the semantic (BRIC) verification level.

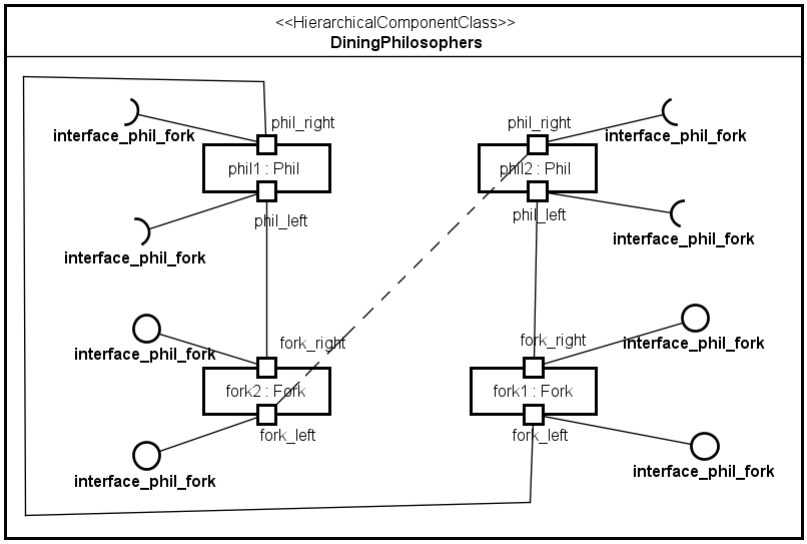
Particularly, there is no specific order to translate and verify a model such as that in Figure 19. However, in this example, the deadlock will always be found when the last connection

[5] The function *inter* returns the events of the intersection between both process events.
[6] Infinite traces are checked by asserting that hiding all events (*HideAll*) introduces divergence.

is processed, since this is the one that entails a cyclic communication topology. Suppose the dotted line between the component instances $fork2$ and $phil2$ is the last one to be processed, causing a deadlock in the Dining Philosophers. When a deadlock is found, FDR yields a trace with the events that led to the deadlock. We convert this result to a sequence diagram, in which the component instances are represented by lifelines where messages are exchanged. In this way we provide the traceability back from the formal specification to the UML view.

This traceability mechanism is different from the previous one because now we are concerned with the connections between the different components. Therefore, as the purpose of a sequence diagram is to illustrate interactions between elements, it provides better visualisation than navigating through state machines.

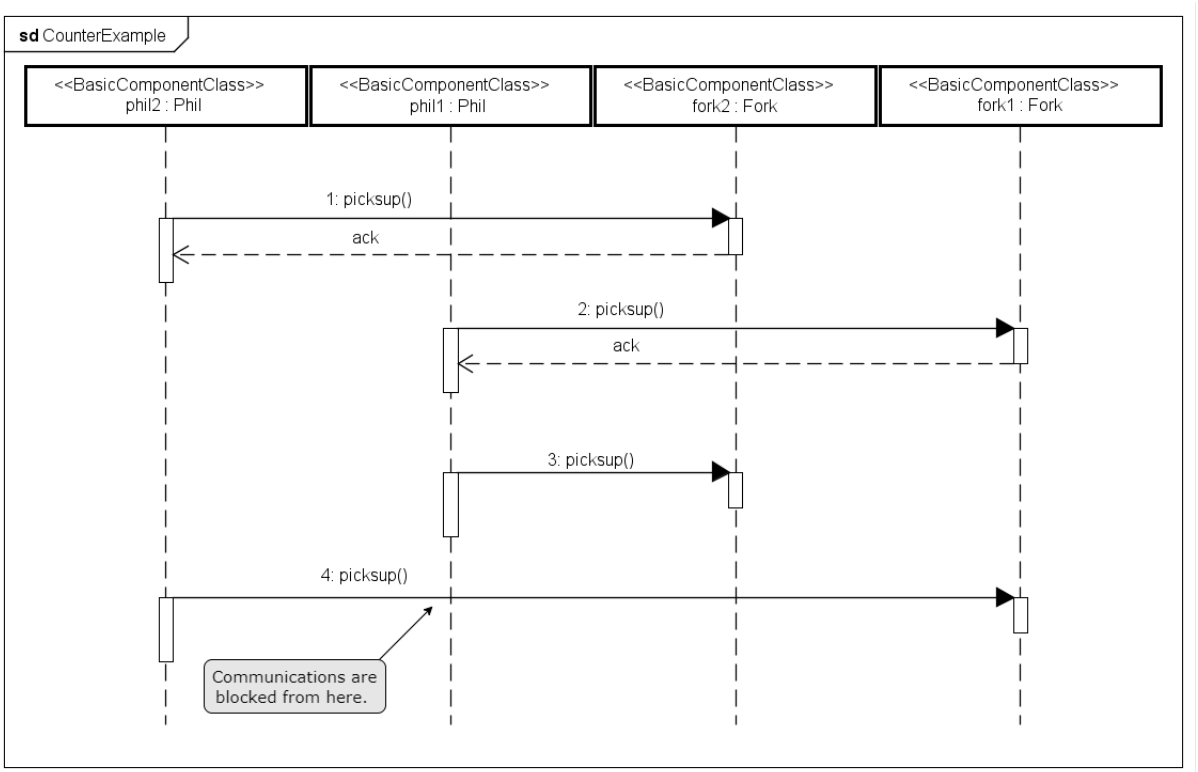
Figure 19 – Deadlock



Source: Author's ownership.

Figure 20 shows the traceability of the deadlock situation as a sequence diagram that shows component instance interactions arranged in time and the respective messages exchanged. Lifelines represent the component instances with the respective component type and stereotype; for example, the lifeline *phil2* is an instance of *Phil*, a *BasicComponentClass*, and *fork2* is an instance of *Fork*. Between these two instances, there is a message exchange *picksup* from *phil2*, that requires this service to *fork2* to provide. The instance *fork2* sends an *ack* message to *phil1* indicating that the communication was successful. However, between the instances *phil1* and *fork2* there is no *ack* message, indicating that there is a problem in the communication, in this case a deadlock. This view can help the user to identify where the flaw is and fix the problem.

Figure 20 – Automatically generated deadlock trace as a Sequence Diagram



Source: Author's ownership.

These different traceability mechanisms help users in identifying flaws in their models and fixing them. The primary purpose here is to refrain the user from being aware of the formal aspects of the approach through the application of hidden formal methods (HORVÁTH et al., 2020).

5 TOOL SUPPORT AND CASE STUDIES

To support the modelling of components according to the proposed meta-model and translate these models to BRIC contracts described in CSP, we have developed a *plug-in* to the Astah modelling environment (CHANGE, 2020), 21. Astah has been chosen due to the following reasons: its extension capabilities facilitate the creation of plug-ins; models can be created using several UML elements and diagrams, which allows us to reuse the notation 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 easy integration with other tools. In our case, we need to integrate with FDR for the purpose of mechanised verification. This plug-in was developed in java and we reuse libraries, with assertions, from (PEREIRA; OLIVEIRA; SILVA, 2016)

Creating models using Astah is considerably intuitive for UML practitioners. With the plug-in, while creating a model, the user may choose to check if the model is deadlock free by clicking a button. This triggers a list of tasks on the model. First, the component model well-formedness conditions are checked; next, the CSP specifications related to the components and their relationships are generated. Afterwards, each component (BRIC Contract) is verified concerning its adherence to an I/O process, that is, the component is semantically well formed; and then, its protocol is automatically generated. Finally, each connection between components is verified in a compositional way. The tool maintains traceability of the UML models in the generated CSP semantics. Therefore, given that a deadlock is identified, the user is notified via UML diagrams and refrained from needing to interact at the CSP semantic level. In this way, the tool support we provide indeed implements the concept of hidden formal methods.

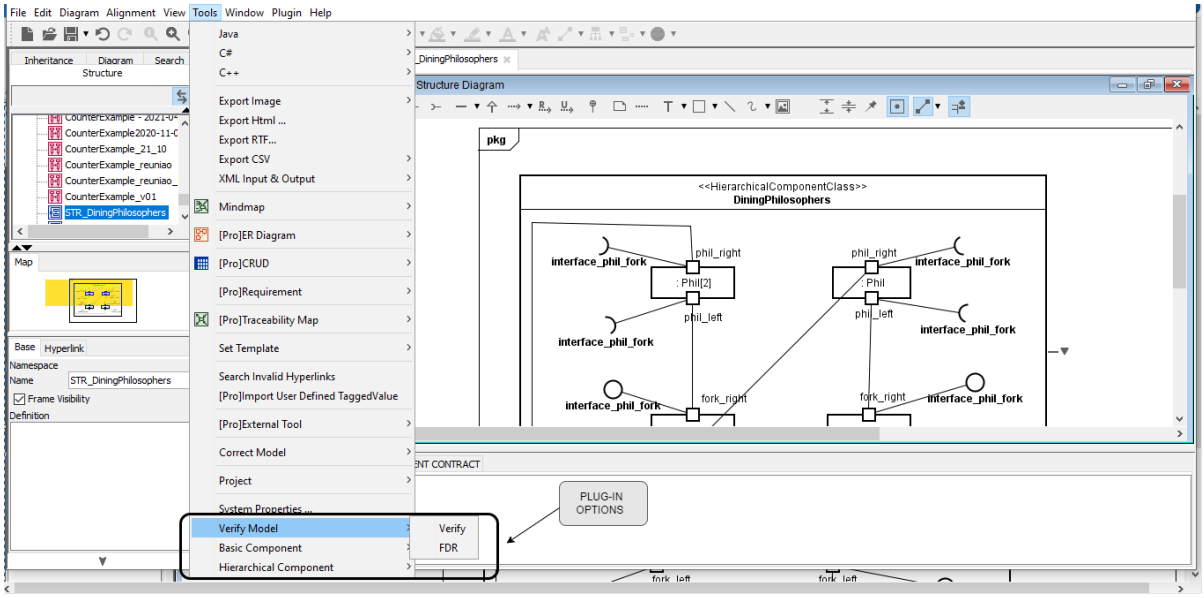
Our plugin supports the creation of basic and hierarchical components. For example, when the user creates a new basic component, a collection of related ($empty$) diagrams is automatically created. This includes a state machine diagram, a composite structure diagram and a class diagram. Afterwards, the user can edit these diagrams to define the complete model of the component. At any time, the user can press a button to verify the component model, which is carried out automatically and with traceability to the model, as already explained.

After the translation of the component model into BRIC, based on the rules presented in Section 4.2, the verification is divided into two phases. The initial step of the formal verification concerns the contract of a component, to check whether the contract is valid concerning the

BRIC conditions related to the notion of an I/O process. If the component contract is not valid, the tool exhibits an animation of its state machine execution where it is possible to navigate and realise the exact point that causes the problem. Otherwise, the tool runs FDR in background to check whether each composition between components is deadlock-free. If the verification fails, the problem is traced back to the UML component level; we show a counterexample as a sequence diagram that can help the user to understand the issue and repair the component model.

We previously used the Dining Philosophers as our running example. In order to show more features of our component model and of the strategy for translation and automatic verification, we present two other examples: a Ring Buffer (Section 5.1) and a Leadership Election (Section 5.2).

Figure 21 – Astah Plug-in



Source: Author's ownership.

5.1 RING BUFFER

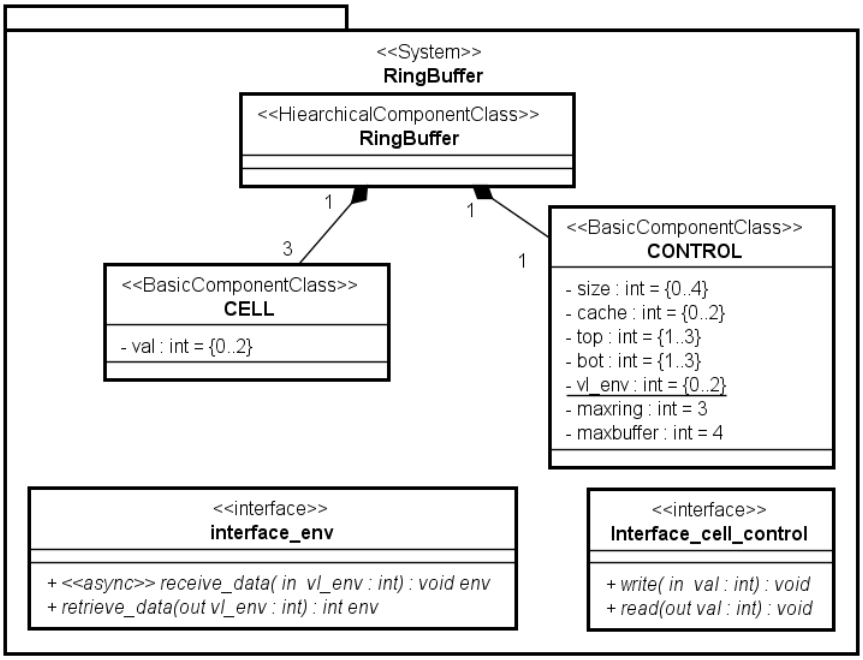
A Ring Buffer is a reactive bounded buffer composed of a ring of storage cells with a controller and a cache. It is a circular queue with FIFO data characteristics. A Ring Buffer is used for memory with a restricted size. It is found in many embedded systems and can be used to control multiple requests for a single resource such as memory, modems and printers.

A Ring Buffer has a cell for storing cached data and a number N of cells to store additional data. This first-in-first-out mechanism of storage keeps the current data to be output in a cache

slot, and the rest of data is stored in the cells. These cells act as slots of a circular list; the buffer keeps the information about which cell is at the top of the list and which one is at the bottom. The top cell is the one that is going to store new data, and the bottom cell has the data to be output immediately after the data held in the cache slot is emptied.

In addition to this information, it also records the current size of the buffer, considering occupied cells and the cache slot. The information about the top and bottom is updated depending on inputs and outputs to the buffer. Moreover, the buffer refuses to input data if completely full, and refuses to output if empty. In addition to the ring, there is a controller that is responsible for storing input data in the appropriate cell, and it possesses the cache slot. The appropriate cell is chosen based on the information about the top and bottom indices of the buffer kept by the controller. The controller also keeps track of the size of the buffer.

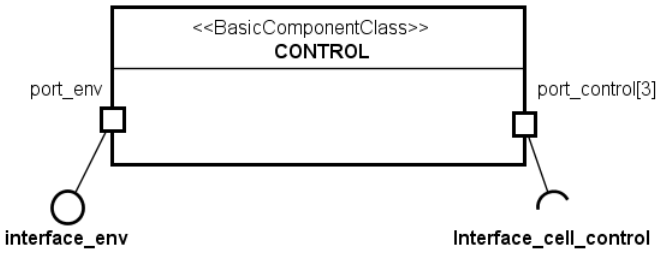
Figure 22 – Ring Buffer Component



Source: Author's ownership.

Part of the structure of the Ring Buffer component model in UML is shown in Figure 22. It is a *System component* composed of a $CELL$ component and a $CONTROL$ component. Both of them, $CELL$ and $CONTROL$, are of the type $BasicComponentClass$. These two basic components have attributes: the *CELL* component has an attribute *val* that stores values; the *CONTROL* component has the cache that stores the head of the ring buffer; the size of the list stored in the buffer; and two indices, bottom and top, to delimit the relevant values. It has two constants: the size limit of the buffer, *maxbuffer*, and the number of storage cells,

Figure 23 – CONTROL Component



Source: Author's ownership.

maxring, defined as $maxbuffer - 1$, which gives the bound for the ring considering the size of the buffer. Also, it has an attribute, vl_env, to store values received from the environment.

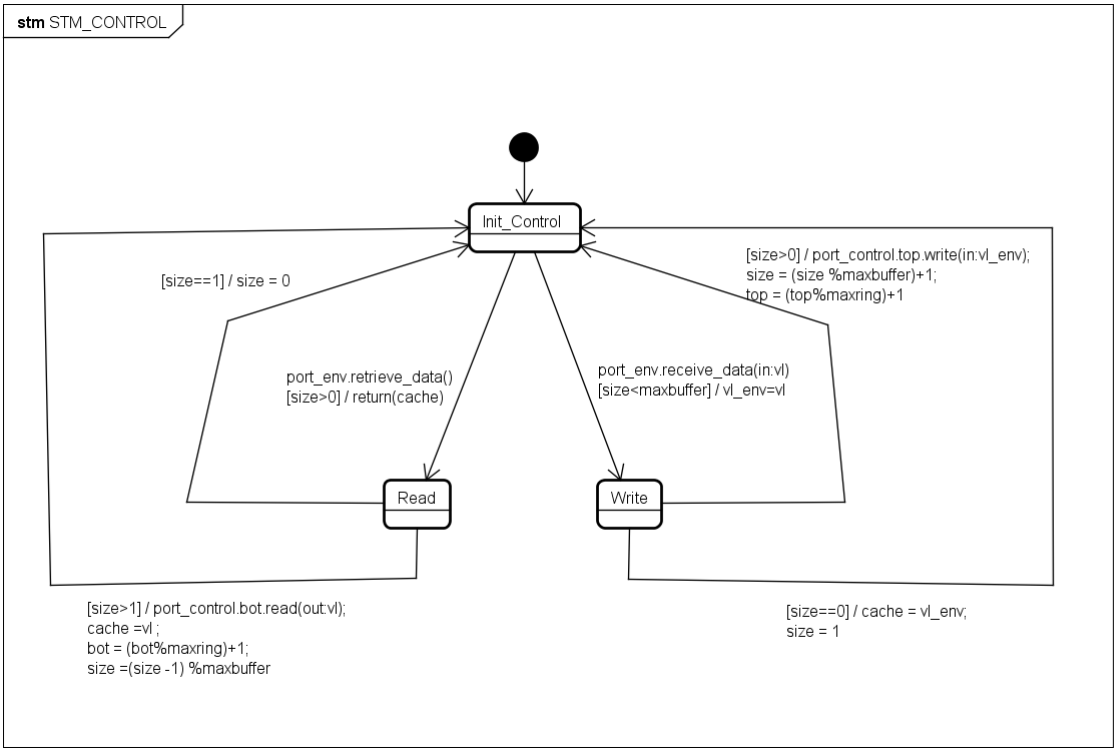
In order to allow communication between the controller component and the cells, a common interface is realised by their ports: $interface\_cell\_control$. While the $CONTROL$ port *port_control* requires this interface, the $CELL$ port provides it. The $interface\_ctr\_cell$ defines the operations: *write* and *read*, which are performed by the components. These operations have parameters: the first one has an *input* parameter that represents the value to be written in a cell, and the second one has an *output* parameter, the current value read from a cell.

Similarly, $interface\_env$ is realised by the $CONTROL$ port $port\_env$ component for the input and output exchanged with the environment. The operation $receive\_data$ of this interface is asynchronous, which means that the caller of the operation does not expect an immediate (if any) return from the caller. This communication is modelled in terms of signals. In the model, a signal is identified by the *async* stereotype; it also may have parameters as an operation. This interface has one signal *receive_data* and an operation *retrieve_data* with *out* and *in* parameters.

Figure 23 shows the basic component *CONTROL* (composite structure diagram perspective). It has two ports: *port_env* that provides the interface *interface_env* and the port *port_control* that requires the interface *interface_cell_control*. The latter has multiplicity 3, represented by $port\_control[3]$, which is indexed from 1 to 3, to establish the connections with the three cells.

The state machine diagram that describes the behaviour of the *CONTROL* component is shown in Figure 24. The *CONTROL* component is responsible for receiving input and output requests from the environment and for interacting accordingly with the ring of storage cells. In this way, the state machine has three states: *Init_Control*, *Read* and *Write*. The *Init_Control* state has a choice between *Read* and *Write* states where the *read()* and *write()* operations

Figure 24 – State Machine of CONTROL Component



Source: Author's ownership.

are handled, respectively. The decision between these two branches is made according to the trigger fired by the events communicated through the port *port_env* and the guard evaluation.

If *port_env.retrieve_data* is triggered (the environment requests some data) and the *size* of the buffer is greater than zero, that is, there is at least one element available to be read, the value in the cache is always communicated through the statement: *return(cache)*. After this, the state *Read* is reached.

The other possibility is when the event triggered is *port_env.receive_data* (the environment sends a data in order to be stored in the buffer) and there is space in the buffer; in other words, when the attribute *size* is less than the constant *maxbuffer*. In this scenario, the *write* state is reached.

In the *Read* state, when the size of the buffer is equal to one, it means that the cells do not store values, and the returned value was the one available in the cache (as in the previous transition). Now the cache should be empty; then, we set the *size* attribute to zero. On the other hand, when the *size* is greater than one, a new value is read from the cell indexed by *bot*, and the cache is updated. Also, the new value of the bottom and size of the buffer are updated, because the cell whose value was retrieved is now available to be written.

In the *Write* state, when the buffer is empty, i.e. the *size* is equal to *zero*, a value *vl_env*

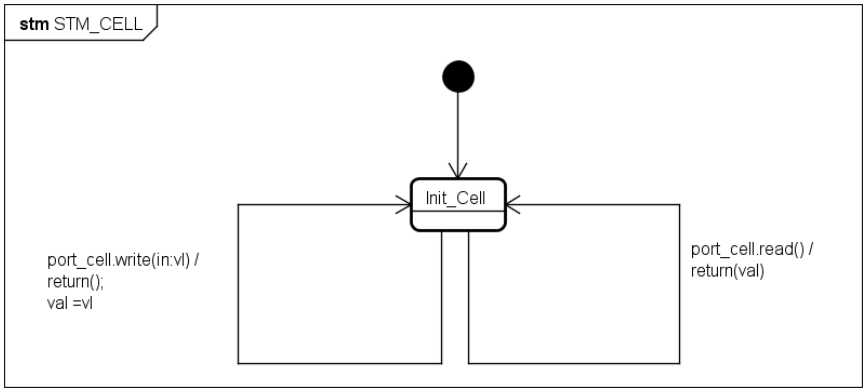
is cached. The ring indices do not change, and the buffer now contains a single item in the cache. If the buffer is not empty, the *CONTROL* sends the *vl_env* value to the ring along with the position *top* in which the value is to be stored. In this case, the *cache* is not changed, but the indices and the size of the ring are updated.

Since the elements of the *CONTROL* component are well-formed, the tool generates CSP files with the corresponding specification. This specification is available in Appendix C.

The process *CONTROL_memory* represents the memory of the component *CONTROL* whose attributes can be accessed and updated. This process has as parameters the attributes of the component. As previously shown in Section 4.3, each attribute is associated to channels *get* and *set*. This process is also responsible for evaluating the guards.

If a guard is associated with a trigger event, this is translated to a CSP guarded statement. For instance, the transition between the states *Init_Control* and *Read* has a guard $size > 0$ and the trigger event $port\_env.retrieve\_data()$ that returns the value of *cache*. We translate it as the CSP guarded statement $size > 0 \quad \& \quad port\_env.id.1.retrieve\_data!cache$, where the number 1 is an identifier of the pair guard and transition. As mentioned, transitions may have guards, which, besides the trigger, impose firing conditions to transitions. The memory process enables or disables particular events depending on whether the guard for the transition with the relevant identifier, 1 in this case, holds or not. In case there is no trigger associated with a transition guard, the statement is formed of a boolean expression, and the event *internal*, as explained in Rule 9 of Section 4.2. For instance, the transition between the states *Read* and *Init_Control* has a guard $size == 1$ and an action $size = 0$ that translates into the statement $size == 1 \quad \& \quad internal.3$. The event *internal.3* is used in *STM_CONTROL* in order to allow the synchronisation when the guard holds.

Figure 25 – State Machine of the CELL Component



Source: Author's ownership.

The behaviour of the memory process of the component *CONTROL* is described in Section 4.2 Rule 9.

The process *STM_CONTROL* is the outcome of the application of the rules described in Section 4.2 related to the state machine. Each state of the diagram becomes a process parameterised by the identifier *id* that represents a unique component instance. We use ellipses for better readability. The complete specification is available in Appendix C.

$$
\begin{aligned}
&STM\_CONTROL(id) = Init\_Control(id)\\
&Init\_Control(id) =\\
&\quad (port\_env.id.1.retrieve\_data\_I \rightarrow get\_control\_cache.id?cache \rightarrow\\
&\quad port\_env.id.1.retrieve\_data\_O?cache \rightarrow Read(id))\\
&\quad \square\\
&\quad ...\\
&Read(id) =\\
&\quad (internal.3 \rightarrow set\_control\_size.id!0 \rightarrow Init\_Control(id))\\
&\quad \square\\
&\quad ...\\
&Write(id) =\\
&\quad ...\\
&\quad \square\\
&\quad (internal.6 \rightarrow get\_control\_vl\_env.id?vl\_env \rightarrow set\_control\_cache.id!vl\_env \rightarrow\\
&\quad set\_control\_size.id!1 \rightarrow Init\_Control(id))
\end{aligned}
$$

The other component of the Ring Buffer is the *CELL* and its behaviour is modelled in Figure 25. The responsibility of this component is to store the value of a buffer cell and make it available when requested. It has one state: *Init_Cell* whose transition can be triggered by *port_cell.write* when the value vl is stored in the attribute val, or can be triggered by *port_cell.read* when the component yields the value currently stored in *val*.

As explained in Section 4.2, a $BasicComponent$ has its semantics defined by the structural and behavioural processes composed in parallel; the synchronisation set is formed of the union of the getter and setter events and the events used to communicate guard evaluations. All these events are hidden outside the parallel composition. Thus, the basic component *CONTROL* has its semantics given by the following process:

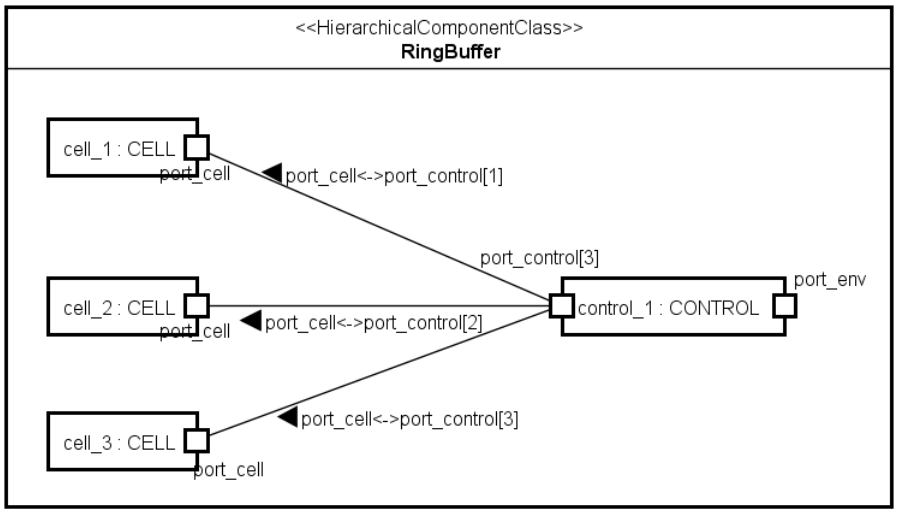
$$CONTROL(id) = STM\_CONTROL(id)$$
$$[|\ \{|\ get\_size.id, set\_size.id, get\_cache.id, \quad set\_cache.id, get\_top.id, set\_top.id, get\_bot.id,$$
$$set\_bot.id, get\_vl\_env.id, set\_vl\_env.id, internal, port\_env.id.2.receive\_data,$$
$$port\_env.id.1.send\_data\ |\}\ |]$$
$$CONTROL\_memory(id, 0, 0, 1, 1, 0)$$
$$\setminus\ \{|\ get\_size.id, set\_size.id, get\_cache.id, set\_cache.id, get\_top.id, set\_top.id,$$
$$get\_bot.id, set\_bot.id, get\_vl\_env.id, set\_vl\_env.id, internal\ |\}$$

The values passed as parameters to the process *CONTROL_memory* are the initial values of the component attributes; these correspond to the first value of the range defined in the class diagram in Figure 22.

Following the same rules, the basic component *CELL* is represented by the process *CELL*:

$$CELL(id) = STM\_CELL(id)[|\ \{|\ get\_val.id, set\_val.id\ |\}\ |]CELL\_memory(id, 0) \setminus \{|\ get\_val.id, set\_val.id\ |\}$$

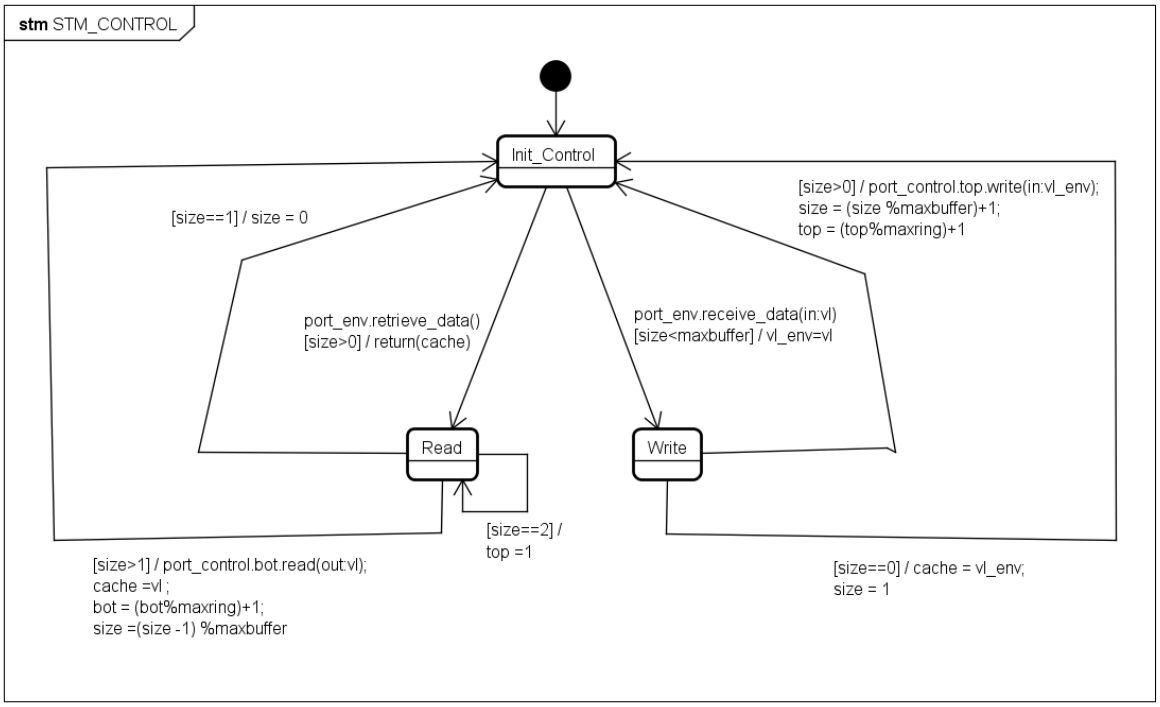
Figure 26 – Ring Bufffer Component



Source: Author's ownership.

After the basic components are translated, it is possible to make connections that allow communication between them. We consider the information provided in the Composite Structure diagram of the hierarchical component to define the corresponding BRIC-CSP semantics. Figure 26 shows an example configuration of the hierarchical component *Ring Buffer* that is composed of three instances of the *CELL* and one instance of the *CONTROL* component. To define how an indexed port of $port\_control$ of instance *control_1* is connected with a port of one of the instances of the $Cell$ component, a label is used in the connector, for instance: $port\_cell$ <-> $port\_control[1]$.

Figure 27 – State Machine of Control Component.

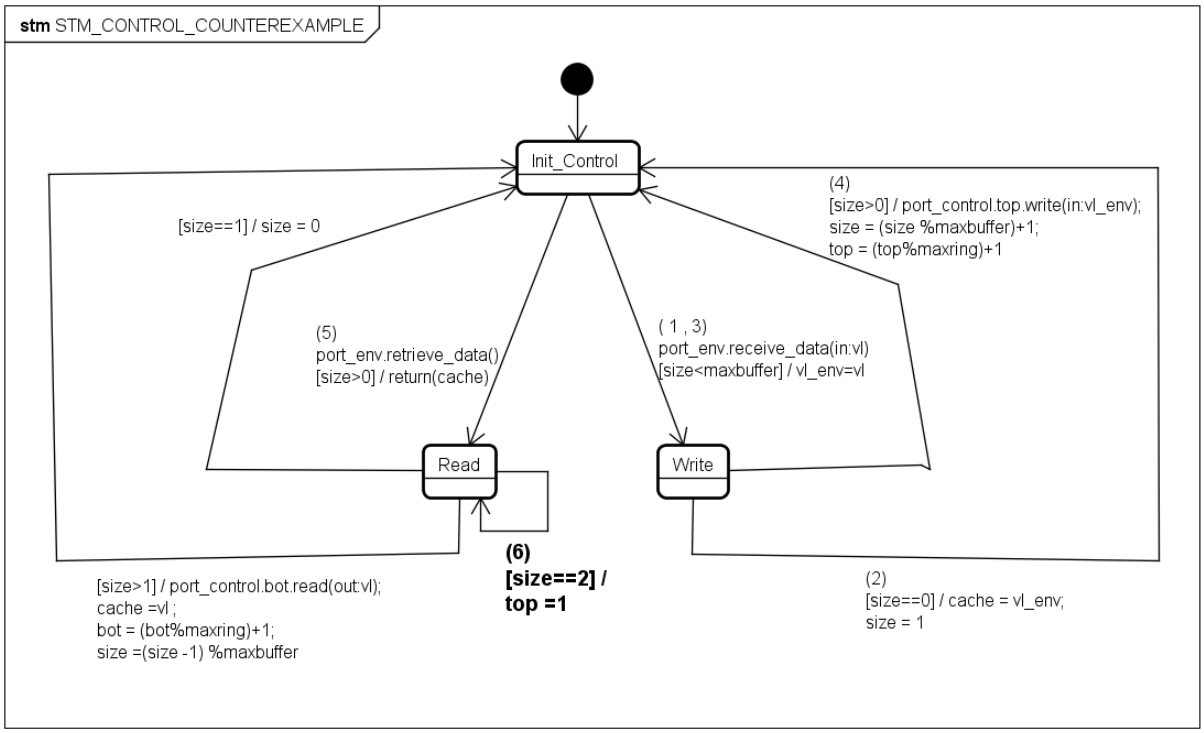


Source: Author's ownership.

Some errors can be introduced when designing a component. For instance, Figure 27 shows a state machine diagram of the component *CONTROL* that describes a behaviour with divergence, violating one of the required properties of an I/O Process (see Definition 4). The divergence is introduced by the self transition (in state $Read$) with guard $size == 2$ and action $top = 1$, as this can be indefinitely fired when the guard is true, and produces no visible effect. When this UML model is submitted to verification, this is automatically identified (verifying by checking assertion using FDR). In order to facilitate the identification of the issue, the tool creates a copy of the state machine diagram with numbered transitions (numbers between parentheses) to express the order of its execution in a counterexample. Note, in particular, that the enumeration format allows to recording multiple firings of a same transition. For instance, the transition from $Init\_Control$ to $write$ is fired on the first and on the third execution steps. This can be seen in the state machine diagram illustrated in Figure 28. The boldface transition indicates the one that introduces divergence. It also allows the user to navigate through the transitions and realise where the problem is.

Another design mistake identified is the structural problems that occur when component instances are connected. In the modelling of the basic component *CONTROL*, its singleton instance has one port named *port_control* with multiplicity three. This implies that the ma-

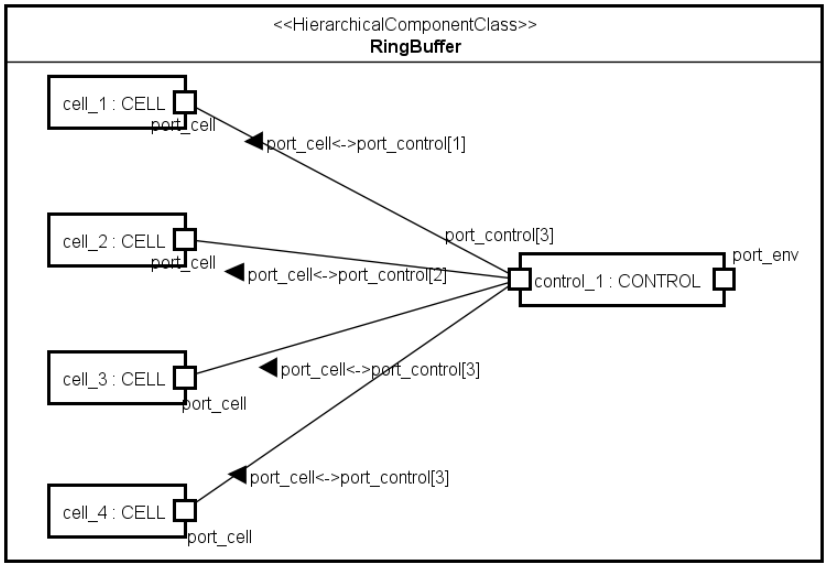
Figure 28 – State Machine with Divergence Counterexample



Source: Author's ownership.

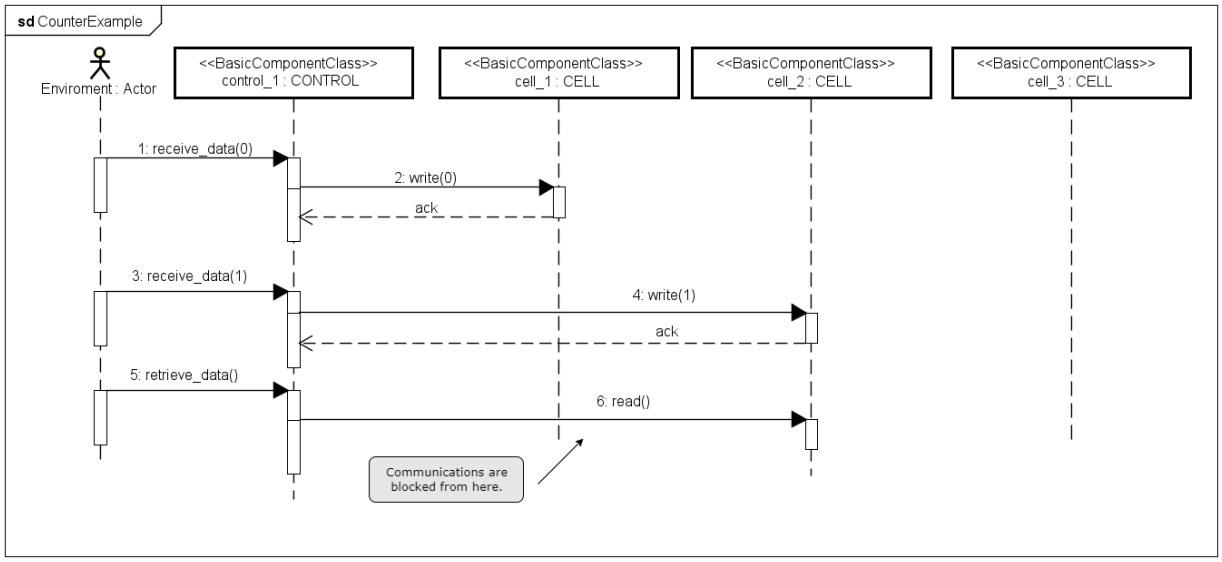
ximum number of cells that can be part of the RingBuffer is three. When a fourth cell is added to the model and connected to the control, a structural problem occurs in the system. In Figure 29 the fourth instance of *CELL*, *cell_4*, is connected to the port *port_control* in the index 3 that is also connected to *cell_3*.

Figure 29 – Ring Bufffer Component with 4 cells



Source: Author's ownership.

Figure 30 – Automatically generated Sequence Diagram with deadlock



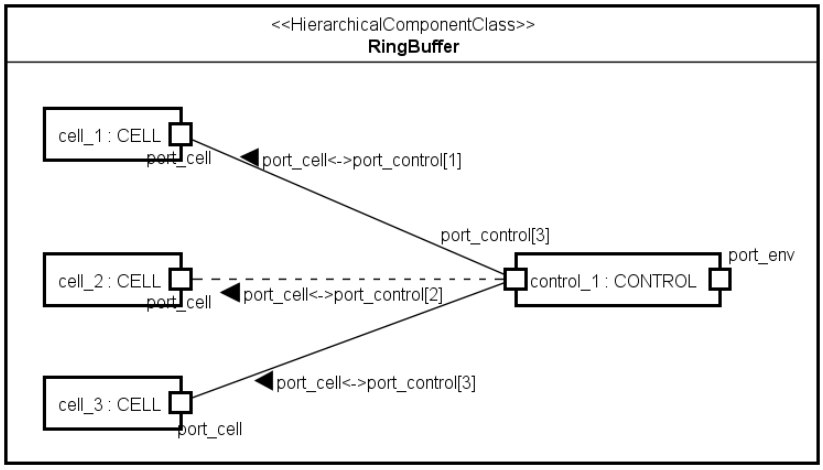
Source: Author's ownership.

Since the multiplicity of the port has been previously defined, it does not allow a connection to more elements. This type of structural verification is part of the well-formedness conditions that raises eventual problems to the user.

Now we describe an example of the introduction of deadlock due to a composition mistake. A possible problematic behaviour of the *CONTROL* component happens if replacing the guard $size > 1$ to $size > 2$ in the transition between the *Read* and *Init_Control* states. It obeys all I/O process criteria. However, when a *CONTROL* instance is composed with three instances of *CELL*, as shown in Figure 26, it produces a deadlock. The *CONTROL* instance, *control_1*, receives through the *port_env.receive_data* signal a value, which is stored; then the buffer size is incremented. When this event happens twice, the buffer size becomes 2. With this configuration *size ==2* and the *port_env.retrieve_data* operation being requested, the *cache* value is returned. However, in this scenario there is no path to proceed, thus, leading to a deadlock.

This problem is identified using FDR assertions, and a sequence diagram is automatically generated to illustrate the communication problem among the component instances, Figure 30. It is also exposed from another perspective through a composite structure diagram, Figure 31. A dotted line between instances *control_1* and *cell_2* evidences the connection in which the deadlock occurs.

Figure 31 – Ring Buffer Deadlock Traceability



Source: Author's ownership.

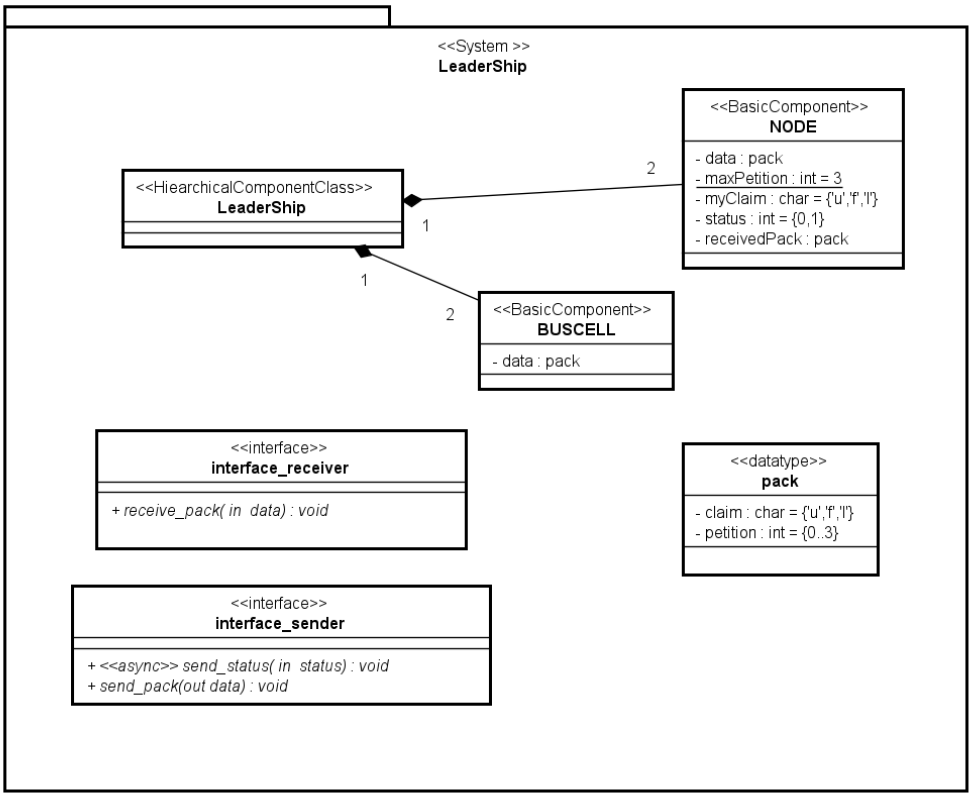
5.2 LEADERSHIP ELECTION

Another case study we explore is the leadership election. It consists of choosing a leader for a distributed system by an election process, which involves a network of participants. An example application is the audio and video system of Bang & Olufsen (B&O). In such a system, several devices are dynamically connected. Commonly, such a device could be a cellphone, a home theatre, a television, and so on. All these devices run in parallel and share information. When these participants notice the absence of a leader, they start an election process in which a leader is elected. One participant communicates with every other participant sending its internal state and receiving the internal information of the others. The state of the participant consists of a priority and a claim, which represents the current status of this participant in the network, either *undecided*, *leader* or *follower*. The leader is elected based on the priority of the nodes.

We model this problem as follows: two basic components are defined. One represents the participants, *NODE* component, and the other a transport layer, *BUSCELL* component. In this model, the participants exchange messages via the transport layer and recursively send messages to all their peers and receive messages from them. We model a concise leadership election that has two devices.

Figure 32 shows the *LeaderShip* System composed by the basic components *NODE* and *BUSCELL*. As participants interact sending by messages from one participant's transmitter to another participant's receiver for a 2-Node configuration the number of instances of $NODE$

Figure 32 – Leadership Election Component



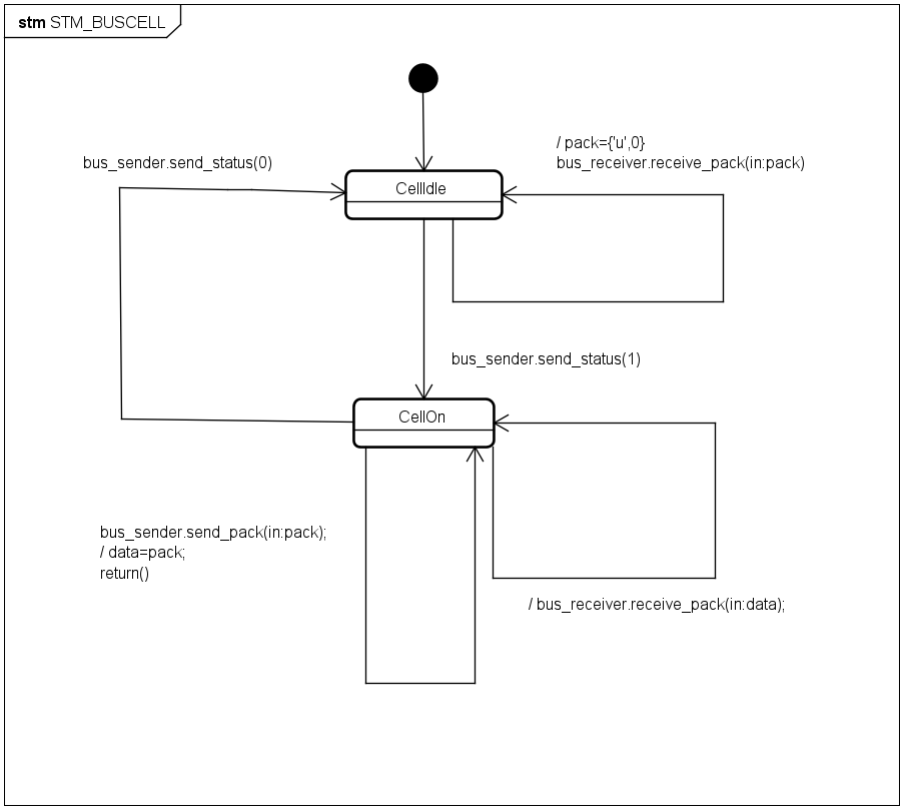
Source: Author's ownership.

and $BUSCELL$ is the same; in our example, two. The ports of *NODE* and *BUSCELL* realise two interfaces: *interface_receiver* and *interface_sender*. The former provides an operation that receives data from its peers, *receive_pack*. The other interface has the asynchronous operation, *send_status*, that allows a participant to send its status; and the operation *send_pack* that allows one participant to send data to other participants.

Figure 33 shows the state machine of the *BUSCELL* component. The responsibility of this component is to provide an unidirectional communication channel between a pair of *NODE*s. This state machine has two states: *CellIdle* state that has a self transition which can be triggered to transfer a node timeout by *bus_receive.receive_pack(pack)*, where the claim is 'u'(undecided) and the petition is 0, and another transition that is triggered to identify an online status from a node by *bus_sender.send_status(1)*; the target of this transition is the *CellON* state. In this state, a pack of information is received from one node by *bus_sender.send_pack(pack)* and transferred to another one by *bus_receiver.receive_pack()*.

Figure 34 shows the state machine with the behaviour of the *NODE* component. Each node is initially turned off, state *OFF*; in this state, it can only turn on. Before turning on,

Figure 33 – BUSCELL Component

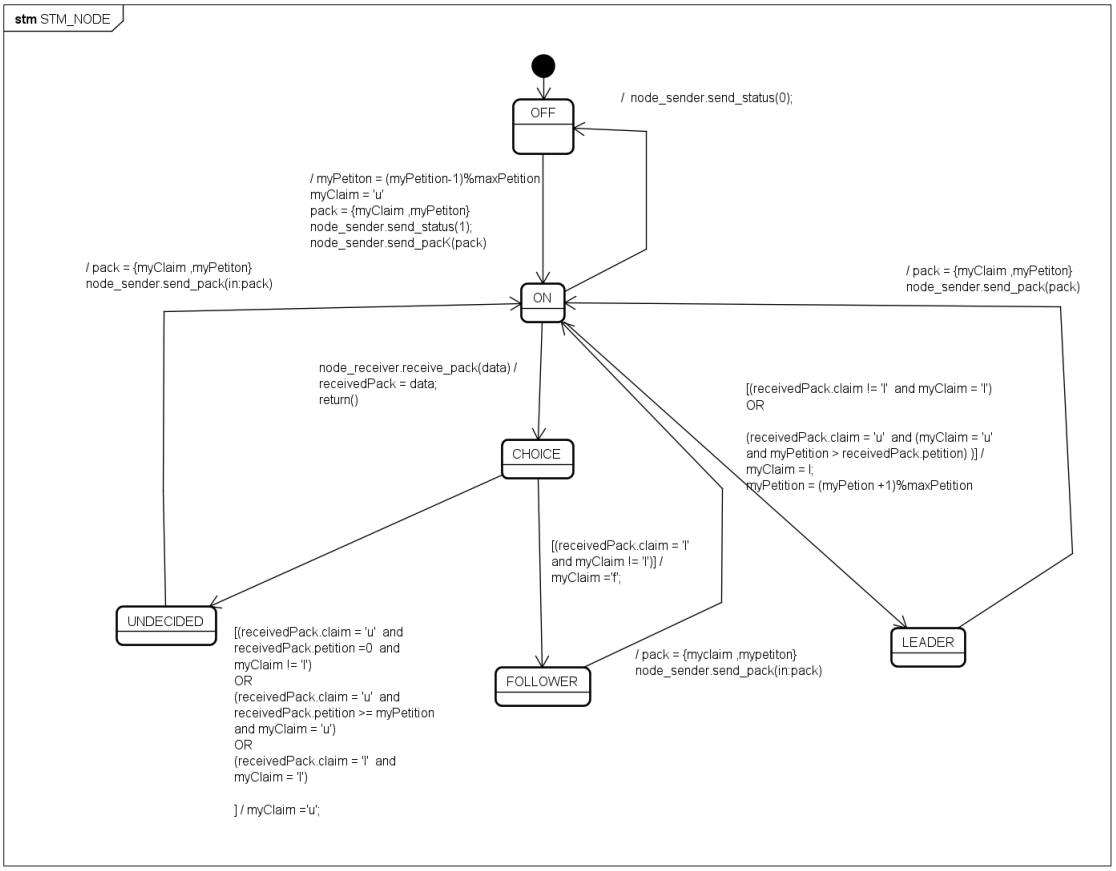


Source: Author's ownership.

the node reduces its petition, sets its claim to undecided ('u'), and sends its status and the pack with its petition and claim to the bus component ($node\_sender.send\_status(1)$ and $node\_sender.send\_pack(pack)$). Then, reaching the *CHOICE* state, the node waits for the information from the other node ($node\_receiver.receive\_pack(pack)$) and updates the attribute $received\_pack$ with the received data. When in the state $Choice$, the role of the node is determined: *undecided ('u')*, *follower('f')* or *leader('l')*. This decision is given by the petition of a node and its claim. For instance, if a $NODE$ verifies that there is no leader and its petition is greater than other petition´s node, the current node changes its claim to $leader('l')$, and another decision can start, recursively.

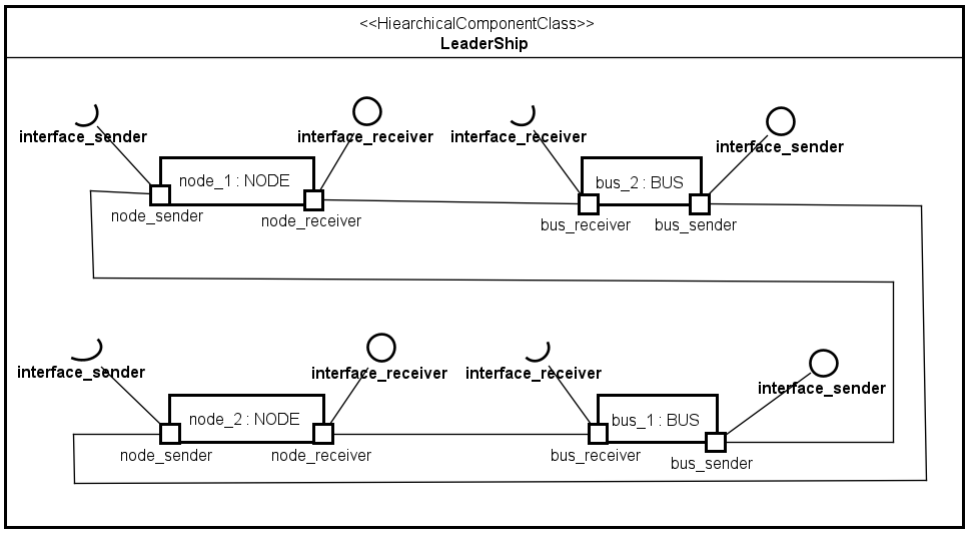
The communication among the instances of these components is illustrated in Figure 35. In this diagram, there are two instances of the *NODE* component that its ports provide the interface *interface_receiver* and require *interface_sender*, and two instances of the *BUSCELL* component that its ports provide the interface *interface_sender* and require *interface_receiver*. Connections are made between conjugated ports. For example, instance *node_1* is connected via its port *node_sender* with *bus_1* via *bus_sender*. This connection allows *node_1* to send information to *bus_1* that itself communicates with *node_2* via *node_receiver*.

Figure 34 – NODE Component



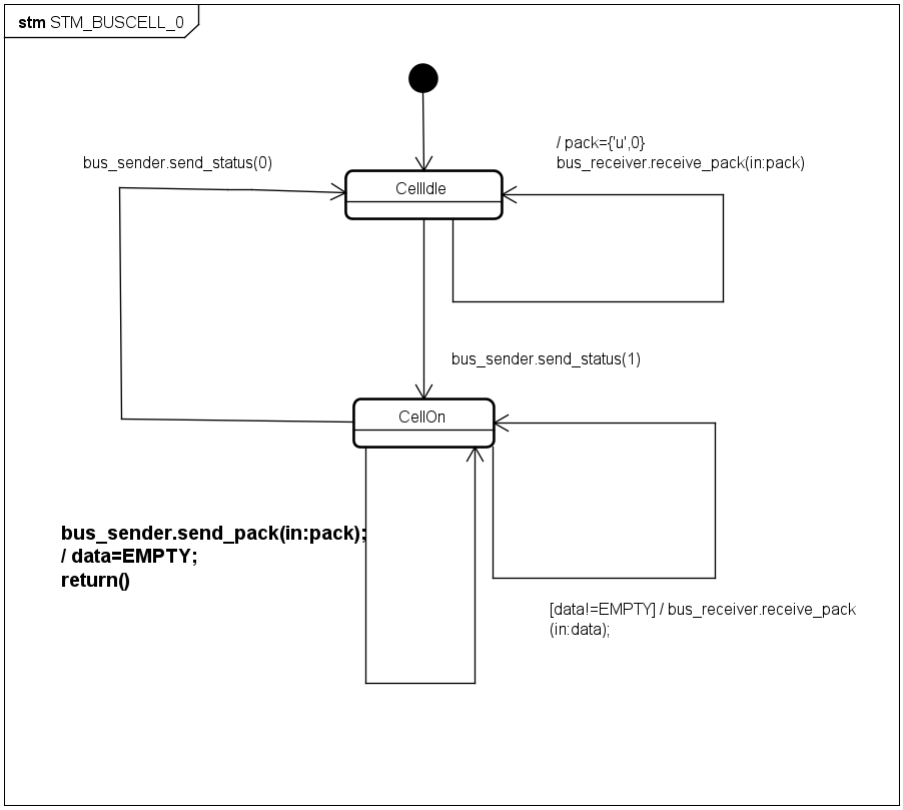
Source: Author's ownership.

Figure 35 – Leadership Election Composition



Source: Author's ownership.

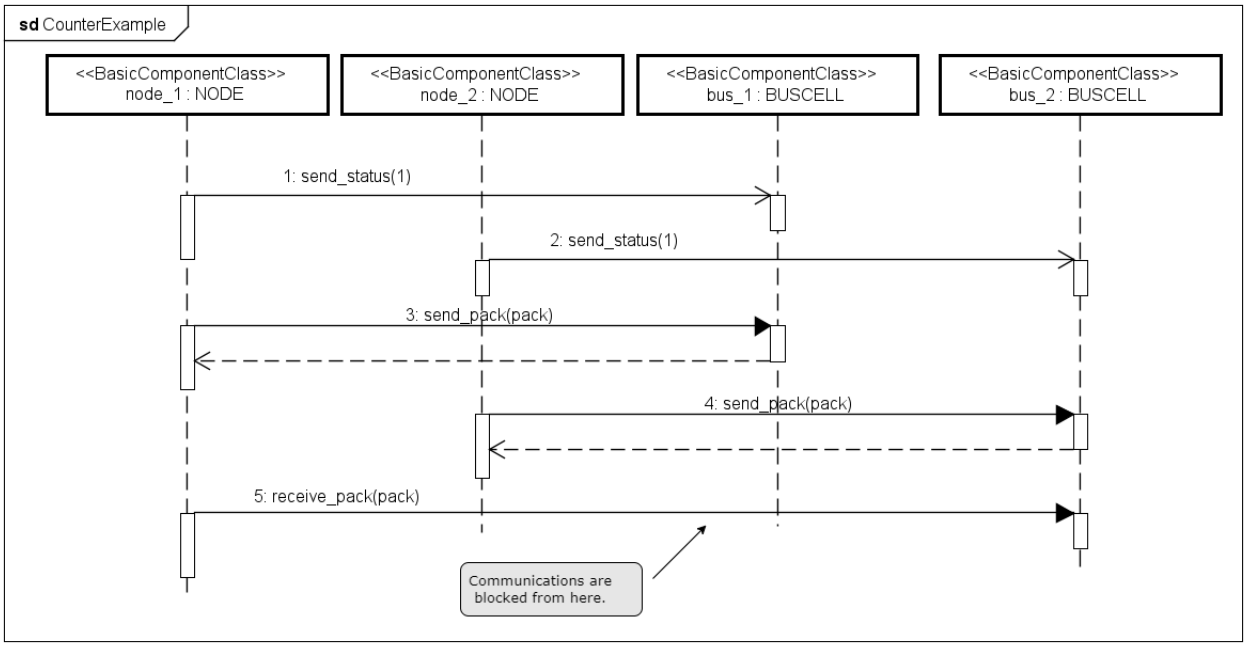
Figure 36 – Component with deadlock



Source: Author's ownership.

Even though each individual component is proved to be an I/O Process, their composition can result in a deadlock. To illustrate the detection of potential deadlock situations in the model, consider the addition of a new pack value, *EMPTY*, a new action, *(data = EMPTY)*, in the self-transition of *CellOn* state that can be triggered by *bus_sender.send_pack*, and a new guard in the self-transition of *CellOn* state has the action *bus_receiver.receive_pack*. This transition has its trigger and action marked in boldface in Figure 36. In this context, a node can send information to the *Buscell*, *send_status(ON)*, and then the *BusCell* moves from *CellIdle* to the *CellOn* state. When the *send_pack* is triggered, *data* receives *EMPTY*. In this way in the *CellOn* state, the *Buscell* is not allowed to receive information from a node since the guard will never hold in the sef-transition that triggers *receive_pack*. This malfunction is traced back as a sequence diagram, as shown in Figure 37.

Figure 37 – Sequence Diagram with deadlock



Source: Author's ownership.

5.3 SCALABILITY

Scalability is always a major concern of verification approaches. At present, the CSP models (generated from their input UML component models) do not scale well. For example, for the Dining Philosophers, we considered two configurations: one with five $phil$ instances and five $fork$ instances; and another configuration with seven instances of each component. The deadlock verification of the former configuration has taken 4 seconds using FDR; the latter has taken about 50 minutes. We considered a 2-Node configuration for the Leadership Election component, which has taken about 5 seconds to complete; for a 3-Node configuration, the verification is completed in about 3 hours. The experiment was executed on an Intel core i7-65000, 2.66GHz with 16GB ram.

In our approach, we decided to separate the concern of a formal (CSP) semantics for the proposed UML component model from that of the performance of deadlock verifications. Based on the results achieved in (OLIVEIRA et al., 2016), we will address scalability using metadata that record relevant component information (like protocol compatibility) so that these are reused in progressive compositions. Another promising direction to improve verification efficiency is to adopt behavioural patterns that impose behavioural and structural restrictions on a process network, as a means to guarantee deadlock freedom (ROSCOE, 1997). The experiments in

(OLIVEIRA et al., 2016) show that these can significantly improve scalability. For example, the deadlock verification of 10.000 forks and the same number of philosophers took 5 hours, and the verification of a leadership election instance with ten fully connected nodes (which involved 180 compositions) took about 2.5 hours.

6 RELATED WORK

There are several approaches to define component models and verification strategies, which are based on a variety of formalisms and each one has its benefits and deficiencies, depending on the context in which it is analysed. In this chapter, we will consider some of the closest related to our work.

For instance, the Foundational UML Subset (fUML) (Object Management Group (OMG), 2017b) provides a precise semantics for UML classes, activities and actions. The operational semantics of fUML maps UML activities to an executable model with methods written in Java. The declarative semantics of fUML is specified in first order logic and based on PSL (Process Specification Language) (GRÜNINGER; MENZEL, 2003). Precise Semantics of UML State Machine (PSSM) (Object Management Group (OMG), 2019b) is an extension of fUML that defines a foundational semantics for UML state machines. Precise Semantics of UML Composite Structure (PSCS) (Object Management Group (OMG), 2019a) is another extension of fUML for dealing with UML composiste strucuture diagrams. However, these works do not embrace component concepts and they focus on specifying a rigorous semantics with a test suite to demonstrate semantic conformance. In our case, the rules presented in Chapter 4 define a CSP semantics for the component model we propose that follows several aspects defined in the fUML work. In addition to these semantic rules, we provide a systematic and automated verification support.

The framework rCOS (CHEN et al., 2009; CHEN; MORISSET; STOLZ, 2010) is a refinement calculus for the design of object and component oriented software systems, in which a component is an aggregation of objects and processes with their interfaces. A use case is taken as a component, and the functionalities of the use case are modelled as methods in the provided interfaces. The functionality of each method of the interface is specified by preconditions and postconditions, and the order of interactions, between an actor and a component, as a set of traces of method invocations, graphically represented by a sequence diagram. A State machine describes how the system internally changes states during execution. Refinement properties can be verified using laws provided by rCOS. Like in our approach components can be composed hierarchically, and the compatibility of the compositions can be checked by using CSP and FDR. Unlike our approach, however, the formal notation is not completely hidden from the user; there is no traceability to the model.

We can also cite BIP (Behaviour, Interaction, Priority) (BONAKDARPOUR et al., 2012; BASU; BOZGA; SIFAKIS, 2006) which is a modelling framework supporting the formal definition of heterogeneous systems. It uses the BIP language and an associated toolset supporting the design flow. The BIP language allows building complex systems by coordinating the behaviour of a set of atomic components. Behaviour is described as a Petri net extended with data and functions described in C. BIP has an operational semantics, which describes the behaviour of both atomic and compound components. The behaviour of atomic components is based on a rigorous transition system model; thus, formal verification of invariant properties and deadlock-freedom is also supported. In (CHEHIDA; BAOUYA; BENSALEM, 2021) the authors present an extension that combines UML and BIP, where UML models are translated into BIP. State machine specifies the behaviour of the system, and the component diagram is used to define the system architecture. Also, they use Statistical Model Checking to verify temporal properties. However, the semantic translation from UML to BIP is presented in an informal way. Also, no well-formedness conditions for the UML model are presented. Finally, no traceability is available and the communication among components is only synchronous.

The strategy described in (GRAICS et al., 2020) provides a formalisation of the Gamma Composition Language that is part of the Gamma Framework (Molnár et al., 2018). It is a modelling tool supporting the hierarchical design, implementation and verification of reactive systems. A component is defined by a statechart that can be composed using the Gamma Composition Language. The three distinguished composition modes support synchronous, asynchronous and cascade; the latter stands for a pipeline-like behaviour. It also provides a Java code generator for the implementation of composition-related code and applies the UPPAAL model checker for formal verification and test generation. The queries representing the properties to be checked on the model are either given directly as UPPAAL temporal logic formulas or constructed using fillable patterns (for the most frequent safety and liveness properties). In (HORVÁTH et al., 2020), which is an extension of (GRAICS et al., 2020; Molnár et al., 2018), the authors proposed a cloud-based, end-to-end verification workflow for SysML (Object Management Group (OMG), 2017a) State Machines and reachability properties using an intermediate language and different model checkers. Model checking is fully automated, and traceability is provided through back annotations of the resulting trace. In this way, formal aspects are hidden from the users, as in our case. Nevertheless, our approach verification is compositional and we focus on deadlock freedom, while their work deals with reachability properties.

In (BREU et al., 1997), they have proposed a formal language called System Model to

specify object oriented systems in the style of UML. A System Model specification has a pre-defined mathematical structure comprising object identifiers, message passing, behaviour, communication histories, states, and so on. A UML diagram is modelled as a projection of a System Model, which is regarded as a complete and unified model of the entire system. Class diagrams, state-machine diagrams and sequence diagrams can be translated to a System Model. On the other hand, although the semantics of these diagrams is defined in a single formalism, the verification of the consistency between the diagrams and the development of tools have not been reported.

In (LIMA et al., 2017), the authors present a formal semantics for a comprehensive subset of SysML (Object Management Group (OMG), 2017a) via a mapping into CML (WOODCOCK et al., 2012), a formalism that combines CSP and VDM (FITZGERALD; LARSEN, 2009). The work proposes guidelines of usage for the construction of meaningful SysML models; a state-rich process algebraic semantics for SysML models, in particular, CML semantics; and applications of the CML model in reasoning at the diagrammatic level. These guidelines 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 (LIMA et al., 2017) is not on component-based design nor on ensuring property preservation compositionally.

Collaborating SysML state machines are used to model systems in (GIBSON et al., 2014), which is translated to Java. The work uses the Java Path Finder (JPF) model checker that provides a basis for checking fault protection design against a defined failure space and enables validation of the logical design against domain specific constraints. When a logical assertion is negated, counterexample trace is output in a textual form, but this is not traced back to the SysML model. Furthermore, in this work, there seems to be no conformance notion for components or compositions.

The ProMoBox framework (MEYERS et al., 2014) enables the automatic verification of Domain Specific Modelling Languages (DSML). Temporal properties are translated to LTL and Promela. These properties are checked by SPIN. The verification results in the case of a counterexample are translated back to the DSML level. Its concrete syntax is defined graphically, and its semantics is given by the transformation model. On the other hand, the verification support does not include analysis of deadlock, and it is not component-based.

In (BESNARD et al., 2019) the authors address a run-time monitoring and formal verification of UML models of embedded systems. Safety properties can be expressed directly in UML

using the same UML subset as the one used for the system model and an extension of the C action language. The verification of these properties is made using model verification with the OBP2 model-checker, and results are compared with verification results of identical properties expressed in LTL (linear-time temporal logic). However, there is no notion of traceability, and its verification does not include analysis of deadlock.

In (DIAS et al., 2020) the authors present an approach that provides a translation from a SysADL architectural description to a CSP-based formal semantics and verify properties such as deadlock freedom, livelock freedom, and consistency among the structural, behavioral, and execution viewpoints of the model. SysADL is a specialisation of SysML to the software architectural description domain, where components are defined as blocks (Block Definition Diagram) and the specification of the behaviour is of the elements described in the structural viewpoint. However, (DIAS et al., 2020) do not provide traceability nor compositional reasoning and it focus on the conceptual architectural model while our work focus on the design phase.

In (PEREIRA; OLIVEIRA; SILVA, 2016) the authors present a tool, BTS (BRIC Tool Support), which provides textual support to create and compose BRIC components and an analysis of the components and their compositions. However, the input to the tool has the form of a specific textual template. There is no support for UML and no notion of traceability.

Some works provide model specification for a specific domain. RoboChart (MIYAZAWA et al., 2017) and RoboSim (CAVALCANTI et al., 2019) have a graphical language that use statechart models tailored to describe design of robotic applications. These models can generate CSP specifications automatically. Its formal semantics allows a verification of the designed systems. However this model is not component based. In (MARTINEZ et al., 2021), they present a component model for embedded systems. It defines elements that are relevant for the domain in form of libraries. It is a graphical modelling language offering known abstractions such as classes and relationships among these components. The customization of these abstractions is based on the UML concept of a profile. Assertions are made with OCL and Othello. This work does not have a formal semantics.

Table 5 – Summary of related works.

	Component Based	Approach to Verification	Modelling Language	Hierarchical Composition	Well-Formedness Conditions	Visual Traceability	Formalism	Properties
fUML (Object Management Group (OMG), 2017c)			UML				PSL	
Breu et al. (BREU et al., 1997)		A Posteriori	UML				System Model	refinement
rCOs (CHEN et al., 2009)	✓	A Posteriori	UML	✓			CSP	refinement and deadlock
BIP (BONAKDARPOUR et al., 2012; BASU; BOZGA; SIFAKIS, 2006; CHEHIDA; BAOUYA; BENSALEM, 2021)	✓	Compositional	Petri Net extend UML	✓			Invariants	Invariance, temporal properties and deadlock
Gamma (GRAICS et al., 2020; Molnár et al., 2018; HORVÁTH et al., 2020)	✓	A Posteriori	UML/SYSML	✓	✓		UPPAAL	liveness
UML-RT (SELIC; GULLEKSON; WARD, 1994)			UML-RT	✓				
BTS (PEREIRA; OLIVEIRA; SILVA, 2016)		Compositional					CSP	deadlock
Lima et al. (LIMA et al., 2017)		Posteriori	SysML		✓	✓	CSP	refinement
Gibson et al. (GIBSON et al., 2014)		A Posteriori	SysML			✓	JPF	fault protection
ProMoBox (MEYERS et al., 2014)		A Posteriori	DSML			✓	SPIN and Promela	temporal properties
Robochart (CAVALCANTI et al., 2019, 2019)		Compositional	UML		✓		CSP	refinement
ArmAssist (MARTINEZ et al., 2021)	✓	Compositional	UML		✓		Invariants	safety properties
Observer Automata (BESNARD et al., 2019)		Compositional	UML		✓		LTL	safety properties
Dias et al (DIAS et al., 2020)			SysADL				CSP	deadlock and livelock
Our Work	✓	Compositional	UML	✓	✓	✓	CSP	deadlock (compositional), livelock, input determinism, output decisiveness and non-termination

UML-RT (SELIC; GULLEKSON; WARD, 1994) is a UML profile that facilitates the modular development of software systems. Communication is modelled by means of input and output message exchange, which can be synchronous or asynchronous. The main UML-RT design elements are capsule, protocol, port and connector. The basic building block of a UML-RT model is a capsule whose behaviour can be defined using statecharts; the unique points of interaction are called ports, which are assembled by connectors and realise communication signals previously declared in a protocol. A capsule can contain parts, which are instances of other capsules; this, as in our case, hierarchical modelling is supported. Even though UML-RT provides constructs to model real time systems, its major drawback is the lack of a precise semantics, despite some attempts like, for example, that in (RAMOS; SAMPAIO; MOTA, 2005) where the authors present a mapping from UML-RT to the Circus process algebra. Also, UML-RT offers no explicit support to create more active classes from existing ones via composition rules.

Table 5 summarises the related work described in this chapter, as well as our own work. Each column represents a feature that we consider relevant of a component based Model, and these are used as a comparison basis. The *Composed Based* column indicates whether the work features a component model; the *Approach to Verification* one denotes the way the verification is made; the next column mentions the representation of the models. The columns *Hierarchical Composition*, *Well-Formedness Conditions* and *Traceability* indicate the presence or absence of these characteristics. *Formalism* shows the formalism used and the last column indicates which kind of property is addressed. Our work is distinctive in its definition of the component model based on well-formed conditions and formal semantics. Components are described as a UML profile; these components can be composed hierarchically. Several properties can be verified to check whether a component meets the required properties of an I/O Process, including livelock analysis. In terms of compositional analysis to ensure sound component integration, we currently focus on deadlock freedom. The formalism is hidden from the user and, when a counterexample is found, this is traced back and presented as UML diagrams.

7 CONCLUSION

In this work, we have presented a UML Component Model with a precise syntax defined in the form of a metamodel (as a UML profile), well-formedness conditions formalised as OCL constraints, and a formal semantics defined in BRIC, which itself has a process algebraic semantics defined in CSP. The proposed approach supports an incremental design and ensures the preservation of desired properties; we have focused on the constructive preservation of deadlock, but the framework also supports the verification of other properties like livelock freedom, input determinism, strong output decisiveness, and non-termination; these properties are required for a component to be adherent to an I/O Process.

The formal semantics is presented as a comprehensive set of denotational rules and auxiliary functions that map each metamodel element into its BRIC denotation. The behaviour of components, instances and connections are captured by CSP processes, and deadlock verifications, as well as the properties required for a component to be an I/O process, are conducted using the FDR tool.

We have also implemented the approach in the form of a plug-in to the Astah modelling tool. Using the plug-in, the CSP notation and the formal verification are hidden from the user. It ensures adherence to the metamodel and the related well-formedness conditions. The plug-in also implements the translation into BRIC.

When using the plug-in, if a verification fails, the problem is traced back to the UML component level, and the problematic composition is exhibited to the developer as a state machine diagram if the component, as a unit, has some unexpected behaviour. Complementary, if there are problems in the connections between component instances, these are exhibited as sequence diagrams.

We applied our approach to three case studies: the classical Dining Philosophers, a Ring Buffer and a Leadership Election protocol. They exemplify the modelling of basic and hierarchical components, with associated state machine and composite structure diagrams, including the connection of component instances. Concerning these examples, we have explored violations of well-formedness conditions, deadlock situations, violation of I/O process properties, and traceability to the component model.

Despite the promising results and the emphasised contributions, our approach has some limitations. The BRIC constraints may reduce the applicability of our approach concerning

modelling, since it requires that component contracts have an associated behaviour and it is not always the case in several components, but they are necessary to ensure the preservation of desired properties. As a major topic for future work we plan to explore is scalability of the verifications, based on metadata and behavioural patterns, as discussed in Chapter 5.

Our view in this thesis is that the translation of the UML component model elements into BRIC provides a formal semantics for these elements. In order to establish a notion of correctness for our translation, a semantics for UML is necessary; unfortunately, to our knowledge, there is no complete formal semantics for UML in the literature. A possible contribution in this direction is to use the fUML approach as a basis for proving correctness. In this way, it would be necessary to extend the fUML work to cover the semantics of all the elements of the proposed component model.

Our component model can be more expressive with the addition of advanced constructs in state machines, such as composed states; or with the addition of other model elements from UML.

Currently, our approach does not allow broadcast in the communication among components, as each composition rule is concerned with a pair of components or with two channels of a same component. In some applications, broadcast communication is certainly useful and we plan to consider this as future work.

As another future direction, we plan to adapt the approach proposed in (CAVALCANTI et al., 2018) for the construction of heterogeneous collections of components that are defined as patterns using generic (rather than concrete) instances. This allows one to parametrise a composite structure diagram with the number of instances involved in a system configuration, rather than being forced to statically determine a particular configuration.

Although our framework supports several properties required for a component to be an I/O Process, including livelock analysis, these are checked for the entire component (which can cause scalability problems in analysis); only deadlock freedom is verified in a compositional way. The proposed approach can also be extended in order to allow the compositional verification of other classical behaviour properties such as determinism and livelock-freedom. There is a theoretical infrastructure for compositional livelock analysis (Conserva Filho et al., 2018) as well as for analysis of determinism (OTONI; CAVALCANTI; SAMPAIO, 2017). We plan to integrate these theoretical results into the proposed component model and related verification strategy.

REFERENCES

BASU, A.; BOZGA, M.; SIFAKIS, J. Modeling heterogeneous real-time components in bip. In: *Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods*. Washington, DC, USA: IEEE Computer Society, 2006. (SEFM '06), p. 3–12. ISBN 0-7695-2678-0. Disponível em: <http://dx.doi.org/10.1109/SEFM.2006.27>.

BESNARD, V.; TEODOROV, C.; JOUAULT, F.; BRUN, M.; DHAUSSY, P. Verifying and monitoring uml models with observer automata: A transformation-free approach. In: *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS)*. [S.l.: s.n.], 2019. p. 161–171.

BONAKDARPOUR, B.; BOZGA, M.; JABER, M.; QUILBEUF, J.; SIFAKIS, J. A framework for automated distributed implementation of component-based models. *Distributed Computing*, v. 25, n. 5, p. 383–409, Oct 2012. ISSN 1432-0452.

BOOCH, G.; RUMBAUGH, J.; JACOBSON, I. *The unified modeling language user guide*. Upper Saddle River, NJ: Addison-Wesley, 2005. ISBN 0321267974 9780321267979. Disponível em: <http://www.amazon.com/Unified-Modeling-Language-Guide-Edition/dp/0321267974/ref=sr_1_1?s=books&ie=UTF8&qid=1339289033&sr=1-1>.

BOULGAKOV, A.; GIBSON-ROBINSON, T.; ROSCOE, A. W. Computing maximal bisimulations. In: MERZ, S.; PANG, J. (Ed.). *Formal Methods and Software Engineering*. Cham: Springer International Publishing, 2014. p. 11–26. ISBN 978-3-319-11737-9.

BREU, R.; HINKEL, U.; HOFMANN, C.; KLEIN, C.; PAECH, B.; RUMPE, B.; THURNER, V. Towards a formalization of the unified modeling language. *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, v. 1241, p. 344–366, 1997. ISSN 1611-3349. Disponível em: <http://dx.doi.org/10.1007/BFb0053386>.

CAVALCANTI, A.; MIYAZAWA, A.; SAMPAIO, A.; LI, W.; RIBEIRO, P.; TIMMIS, J. Modelling and verification for swarm robotics. In: FURIA, C. A.; WINTER, K. (Ed.). *Integrated Formal Methods*. Cham: Springer International Publishing, 2018. p. 1–19. ISBN 978-3-319-98938-9.

CAVALCANTI, A.; SAMPAIO, A.; MIYAZAWA, A.; RIBEIRO, P.; Conserva Filho, M.; DIDIER, A.; LI, W.; TIMMIS, J. Verified simulation for robotics. *Science of Computer Programming*, v. 174, p. 1–37, 2019. ISSN 0167-6423. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0167642318301655>.

CHANGE, V. *Astah professional*. 2020. Url http://astah.net/editions/professional. [Online].

CHEHIDA, S.; BAOUYA, A.; BENSALEM, S. Component-based approach combining uml and bip for rigorous system design. In: SALAÜN, G.; WIJS, A. (Ed.). *Formal Aspects of Component Software*. Cham: Springer International Publishing, 2021. p. 27–43. ISBN 978-3-030-90636-8.

CHEN, Z.; LIU, Z.; RAVN, A. P.; STOLZ, V.; ZHAN, N. Refinement and verification in component-based model-driven design. *Sci. Comput. Program.*, Elsevier North-Holland, Inc., Amsterdam, The Netherlands, The Netherlands, v. 74, n. 4, p. 168–196, fev. 2009. ISSN 0167-6423.

CHEN, Z.; MORISSET, C.; STOLZ, V. Specification and validation of behavioural protocols in the rcos modeler. In: *Proceedings of the Third IPM International Conference on Fundamentals of Software Engineering*. Berlin, Heidelberg: Springer-Verlag, 2010. (FSEN'09), p. 387–401. ISBN 3-642-11622-1, 978-3-642-11622-3.

CLARKE, E. M.; KLIEBER, W.; NOVÁČEK, M.; ZULIANI, P. Model checking and the state explosion problem. In: _____. *Tools for Practical Software Verification: LASER, International Summer School 2011, Elba Island, Italy, Revised Tutorial Lectures*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. p. 1–30. ISBN 978-3-642-35746-6. Disponível em: <https://doi.org/10.1007/978-3-642-35746-6_1>.

CLARKE, E. M.; WING, J. M. Formal methods: State of the art and future directions. *ACM Comput. Surv.*, Association for Computing Machinery, New York, NY, USA, v. 28, n. 4, p. 626–643, dec 1996. ISSN 0360-0300. Disponível em: <https://doi.org/10.1145/242223.242257>.

COLDPLAY. *The Scientist*. United Kingdom: [s.n.], 2002.

Conserva Filho, M.; OLIVEIRA, M.; SAMPAIO, A.; CAVALCANTI, A. Compositional and local livelock analysis for csp. *Information Processing Letters*, v. 133, p. 21–25, 2018. ISSN 0020-0190. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0020019018300036>.

DIAS, F.; OLIVEIRA, M.; BATISTA, T.; CAVALCANTE, E.; LEITE, J.; OQUENDO, F.; ARAÚJO, C. Empowering sysml-based software architecture description with formal verification: From sysadl to csp. In: JANSEN, A.; MALAVOLTA, I.; MUCCINI, H.; OZKAYA, I.; ZIMMERMANN, O. (Ed.). *Software Architecture*. Cham: Springer International Publishing, 2020. p. 101–117. ISBN 978-3-030-58923-3.

FITZGERALD, J.; LARSEN, P. G. *Modelling Systems: Practical Tools and Techniques in Software Development*. New York, NY, USA: Cambridge University Press, 2009. ISBN 0521899117.

FREIRE, P. *Pedagogia do oprimido*. Paz e Terra, 1970. ISBN 9788577532285. Disponível em: <https://books.google.com.br/books?id=SL3NAgAAQBAJ>.

GIBSON, C.; KARBAN, R.; ANDOLFATO, L.; DAY, J. Abstractions for executable and checkable fault management models. *Procedia Computer Science*, v. 28, p. 146–154, 2014. ISSN 1877-0509. 2014 Conference on Systems Engineering Research. Disponível em: <https://www.sciencedirect.com/science/article/pii/S1877050914000829>.

GIBSON-ROBINSON, T.; ARMSTRONG, P.; BOULGAKOV, A.; ROSCOE, A. Fdr3: A modern refinement checker for csp. In: ABRAHAM, E.; HAVELUND, K. (Ed.). *Tools and Algorithms for the Construction and Analysis of Systems*. [S.l.]: Springer Berlin Heidelberg, 2014, (Lecture Notes in Computer Science, v. 8413). p. 187–201. ISBN 978-3-642-54861-1.

GORRIERI, R.; VERSARI, C. *Introduction to Concurrency Theory: Transition Systems and CCS*. 1st. ed. [S.l.]: Springer Publishing Company, Incorporated, 2015. ISBN 331921490X.

GRAICS, B.; MOLNáR, V.; VöRöS, A.; MAJZIK, I.; VARRó, D. Mixed-semantics composition of statecharts for the component-based design of reactive systems. *Softw. Syst. Model.*, Springer-Verlag, Berlin, Heidelberg, v. 19, n. 6, p. 1483–1517, nov 2020. ISSN 1619-1366. Disponível em: <https://doi.org/10.1007/s10270-020-00806-5>.

GRÜNINGER, M.; MENZEL, C. The process specification language (psl) theory and applications. *AI Magazine*, v. 24, p. 63–74, 2003. Disponível em: <https://ojs.aaai.org/index.php/aimagazine/article/view/1719>.

HEINEMAN, G. T.; COUNCILL, W. T. (Ed.). *Component-based Software Engineering: Putting the Pieces Together*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001. ISBN 0-201-70485-4.

HOARE, C. A. R. *Communicating Sequential Processes*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1985. ISBN 0-13-153271-5.

HORVÁTH, B.; GRAICS, B.; HAJDU, Á.; MICSKEI, Z.; MOLNÁR, V.; RÁTH, I.; ANDOLFATO, L.; GOMES, I.; KARBAN, R. Model checking as a service: Towards pragmatic hidden formal methods. In: *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*. [S.l.: s.n.], 2020.

LAU, K. kiu; WANG, Z. *A Survey of Software Component Models*. [S.l.], 2005.

LIMA, L.; MIYAZAWA, A.; CAVALCANTI, A.; CORNÉLIO, M.; IYODA, J.; SAMPAIO, A.; HAINS, R.; LARKHAM, A.; LEWIS, V. An integrated semantics for reasoning about sysml design models using refinement. *Softw. Syst. Model.*, Springer-Verlag New York, Inc., Secaucus, NJ, USA, v. 16, n. 3, jul. 2017. ISSN 1619-1366.

MARTINEZ, J.; RUIZ, A.; GARZO, A.; KELLER, T.; RADERMACHER, A.; TONETTA, S. Modelling the component-based architecture and safety contracts of armassist in papyrus for robotics. In: *2021 IEEE/ACM 3rd International Workshop on Robotics Software Engineering (RoSE)*. [S.l.: s.n.], 2021. p. 13–18.

MEYERS, B.; DESHAYES, R.; LUCIO, L.; SYRIANI, E.; VANGHELUWE, H.; WIMMER, M. Promobox: A framework for generating domain-specific property languages. In: COMBEMALE, B.; PEARCE, D. J.; BARAIS, O.; VINJU, J. J. (Ed.). *Software Language Engineering*. Cham: Springer International Publishing, 2014. p. 1–20. ISBN 978-3-319-11245-9.

MIYAZAWA, A.; RIBEIRO, P.; LI, W.; CAVALCANTI, A.; TIMMIS, J. Automatic property checking of robotic applications. In: *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. [S.l.: s.n.], 2017. p. 3869–3876.

Molnár, V.; Graics, B.; Vörös, A.; Majzik, I.; Varró, D. The gamma statechart composition framework: Design, verification and code generation for component-based reactive systems. In: *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*. [S.l.: s.n.], 2018. p. 113–116.

Object Management Group (OMG). *Object Constraint Language - Specification v2.4*. 2014. OMG Document Number: formal/2014-02-03 (<https://www.omg.org/spec/OCL/2.4/>).

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

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

Object Management Group (OMG). *Semantics of a Foundational Subset for Executable UML Models, Version 1.3*. 2017. OMG Document Number formal/formal/17-07-02 (<https://www.omg.org/spec/FUML/1.3/>).

Object Management Group (OMG). *Semantics of a Foundational Subset for Executable UML Models, Version 1.3*. 2017. OMG Document Number formal/formal/17-07-02 (<https://www.omg.org/spec/FUML/1.3/>).

Object Management Group (OMG). *Precise Semantics of UML Composite Structures - Specification v1.2*. 2019. OMG Document Number: formal/19-05-01 (<https://www.omg.org/spec/PSCS/1.2/>).

Object Management Group (OMG). *Precise Semantics of UML State Machines - Specification v1.0*. 2019. OMG Document Number: formal/19-05-01 (<https://www.omg.org/spec/PSSM/1.0/>).

OLIVEIRA, M. V. M.; ANTONINO, P.; RAMOS, R.; SAMPAIO, A.; MOTA, A.; ROSCOE, A. W. Rigorous development of component-based systems using component metadata and patterns. *Formal Aspects of Computing*, v. 28, n. 6, p. 937–1004, Nov 2016. ISSN 1433-299X.

OTONI, R.; CAVALCANTI, A.; SAMPAIO, A. Local analysis of determinism for CSP. In: CAVALHEIRO, S. A. da C.; FIADEIRO, J. L. (Ed.). *Formal Methods: Foundations and Applications - 20th Brazilian Symposium, SBMF 2017, Recife, Brazil, November 29 - December 1, 2017, Proceedings*. Springer, 2017. (Lecture Notes in Computer Science, v. 10623), p. 107–124. Disponível em: <https://doi.org/10.1007/978-3-319-70848-5_8>.

PEREIRA, D. I. A.; OLIVEIRA, M. V. M.; SILVA, S. R. R. *Tool Support for Formal Component-based Development*. 2016.

RAMOS, R.; SAMPAIO, A.; MOTA, A. A semantics for uml-rt active classes via mapping into circus. In: STEFFEN, M.; ZAVATTARO, G. (Ed.). *Formal Methods for Open Object-Based Distributed Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005. p. 99–114. ISBN 978-3-540-31556-8.

RAMOS, R.; SAMPAIO, A.; MOTA, A. Systematic development of trustworthy component systems. In: CAVALCANTI ANAAND DAMS, D. R. (Ed.). *FM 2009: Formal Methods*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009. p. 140–156. ISBN 978-3-642-05089-3.

RAMOS, R. T. *Systematic Development of Trustworthy Component-based Systems*. Tese (Doutorado), Brazil, 2011.

ROSCOE, A. *The pursuit of buffer tolerance*. [S.l.], 2005. Disponível em: <http://www.cs.ox.ac.uk/people/bill.roscoe/publications/106.pdf>.

ROSCOE, A. *Understanding Concurrent Systems*. 1st. ed. Berlin, Heidelberg: Springer-Verlag, 2010. ISBN 184882257X.

ROSCOE, A. W. *The Theory and Practice of Concurrency*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1997. ISBN 0136744095.

SAMPAIO, A.; NOGUEIRA, S.; MOTA, A.; ISOBE, Y. Sound and mechanised compositional verification of input-output conformance. *Software Testing, Verification and Reliability*, v. 24, n. 4, p. 289–319, 2014. Disponível em: <https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.1498>.

SCHNEIDER, S. *Concurrent and Real Time Systems: The CSP Approach*. 1st. ed. New York, NY, USA: John Wiley and Sons, Inc., 1999. ISBN 0471623733.

SELIC, B.; GULLEKSON, G.; WARD, P. T. *Real-Time Object-Oriented Modeling*. USA: John Wiley and Sons, Inc., 1994. ISBN 0471599174.

WOODCOCK, J.; CAVALCANTI, A.; FITZGERALD, J.; LARSEN, P.; MIYAZAWA, A.; PERRY, S. Features of cml: A formal modelling language for systems of systems. In: *2012 7th International Conference on System of Systems Engineering (SoSE)*. [S.l.: s.n.], 2012. p. 1–6.

APPENDIX A – WELL-FORMEDNESS CONDITIONS

This appendix summarises the constraints that represent well-formedness conditions in our component metamodel:

The first OCL constraint is related to the System; there is only one System in the component specification:

```
context System
    inv UniqueSystem:
        self->size()=1
```

A component can be a *BasicComponent* or a *HierarchicalComponent*. The *AbstractComponent* represents this generalisation. There are, in this context, some constraints on both types of components.

A *BasicComponent* must have a state machine diagram with at least one state.

```
context BasicComponent

inv STMBasicComponent:
   self.componentstatemachine->size()=1
     and self.componentstatemachine.region.state->size()>1
```

Component instances and parts are present only in hierarchical components.

```
context AbstractComponent

inv STRBasicComponent:
 self.oclIsTypeOf(BasicComponent)
  implies
   self.componentclass.part-> size()=0

inv STRHierarchicalComponent:
 self.oclIsTypeOf(HierarchicalComponent)
  implies  self.componentclass.part->size()>0
```

```
inv multiplicityLimited:
  self.oclIsTypeOf(HierarchicalComponent) implies
   self.componentclass.part->size() <> UnlimitedNatural
```

The State Machine of a component has the same name as the owner component.

```
context ComponentStateMachine

inv NameStateMachine:
 self.name = self.abstractcomponent.name
```

A *BasicComponent* has the same name as its *BasicComponentClass*.

```
context BasicComponent

inv BasicName:
 self.name =
       self.basiccomponentclass.name
```

A *HierarchicalComponent* has the same name of its *HierarchicalComponentClass*. And a *HierarchicalComponent* has at least one other component.

```
context HierarchicalComponent

inv HierarchicalName:
 self.name =
  self.hierarchicalcomponentclass.name

inv MultiHierarchical:
 self.abstractcomponent->
  forAll(AbstractComponent->size()>=1)
```

Each *HierarchicalComponentClass* must have at least one port. Port names must be unique. Ports must realise or provide an interface.

```
context BasicComponentClass

inv qtdPortBC:
```

```
 self.ownedPort->size()>=1


inv namedPort:
 self.ownedPort->
  forAll(c1,c2|c1<>c2 implies
    c1.name<>c2.name and
      c1.name <>'' and c2.name <>'')



inv realizedPort:
 self.ownedPort->
  forAll(c1| c1.required()->size()>0
   or
    c1.provided()->size()>0 )
```

APPENDIX B – RULES

In this appendix, we present the translations rules and auxiliary functions.

Auxiliary function that returns input operations of a component:

Rule 20. Function Subtype Operations Input

subtype_operationsInput(c : AbstractComponent) :: CSPSpecification =

for each class in c.Interface
  for each operation in class.operation sep |
    *operation.name*_I
  end for
end for

Auxiliary function that returns output operations of a component:

Rule 21. Function Subtype Operations Output

subtype_operationsOutput(c : AbstractComponent) :: CSPSpecification =

for each class in c.Interface
  for each operation in class.operation sep |
    operation.name _0
  end for
end for

Auxiliary function that define *get* channels.

Rule 22. Function Channel get_var

channel_get_var(c : AbstractComponent) :: CSPSpecification =

for each class in c.classes
  for each v in class.attributes
    channel get_vid(v) : ID_*c.name*.*type*_vid(v)
  end for
end for

Auxiliary function that define *set* channels:

Rule 23. Function Channel set_var

channel_set_var(c : AbstrcatComponent) :: CSPSpecification =

for each class in c.classes
  for each v in class.attributes
    channel set_vid(v) : ID_*c.name*.*type*_vid(v)
  end for
end for

It provides a set of synchronisation component:

Rule 24. Function Set Sync

setSync(c : AbstractComponent) :: CSPSpecification =

getvarname(c) ∪ setvarname(c) ∪ guard(c)
  ∪ ⟨*internal*⟩

List of get channels:

Rule 25. Function Get Variable Names

getvarname(c : AbstractComponent) :: CSPSpecification =

for each class in c.classes
  for each v in class.attributes sep ,
    get_vid(v).id
  end for
end for

List of set channels:

Rule 26. Function Set Variable Names

setvarname(c : AbstractComponent) :: CSPSpecification =

for each class in c.classes
  for each v in class.attributes sep ,
    set_vid(v).id
  end for
end for

Auxiliary function that gives the set of channels that must be hidden in a component definition:

Rule 27. Function set Hidden

setHidden(c : AbstractComponent) :: CSPSpecification =

getvarname(c) ∪ setvarname(c) ∪ {internal}

List of attributes´s component:

Rule 28. Function Variable List

varList(c : AbstractComponent) :: CSPSpecification =

for each cl inc.Classes
  for each v in cl.vars sep ,
    v.name
  end for
end for

A process of hierarchical port.

Rule 29. Function Process Port

portProcess(c : AbstractComponent) :: CSPSpecification =

for each class inc.StructureClass
  for each port in class.port
    if(*port.OwnedbyHierarchicalComponent*)
      port_process_*c.name*(id) =
        (□ *op* : *port.Operations* • *port.name*_internal.*op* → *port.name*_external.*op* →
            port_process_*c.name*(id))
        □
        (□ *op* : *port.Operations* • *port.name*_external.*op* → *port.name*_internal.*op* →
            port_process_*c.name*(id))

    end if
  end for
end for

APPENDIX C – CSP -CASE STUDIES

This appendix contains the UML component model translation to CSP written using the machine-readable CSP_M, an ASCII syntax for CSP combined with a functional programming language. Details about the constructions of CSP_M are presented in (ROSCOE, 1997).

C.1 DINING PHILOSOPHER

```
channel fork_left: id_Fork.operation
channel fork_right: id_Fork.operation
channel phil_right: id_Phil.operation
channel phil_left: id_Phil.operation

datatype operation = picksup_I | picksup_O | putsdown_I | putsdown_O
subtype Phil_I = picksup_I | putsdown_I
subtype Fork_I = picksup_I | putsdown_I

subtype Phil_O = picksup_O | putsdown_O
subtype Fork_O = picksup_O | putsdown_O


-- Fork
STM_Fork(id) = Availablle(id)
Availablle(id) = (fork_right.id.picksup_I->  fork_right.id.picksup_O
                ->Busy1(id))
            []
                (fork_left.id.picksup_I -> fork_left.id.picksup_O
                ->Busy2(id))
Busy1(id) = (fork_right.id.putsdown_I -> fork_right.id.putsdown_O
              -> Availablle(id))

Busy2(id) = (fork_left.id.putsdown_I -> fork_left.id.putsdown_O
             ->  Availablle(id))

Fork(id) =STM_Fork(id)

--Philosopher
```

```
STM_Phil(id) = HoldForkR(id)
HoldForkR(id) = (phil_right.id.picksup_I -> phil_right.id.picksup_O
                -> HoldForkL(id))
HoldForkL(id) = (phil_left.id.picksup_I -> phil_left.id.picksup_O
               -> PutsDownR(id))
PutsDownR(id) = (phil_right.id.putsdown_I -> phil_right.id.putsdown_O
                                  ->PutsDownL(id))
PutsDownL(id) = (phil_left.id.putsdown_I -> phil_left.id.putsdown_O
              ->HoldForkR(id))

Phil(id) =STM_Phil(id)
```

C.2 RING BUFFER

```

channel port_env:id_CONTROL.t_id.operation
channel port_control:id_CONTROL.index.operation
channel port_cell:id_CELL.t_id.operation

index = {1..3}
maxring=3
maxbuffer=4

type_size={0..4}
type_cache={0..2}
type_top={1..3}
type_bot={1..3}
type_vl_env={0..2}
type_val={0..2}
type_data={0..1}

t_id = {0..9}
channel internal: t_id

channel get_size : id_CONTROL.type_size
channel set_size : id_CONTROL.type_size
channel get_cache : id_CONTROL.type_cache
channel set_cache : id_CONTROL.type_cache
```

```
channel get_top : id_CONTROL.type_top
channel set_top : id_CONTROL.type_top
channel get_bot : id_CONTROL.type_bot
channel set_bot : id_CONTROL.type_bot
channel get_vl_env : id_CONTROL.type_vl_env
channel set_vl_env : id_CONTROL.type_vl_env
channel get_val : id_CELL.type_val
channel set_val : id_CELL.type_val
channel get_data : id_CELL.type_data
channel set_data : id_CELL.type_data


datatype operation = retrieve_data_I| retrieve_data_O.type_vl_env|
   write_I.type_val| write_O | read_I |
   read_O.type_val|  receive_data.type_vl_env
subtype CELL_I = write_I.type_val | read_I
subtype CONTROL_I = retrieve_data_I | receive_data.type_vl_env |
   write_I.type_val| read_I
subtype CELL_O = write_O | read_O.type_val
subtype CONTROL_O= retrieve_data_O.type_vl_env|write_O|
    read_O.type_val

CONTROL_memory(id,size,cache,top,bot,vl_env)=
 get_size.id!size->CONTROL_memory(id,size,cache,top,bot,vl_env)
 []
 get_cache.id!cache->CONTROL_memory(id,size,cache,top,bot,vl_env)
 []
 get_top.id!top->CONTROL_memory(id,size,cache,top,bot,vl_env)
 []
 get_bot.id!bot->CONTROL_memory(id,size,cache,top,bot,vl_env)
 []
 get_vl_env.id!vl_env->CONTROL_memory(id,size,cache,top,bot,vl_env)
 []
 set_size.id?vl -> CONTROL_memory(id,vl,cache,top,bot,vl_env)
 []
 set_cache.id?vl -> CONTROL_memory(id,size,vl,top,bot,vl_env)
 []
 set_top.id?vl ->CONTROL_memory (id,size,cache,vl,bot,vl_env)
 []
```

```
64  set_bot.id?vl -> CONTROL_memory(id,size,cache,top,vl,vl_env)
    []
66  set_vl_env.id?vl -> CONTROL_memory(id,size,cache,top,bot,vl)
    []
68  size>0 & port_env.id.1.retrieve_data_I ->CONTROL_memory(id,size,cache,
        top,bot,vl_env)
    []
70  size<maxbuffer & port_env.id.2.receive_data?vl->CONTROL_memory(id,size,
        cache,top,bot,vl_env)
    []
72  size==1 & internal.3 ->CONTROL_memory(id,size,cache,top,bot,vl_env)
    []
74  size>0 & internal.4 ->CONTROL_memory(id,size,cache,top,bot,vl_env)
    []
76  size>1 & internal.5 ->CONTROL_memory(id,size,cache,top,bot,vl_env)
    []
78  size==0 & internal.6 ->CONTROL_memory(id,size,cache,top,bot,vl_env)


80


82 CELL_memory(id,val) =
    get_val.id!val-> CELL_memory ( id, val )
84  []
    set_val.id?vl -> CELL_memory ( id, vl)
86


88 STM_CONTROL(id) = Init_Control(id)
   Init_Control(id) =
90  (port_env.id.1.retrieve_data_I-> get_cache.id?cache->
     port_env.id.1.retrieve_data_O?cache -> Read(id))
92  []
    (port_env.id.2.receive_data?vl -> set_vl_env.id!vl ->Write(id) )
94 Read(id) = (internal.3->  set_size.id!0->Init_Control(id))
    []
96  (internal.5-> get_bot.id?bot->  port_control.id.bot.read_I->
     port_control.id.bot.read_O?vl -> set_cache.id!vl ->get_bot.id?bot->
98   set_bot.id!(bot%maxring) +1 -> get_size.id?size->
     set_size.id!(size -1)%maxbuffer -> Init_Control(id) )
100 Write(id) = (internal.4-> get_vl_env.id?vl_env->get_top.id?top->
```

```
  port_control.id.top.write_I!vl_env->  port_control.id.top.write_O ->
  get_size.id?size-> set_size.id!(size %maxbuffer)+1 ->
  get_top.id?top-> set_top.id!(top%maxring)+1 ->  Init_Control(id))
 []
 (internal.6-> get_vl_env.id?vl_env->  set_cache.id!vl_env
  ->set_size.id!1-> Init_Control(id) )


STM_CELL(id) = Init_Cell(id)
Init_Cell(id) = (port_cell.id.write?vl-> port_cell.id.write_ack
  ->set_val.id!vl -> Init_Cell(id))
[]
(port_cell.id.read->get_val.id?val->  port_cell.id.read_ack!val
  ->Init_Cell(id) )


CELL(id) = STM_CELL(id) [|{|get_val.id,set_val.id,internal|}|]
   CELL_memory(id,0) \{|get_val,set_val,internal |}

CONTROL(id) =STM_CONTROL(id)
 [|{|get_size.id,set_size.id,  get_cache.id,set_cache.id,get_top.id,
  set_top.id,get_bot.id,set_bot.id,  get_vl_env.id,set_vl_env.i,
  port_env.id.2.receive_data,  port_env.id.1.retrieve_data_I,internal|}|]
CONTROL_memory(id,0,0,1,1,0)
 \{|get_size,set_size,get_cache,set_cache,   get_top,set_top,get_bot,
    set_bot,   get_vl_env,set_vl_env,internal |}
```

C.3 LEADERSHIP ELECTION

```

channel bus_receiver : id_BUS.operation
channel bus_sender : id_BUS.operation
channel node_receiver : id_NODE.operation
channel node_sender : id_NODE.operation


type_status={0,1}
type_claim ={'u', 'f', 'l'}
```

```
type_petition ={0..3}
claim = {'u', 'f', 'l'}
petition ={0..3}
maxPetition =4

datatype type_data =  pack.claim.petition


t_id = {1..5}
channel internal: t_id

channel get_BUSCELL_data : id_BUS.type_data
channel set_BUSCELL_data : id_BUS.type_data
channel set_Node_data : id_NODE.type_data
channel get_Node_data : id_NODE.type_data
channel get_Node_myClaim : id_NODE.type_claim
channel set_Node_myClaim : id_NODE.type_claim
channel get_Node_myPetition: id_NODE.type_petition
channel set_Node_myPetition: id_NODE.type_petition
channel set_Node_receivedPack : id_NODE.type_data
channel get_Node_receivedPack : id_NODE.type_data


datatype operation = send_pack_I | send_pack_O.type_data | receive_pack_I
    |      receive_pack_O.type_data| send_status.type_status |


subtype BUS_I = receive_pack_I |  send_pack_I
subtype BUS_O = receive_pack_O.type_data  | send_pack_O.type_data|
              send_status.type_status

subtype NODE_O = receive_pack_I |   send_pack_I  | send_status.
    type_status

subtype NODE_I = receive_pack_O.type_data  | send_pack_O.type_data


pack_claim(pack.claim.petition) = claim  pack_petition(pack.claim.petition
```

```
    ) = petition
47
49 NODE_memory(id,myClaim,myPetition,data ,receivedPack) =
   get_Node_myClaim.id!myClaim -> NODE_memory(id, myClaim, myPetition,data ,
       receivedPack)
51 []
   set_Node_myClaim.id?vl -> NODE_memory(id,vl,myPetition,data,receivedPack)
53 []
   get_Node_myPetition.id!myPetition   -> NODE_memory(id,myClaim,myPetition,
       data , receivedPack)
55 []
   set_Node_myPetition.id?vl -> NODE_memory(id, myClaim, vl, data ,
       receivedPack)
57 []
   get_Node_data.id!data -> NODE_memory(id, myClaim, myPetition,data,
       receivedPack)
59 []
   set_Node_data.id?vl -> NODE_state(id,myClaim, myPetition,vl,receivedPack)
61 []
   get_Node_receivedPack.id!data -> NODE_memory(id, myClaim, myPetition,
       data , receivedPack)
63 []
   set_Node_receivedPack.id?vl -> NODE_memory(id, myClaim, myPetition, data
       , vl)
65 []
   ((pack_claim(receivedPack) == 'u' and pack_petition(receivedPack) ==0
       and myClaim != 'l')
67 or ((pack_claim(receivedPack) == 'u' and pack_petition(receivedPack)
      >= myPetition) and myClaim == 'u')
   or (pack_claim(receivedPack) == 'l' and myClaim == 'l'))
69 & internal.2 -> NODE_memory(id, myClaim, myPetition, data , receivedPack
       )
   []
71 (pack_claim(receivedPack) == 'l' and myClaim != 'l') & internal.3 ->
       NODE_memory(id, myClaim, myPetition, data , receivedPack)
   []
73
   ((pack_claim(receivedPack) != 'l' and myClaim == 'l')
```

```
75 or
   (pack_claim(receivedPack) == 'u'  and (myClaim == 'u' and myPetition >
       pack_petition(receivedPack) )))
77 & internal.4 -> NODE_memory(id, myClaim, myPetition,  data , receivedPack
       )

79 STM_NODE(id) = OFF(id)
   OFF(id) = node_sender.id.send_status!1 -> set_Node_myClaim.id!'u' ->
81           get_Node_myPetition.id?myPetition -> set_Node_myPetition.id!((
                myPetition-1)% maxPetition) ->
             get_Node_myPetition.id?myPetition ->
83           set_Node_data.id!pack.'u'.myPetition  ->
             get_Node_data.id?data ->
85           node_sender.id.send_pack_I -> node_sender.id.send_pack_O!data
                ->
             ON(id)
87
   ON(id) =  node_receiver.id.receive_pack_I ->  node_receiver.id.
       receive_pack_O?data_ ->
89           set_Node_receivedPack.id!data_ ->
             CHOICE(id)
91           []
            node_sender.id.send_status!0 -> OFF(id)
93
   CHOICE(id) = internal.2 ->   set_Node_myClaim.id!'u' -> UNDECIDED(id)
95
                []
97
                internal.4 ->   set_Node_myClaim.id!'l' ->
                    get_Node_myPetition.id?mypetion -> set_Node_myPetition.id
                    !(mypetion +1)%maxPetition  ->  LEADER(id)
99

101             []

103             internal.3  ->   set_Node_myClaim.id!'f' ->       FOLLOWER(id)

105
   UNDECIDED(id) =   get_Node_myClaim.id?myclaim -> get_Node_myPetition.id?
```

```
    mypetion ->
107         node_sender.id.send_pack_I -> node_sender.id.send_pack_O!pack.
                myclaim.mypetion -> ON(id)


109


111
LEADER(id)=    get_Node_myClaim.id?myclaim -> get_Node_myPetition.id?
    mypetion ->
113         node_sender.id.send_pack_I -> node_sender.id.send_pack_O!pack.
                myclaim.mypetion  -> ON(id) --valLeaders > 0


115


117
FOLLOWER(id) =    get_Node_myClaim.id?myclaim -> get_Node_myPetition.id?
    mypetion ->
119         node_sender.id.send_pack_I -> node_sender.id.send_pack_O!pack.
                myclaim.mypetion  -> ON(id)


121


123 NODE(id) = STM_NODE(id)
    [|{|
125 set_Node_data.id, get_Node_data.id,
    get_Node_myClaim.id,set_Node_myClaim.id,
127 set_Node_myPetition.id,get_Node_myPetition.id,
    set_Node_receivedPack.id,get_Node_receivedPack.id,
129 internal
    |}|]
131 NODE_memory(id,'u', 1,  pack.'u'.0 ,pack.'u'.0 )
    \{| set_Node_data, get_Node_data,get_Node_myClaim,set_Node_myClaim,
133 set_Node_myPetition,get_Node_myPetition,set_Node_receivedPack,
    get_Node_receivedPack,internal |}


135


137
```

```
139
    BUS_memory( id , data ) =
141 get_BUSCELL_data.id!data -> BUS_memory( id , data )
    []
143 set_BUSCELL_data.id?vl -> BUS_memory( id , vl )


145


147 STM_BUS(id) = CellIddle(id)
    CellIddle(id) =
149  (bus_sender.id.send_status.1 ->CellOn(id))
    []
151 (bus_receiver.id.receive_pack_I->
        bus_receiver.id.receive_pack_O!pack.'u'.0  -> CellIddle(id) )
153
    CellOn(id) =
155  (bus_sender.id.send_status.0 ->CellIddle(id) )
      []
157 (bus_receiver.id.receive_pack_I->get_BUSCELL_data.id?data->
              bus_receiver.id.receive_pack_O!data ->CellOn(id))
159  []
     (bus_sender.id.send_pack_I->
161    bus_sender.id.send_pack_O?pack.claim.petition->
        set_BUSCELL_data.id!pack.claim.petition ->CellOn(id) )
163  []
     get_BUSCELL_data.id?data -> bus_receiver.id.receive_pack_I ->
         bus_receiver.id.receive_pack_O!data ->  CellOn(id)
165


167 BUS(id) =STM_BUS(id)
     [|{|get_BUSCELL_data.id,set_BUSCELL_data.id,internal|}|]
169 BUS_memory(id,pack.'u'.0)
    \{|get_BUSCELL_data,set_BUSCELL_data,internal |}
```