UML Component-Based Design Using a Safe Stepwise Strategy

Flávia Falcão[1*], Lucas Lima[2*] and Augusto Sampaio[1*]

[1]Centro de Informática, Universidade Federal de Pernambuco, Brazil.
[2]Departamento de Computação, Universidade Federal Rural de Pernambuco, Brazil.

*Corresponding author(s). E-mail(s): fmcf2@cin.ufpe.br; lucas.albertins@ufrpe.br; acas@cin.ufpe.br;

Abstract

Model-based engineering emerged as an approach to tackle the complexity of current system 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. 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 (our previous work on) BRIC as an underlying (and totally hidden) component development framework so that our approach benefits from all the formal infrastructure developed for BRIC using CSP. Component composition, specified via UML structural diagrams, ensures 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

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 [1]. 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 [2].

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 [3]. 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 [4] the authors present component-based refinement that focuses on the separation of interface and functional contracts, supporting different levels of abstraction. The approach in [5, 6] 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 [7], the authors proposed a cloud-based, end-to-end verification workflow for SysML [8] 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 verificaion of classical properties such as deadlock freedom, and traceability from the counterexample verification to the component model. For instance, the work in [5–7] features traceability, but does embody a component model, nor provides compositional reasoning. The approach in [4] contemplates a component model, but does not support compositional verification nor traceability.

In previous work, we have proposed a formal component model, together with a rule-based composition strategy, called BRIC [2, 9, 10]. BRIC has the process algebra CSP [11] 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 is well-suited for modelling software systems in general; however, it lacks support 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.

In [12], we have introduced a formal CBSD model for UML [13], motivated by the fact that UML is a widely used notation in industry and amenable to mechanised analysis. The current paper significantly improves and extends the benefits from the overall formal infrastructure built around BRIC as a semantic model for a UML component model. Our major contributions are detailed as follows.

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. In [12], we have proposed an initial set of well-formedness conditions that were described in an informal way. In the current work these are extended to consider more relevant features. In particular, we define how components can be composed to give rise to more elaborate components. As a new contribution, the well-formedness conditions are now

presented as rules expressed using OCL (Object Constraint Language).

As another important 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 component 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 for the early verification of the assumed properties of BRIC contracts, which must be an I/O Process (Definition 4) 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 [14]. 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-fomedness condition is violated, a warning message is shown indicating where the problems is (Step 3). Step 4 corresponds to the automatic translation of the UML model into BRIC. This allows the analysis using the FDR model checker, which is shown in Step 5. If an analysis fails, the resulting counterexample is traced back to the component model in the

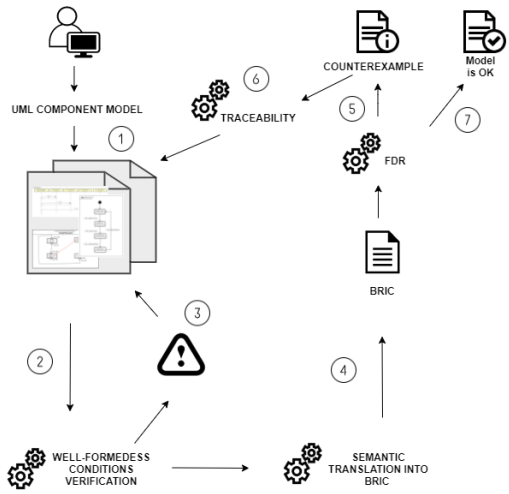


Fig. 1 The major steps of the proposed approach

form of UML diagrams in Step 6. Otherwise, the model was correctly specified, step 7.

In summary, the contributions of the current paper with respect to [12] are the following:

- A complete account of the component metamodel;
- 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;
- Tracealibity via the presentation of verification results in FDR as UML diagrams;
- Significant improvement of an initial proof of concept tool to include the fully automatic semantic generation and support for traceability of well-formedness condition violations as well as deadlock scenarios;
- Additional, and much more elaborate, case study of a Leadership election protocol; and
- A detailed account of related work.

The next section introduces some background of CSP and BRIC. In Section 3, we present the proposed UML component model, the well-formedness conditions, and the approach to component (instance) composition. The semantic translation into BRIC together with the strategy for protocol automation are described in Section 4. Section 5 is dedicated to tool support and the

development of the case studies. Section 6 discusses related work, and Section 7 summarises our results, and discusses future work.

2 Background

This section introduces CSP, LTS, weak bisimulation and BRIC.

2.1 CSP

The process algebra CSP (Communicating Sequential Processes) [11], supports the modelling and verification of concurrent reactive systems. It provides a set of semantic models that helps one to reason about processes and how they interact.

In CSP, communication between processes takes place over named channels and their synchronous events. CSP has two basic processes: $STOP$, which does nothing and represents the deadlocked process, and the process $SKIP$, which indicates successful termination.

Processes are defined in terms of a set of events 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 P. A structured event is given by a communication channel that may carry values, and its declaration form is: *channel* $c : T$. 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 $c.T = \{c.v \mid v \in T\}$, which is a subset of Σ. An input communication on c has the form $c?v$, and an output communication has the form $c!v$, which is the same as $c.v$.

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 of it [15]. 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 $\Box\ x : A \bullet P(x)$ valuates $P(x)$ 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.

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 [11]. 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.

$$
\begin{aligned}
&Fork(id) = STM\_Fork(id)\\
&STM\_Fork(id) = Available(id)\\
&Available(id) = (fork\_right.id.picksup\_I \rightarrow\\
&\quad fork\_right.id.picksup\_O \rightarrow Busy1(id))\\
&\qquad \Box\\
&\quad (fork\_left.id.picksup\_I \rightarrow\\
&\qquad fork\_left.id.picksup\_O \rightarrow Busy2(id))\\
&Busy1(id) = (fork\_right.id.putsdown\_I \rightarrow\\
&\quad fork\_right.id.putsdown\_O \rightarrow Available(id))\\
&Busy2(id) = (fork\_left.id.putsdown\_I \rightarrow\\
&\quad fork\_left.id.putsdown\_O \rightarrow Available(id))
\end{aligned}
$$

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 are values representing the actions that are offered by forks. Initially, a fork is available for both philosophers; however, two philosophers cannot 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 left holds the fork, and similarly for the one on the right. 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 written in CSP may be understood in terms of operational, denotational, and algebraic semantics. The operational semantics is defined in terms of 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 model in such a way that the behaviour can be represented in terms of *traces, failures and divergences*.

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.

In the *Traces Model* for CSP, a process P is a trace refinement of a process Q (written as $Q \sqsubseteq_{\mathrm{T}} P$) if, and only if, Q contains all traces within P: $traces(P) \subseteq traces(Q)$. These processes are trace equivalent (written as $P \equiv_{\mathrm{T}} Q$) if $Q \sqsubseteq_{\mathrm{T}} P$ and $P \sqsubseteq_{\mathrm{T}} Q$, i.e., $traces(P) = traces(Q)$. This model does not give a complete description of a process; it is confined to express safety properties.

The *Stable Failures Model* gives more information about processes. For instance, it allows distinguishing between internal and external choice. It also allows one to detect deadlocked processes. 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; and X is a refusal set. The *stable* in this model means that the sequences represented by s are those that reach a stable state where no internal transition (τ) is possible.

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 within P, and P presents less stable failures than Q; it refuses 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 failure-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 $failures(P) = failures(Q)$.

An important phenomenon captured by the stable failures model is deadlock. Deadlock happens to networks of communicating processes when two processes cannot agree to communicate with each other; thus, the whole system becomes permanently halted.

The *Failures/Divergence Model* allows one to detect not only deadlocked but also livelocked (divergence) processes. A divergence (or livelock) occurs when a process can perform only internal events indefinitely.

[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.

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 [11]. For instance, consider the processes $P = P$ and $Q = (a \rightarrow Q) \setminus a$. Q converts the 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. 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.

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

2.2 LTS and Weak Bissimulation

In this section, we define the concepts of an LTS and Weak Bisimulation, which are essencial 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 System (LTS).

An LTS, Definition 1, 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 to start from [11].

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 [16].

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 [16].

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

where if $\alpha \neq \tau$, then $s \stackrel{\alpha}{\Longrightarrow} t$ 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 $s \stackrel{\alpha}{\Longrightarrow} t$ means that from s one can reach t by doing zero or more τ actions.

2.3 The BRIC Component Model

BRIC formalises concepts of interfaces, dynamic behaviour, component contracts, and communication protocols with focus on the interaction points of black box components and their runtime behaviour.

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 $R : C \rightarrow I$

between channels and interfaces, 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 of fork and philosopher are defined as follows:

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

$PHIL$:
$B_{PHIL} = Phil(id)$,
$R_{PHIL} = \langle (phil\_right.id, \{picksup\_I, picksup\_O, putsdown\_I, putsdown\_O\}), (phil\_left.id, \{picksup\_I, picksup\_O, putsdown\_I, putsdown\_O\}) \rangle$,
$C_{PHIL} = \{phil\_right.id, phil\_left.id\}$,
$I_{PHIL} = \{picksup\_I, picksup\_O, putsdown\_I, putsdown\_O\}$

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,that is:

$$inputs(c, P) \cup outputs(c, P) \subseteq \{|\ c\ |\} \wedge \quad (1)$$
$$inputs(c, P) \cap outputs(c, P) = \{\}$$

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.
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 a 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 in 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.
- **Input determinism**: If a set of input events in 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 in P are internal. The process, however, must offer at least one output on that channel.

Normally, a component is defined once and reused multiple times and in multiple different contexts. In the example, 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 5.

Communication protocols are commonly associated to 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 (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 6 (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 failure model. Hence, $ProtIMP(P, C)$ may have more failures than $P \upharpoonright C$ (first refinement statement of Definition 6). 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 6.

One of the important contributions of this paper is that, rather than asking the user to propose a valid protocol implementation from a projection over a set of channels, we constructively 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 In this way, a protocol implementation for the $fork1$ process over the channel $fork\_right.1$ is 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 will be hidden, except the ones related to channel $fork\_right.1$.

$$\begin{aligned}
&fork1 = FORK(1) = STM_FORK(1)\\
&STM_FORK(1) = Available(1)\\
&Available(1) = (fork_right.1.picksup_I \rightarrow\\
&\quad fork_right.1.picksup_O \rightarrow Busy1(1))\\
&\quad \Box\\
&\quad Busy2(1)\\
&Busy1(1) = (fork_right.1.putsdown_I \rightarrow\\
&\quad fork_right.1.putsdown_O \rightarrow Available(1))\\
&Busy2(1) = Available(1)
\end{aligned}$$

In this case, the projection is directly obtained by hiding the relevant events, but the resulting process has the following livelock sequence: $STM\_FORK(1)$; $Available(1)$; $Busy2(1)$; $Available(1)$. 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:

$$\begin{aligned}
&PROT_FORK(ch) = ch.picksup_I \rightarrow\\
&\quad ch.picksup_O \rightarrow ch.putsdown_I \rightarrow\\
&\quad ch.putsdown_O \rightarrow PROT_FORK(ch)
\end{aligned}$$

In general, however, it might not be easy to construct a valid protocol manually. As presented in Section 4.3.1, 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 7 (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 8, $CTX(P)$, represents its possible communications, and is formally defined by a deadlock-free deterministic process with the same traces as those of P.

Definition 8 (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 $traces(CTX(P)) = traces(P)$.

2.3.2 BRIC Composition Rules

The constructive design of a BRIC component architectural model is based on composition rules for components. 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.

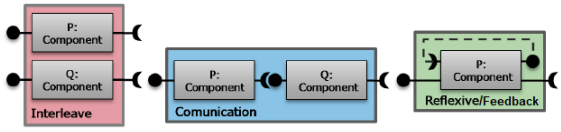


Fig. 2 Composition Rules

Each rule (see Figure 2) has well-defined side-conditions that ensure a sound composition [10]. 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 [11]). Reflexive composition deals with more complex systems that indeed present cycles of dependencies in the system topology.

In order to illustrate these communication rules, we consider two philosophers and two forks *(phil1, phil2, fork1, fork2)*. The first rule, interleave composition, is represented by the interleaving of all components:

$$CON\_0 = fork1 \;|||\; fork2 \;|||\; phil1 \;|||\; phil2$$

We may connect the ports of the component that resulted from the interleave composition using feedback composition. In our example, we connect the port $phil\_right.1$ with the port $fork\_left.1$, where $\hookrightarrow$ represents a feedback composition:

$$CON\_feedback = CON\_0[phil\_right.1 \hookrightarrow fork\_left.1]$$

In our strategy for component composition proposed in this work, we decided not to use the communication composition rule because we put all components in interleaving, first, which results in a new component. The communications after this are made by feedback and reflexive rules.

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. 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 [17]. Component models can be defined for different levels of component abstractions. They can be very general, but need to define [18]:

- *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 3, 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 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 4(a) (class diagram view), 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 as shown in Figure 4(b) (composite structure diagram perspective). Also, each *BasicComponent* has one state machine whose name is formed of the prefix *STM_* and the component's name.

Figure 5 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 or right ports and then waits for the fork to be put down via the same port. Figure 6 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 Composite Structure Diagram. A *HierarchicalComponentClass* is a UML *EncapsulatedClassifier* element, hence, it 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.

The Dining Philosophers is modelled as a *System* element and, therefore, as a *HierarchicalComponent*; see Figure 7. It has a *HierarchicalComponentClass* that is related to one or more *Fork* and one or more *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.

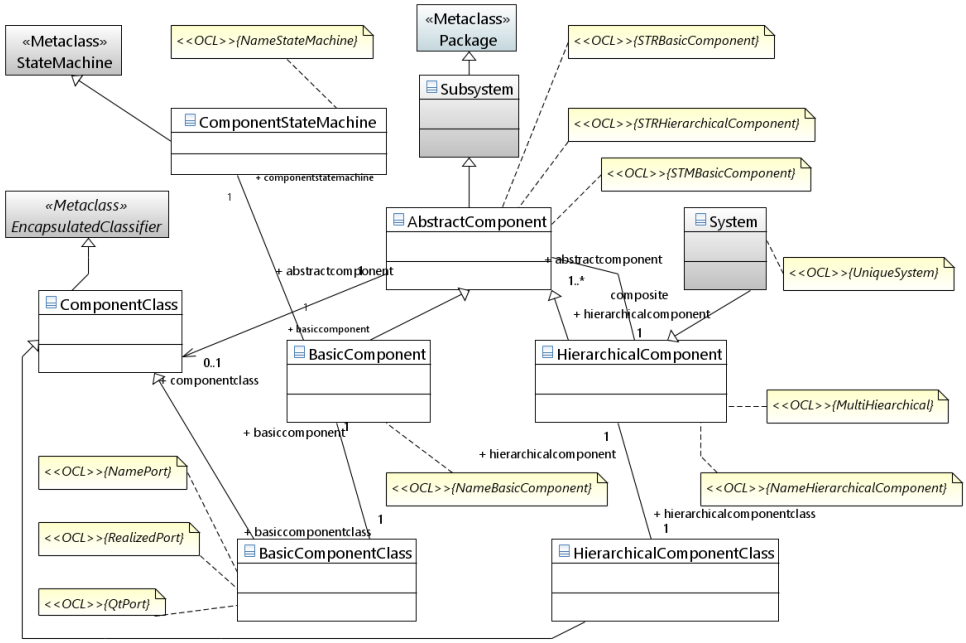
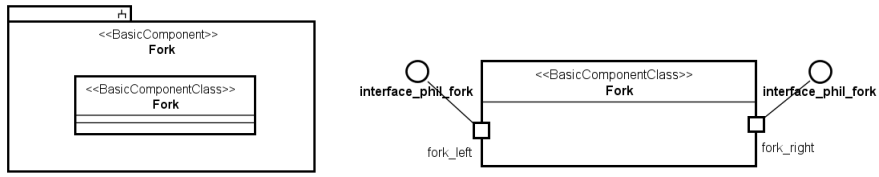


Fig. 3 The Component Metamodel



(a) Basic Component

(b) Fork Component Class

Fig. 4 Fork Component Perspectives

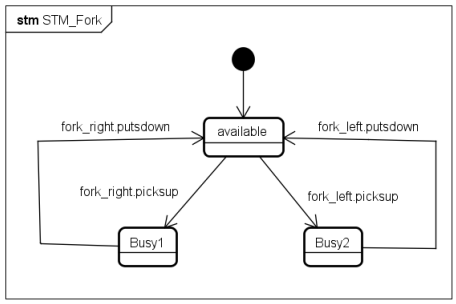


Fig. 5 State Machine STM_Fork

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 3, these conditions are formally specified in OCL (Object Constraint Language)[19]; their titles appear inside notes and their definitions are available in [20]. 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 can have attributes. In OCL, this is captured by the Constraint 1 that is about the *BasicComponentClass* context

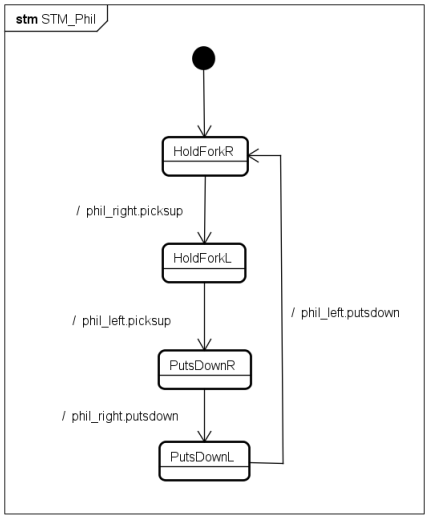


Fig. 6 State Machine STM_Phil

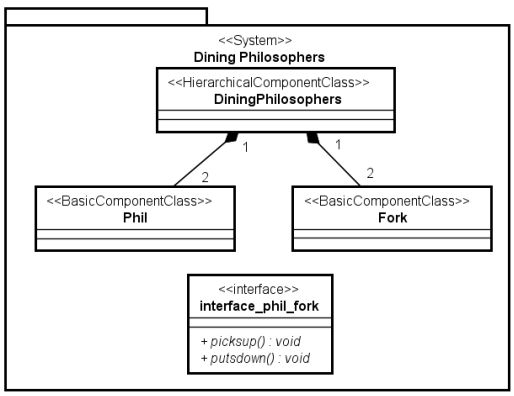


Fig. 7 Hierarchical Component

where the invariant *qtPortBC* determines that the number of ports is at least one; 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*.

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 )
```

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

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

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 hierarchical component. Figure 8 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.

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. 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 perform.

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 7, *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 parameter, 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 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 actions.

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* and are the index 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: a component previously defined. For example, in Figure 8 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.

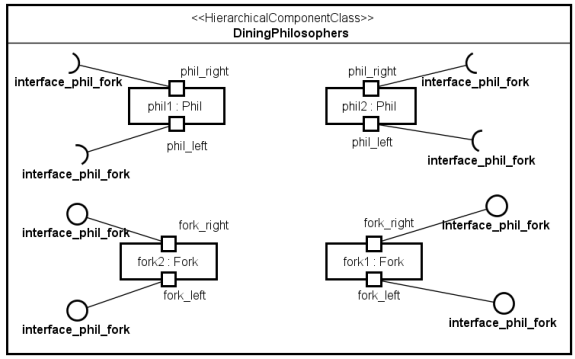


Fig. 8 Interleave

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 one.

Figure 9 illustrates a communication between *fork*1 and *phil*1. 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*.

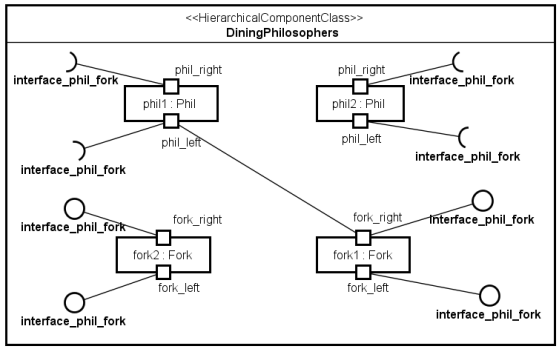


Fig. 9 Communication

4 Formal Semantics and Compositional Verification

In order to perform a mechanised compositional verification during the model construction, we translate the UML models to BRIC, 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 an independent semantics as reference. Unfortunately, there is no complete semantics for UML. However, the problem is minimised in our particular case, since we our semantics is inspired by the Foundational UML works [21–23].

Concerning component interaction, in particular, they are asynchronous: always intermediated by a buffer. 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 schedulability which may impact the order or the priority of messages, is not our concern.

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

4.1 Overview

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.

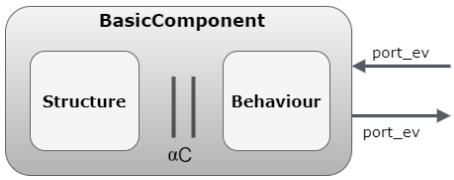


Fig. 10 Illustration of a BasicComponent in CSP

As we have previously explained, our UML Component Model has two types of components: *BasicComponent* and *HierarchicalComponent*. Figure 10 shows the high-level architecture of 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.

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.

A *BasicComponent* allows communication with other components or with the environment

through ports. In Figure 10, 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 to CSP events.

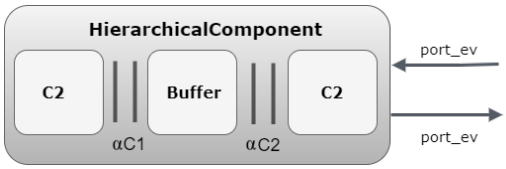


Fig. 11 Illustration of a HierarchicalComponent in CSP

A *HierarchicalComponent* is specified by the parallelism of its internal component instances. As shown in Figure 11, consider a *HierarchicalComponent* that has two internal component instances, namely *C1* and *C2*. 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 9. 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 11, 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 9 the port *phil_left* from the instance *phil*1 of the component *Phil* is connected to the port *fork_right* from the instance *fork*1 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 11 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 semantic functions 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; the translation notation (meta-language) is 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 [20].

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. 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. 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 =

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

As third element of the contract in Rule 2, we have the interface of communications interface(c), Rule 4, that 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 =

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

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.

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

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

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 semantic function $[\![\_]\!]_{\mathcal{C}}$, Rule 6, defines the semantics of a 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$]\!]_{\mathscr{C}}$: CSPSpecification =

if c.basicComponent *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 datatype and two subtypes (one for inputs and another for outputs).

datatype $fork\_operations = picksup\_I \mid picksup\_O \mid putsdown\_I \mid putsdown\_O$
subtype $fork\_I = picksup\_I \mid putsdown\_I$
subtype $fork\_O = picksup\_O \mid putsdown\_O$

Rule 7. Semantics of Basic Component BRIC

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

datatype c.name_operations = 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$]\!]_{\mathcal{STM}}$
mainProcess(c)

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.

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

channel $fork\_right : id\_Fork.fork\_operation$
channel $fork\_left : id\_Fork.fork\_operation$

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.operation
  else
   channel port.name_internal = ID_c.name.operation
   channel port.name_external = ID_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 10). 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 captured by the function varList(c). The function vid calculates 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 with 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. In the case where there is a trigger, a boolean expression is formed of the semantics of a guard, $[\![\_]\!]_{\mathcal{GRD}}$, and the semantics of a trigger, $[\![\_]\!]_{\mathcal{TRG}}$, that 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.

Whenever a transition has both a trigger and a guard, the string that represents this trigger is concatenated with the identifier x. 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 a guard and an action, this guard is associated to a channel named *internal* and, in a similar way to guards and triggers, it has an identifier x to avoid racing conditions.

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]]_GRD → & [[t.trigger]]_TRG.generatedIdTrs()
      → c.name_memory ( id , varList(c))
  elseif(not empty(t.guard) and empty(t.trigger)) then
    [[t.guard]]_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 updates with this value in the subsequent recursive call. Events get e 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.

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

Rule 11 specifies the semantic function $[\![\_]\!]_{STM}$ 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 $[\![\_]\!]_{STATE}$. The first invoked process is the one reachable from the initial pseudostate. It is represented by *c.StateMachine.FirstState.name(id)*.

Rule 11. Semantics of State Machine

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

```
STM_c.name(id) = stm.FirstState.name(id)
for each st in stm.State
  st.name(id) = [[st]]_STATE
end for
```

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

Rule 12. Semantics of State

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

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

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 an 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 the event described in Rule 10.

Rule 13. Semantics of Transition

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

```
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 6 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 6. It cyclically picks up a fork in the right port, then in the left one, and finally puts down via the same port.

$$\begin{aligned}
&STM_Phil(id) = HoldForkR(id)\\
&HoldForkR(id) = (phil_right.id.picksup_I \rightarrow\\
&\quad phil_right.id.picksup_O \rightarrow HoldForkL(id))\\
&HoldForkL(id) = (phil_left.id.picksup_I \rightarrow\\
&\quad phil_left.id.picksup_O \rightarrow PutsDownR(id))\\
&PutsDownR(id) = (phil_right.id.putsdown_I \rightarrow\\
&\quad phil_right.id.putsdown_O \rightarrow PutsDownL(id))\\
&PutsDownL(id) = (phil_left.id.putsdown_I \rightarrow\\
&\quad phil_left.id.putsdown_O \rightarrow HoldForkR(id))
\end{aligned}$$

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.

$$\begin{aligned}
&FORK(id) = STM_Fork(id)\\
&PHIL(id) = STM_Phil(id)
\end{aligned}$$

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 15, the semantic function $[\![\_]\!]_{\mathcal{HC}}$ 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)*. The ports from a *HierarchicalComponent* 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 16) that orchestrates the message communication order; the synchronisation set is formed of the ports involved in the connection ($i.port[1]$ and $i.port[2]$).

Rule 15. Semantics of Hiearchical Component Behaviour

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

let
  CON(0) = (||| $i : c.instances \bullet i.name(id)$)
    ||| portProcess(c)
  for each i in c.connections
  CON(i) = CON($i-1$) $\underset{\{|i.port[1], i.port[2]|\}}{||}$ Buffer($i.port[1]$,$i.port[2]$)
  end for
within
  $c.name(id)$ = CON($length(c.connections)$)

Figure 12 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 connection between components instances.

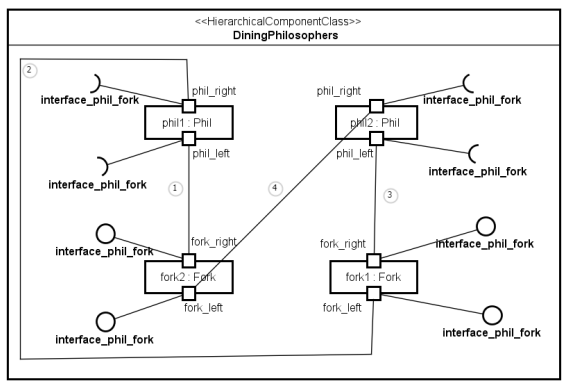


Fig. 12 Dining Philosophers - connections

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, Rule 16, with two channels, one input (ci) and one output (co) of the same type. 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 16. 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, directions, Rule 17.

Rule 17. Function Buffer Auxiliary Process

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

let
  $B(\langle\rangle) = \Box\ x : outputsC\_All(ci) \bullet ci.x \rightarrow B(\langle x \rangle)$
  $B(s) = (co!head(s) \rightarrow B(tail(s)))$
    $\Box$
      $(\#s < n \quad \&$
    $\Box\ x : outputsC\_All(ci) \bullet ci.x \rightarrow \quad B(s \frown \langle x \rangle))$
within
  $B(\langle\rangle)$

Once the semantics is defined, it is possible to translate well-formed UML component models to BRIC components and, afterwards, 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 [24]. 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 13.

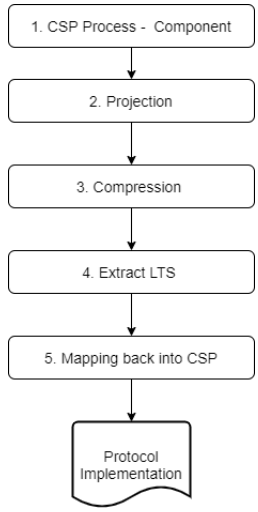


Fig. 13 Protocol Implementation Steps

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 the process P with the projection over the communication channels: $Q = P \setminus (diff(\Sigma, C))$, 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, GLTS (Generalised LTS) [25], 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 13, an LTS is extracted from the GLTS, and these τ self-loop 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 [26], 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 6. 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 [25].

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 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 checks some essential properties of a component, such as whether its behaviour conforms to an I/O Process, described in Section 2.3.

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 until the point where the violation occurs. This may help 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 14. However, in this example, the deadlock will always be found when the last connection is processed, since this is the one that entails a cyclic communication topology. Suppose the dotted line between the component instances *fork*2 and *phil*2 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.

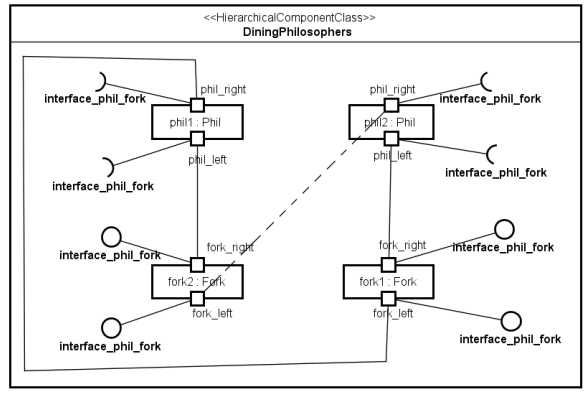


Fig. 14 Deadlock

Figure 15 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 a service picking up, to *fork2* that offers it. 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.

These different traceability mechanisms may 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 [7].

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 [14]. 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.

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, and 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 in that it must obey 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 communication 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 protocol (Section 5.2).

5.1 Ring Buffer

A Ring Buffer is a reactive bounded buffer composed by a ring of storage cells with a controller

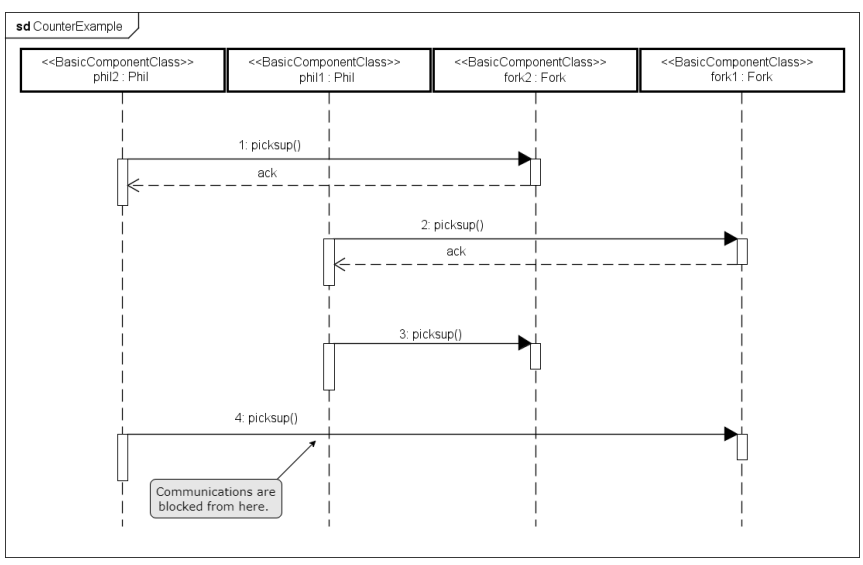


Fig. 15 Automatically generated deadlock trace as a Sequence Diagram

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.

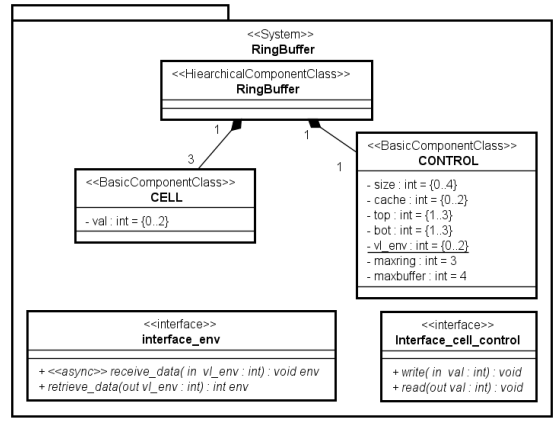


Fig. 16 Ring Buffer Component

Part of the structure of the Ring Buffer component model in UML is shown in Figure 16. It is a *System component* composed by 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, *maxring*, defined as $maxbuffer - 1$, which gives the bound for the ring considering the

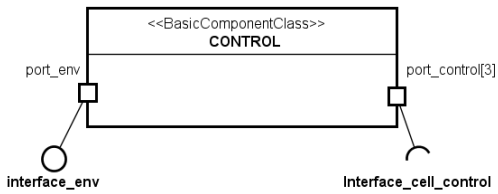


Fig. 17 CONTROL Component

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 a return from the call. 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 17 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 18. 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*, *Read* and *Write*. The *Init* state has a choice between *Read* and *Write* states where the *read()* and *write()* operations 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 *top*, 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. 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. It is fully described in [20].

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

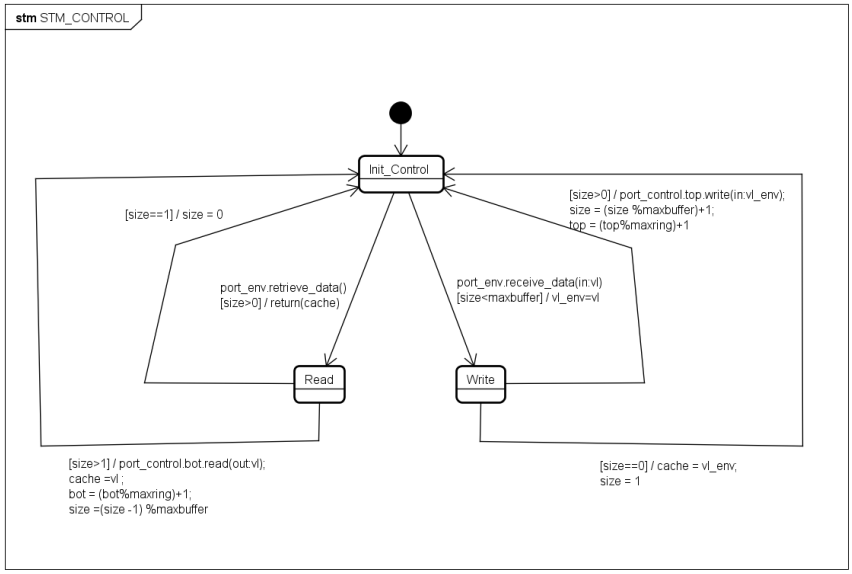


Fig. 18 State Machine of CONTROL Component

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 $size > 0 \ \& \ port\_env.id.1.retrieve\_data!cache$, 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 $size == 1 \ \& \ internal.3$. The event *internal.3* is used in *STM_CONTROL* in order to allow the synchronisation when the guard holds.

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

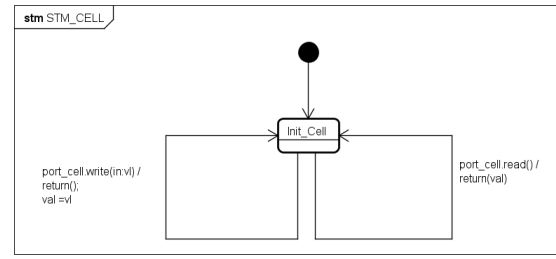


Fig. 19 State Machine of the CELL Component

The process *STM_CONTROL* is the outcome of the application of the rules described in Section 4.2 related to the state machine diagram. Each state of the diagram becomes a process parameterised by the identifier *id* that represents a unique component.

$STM\_CONTROL(id) = Init\_Control(id)$
$Init\_Control(id) =$
$(port\_env.id.1.retrieve\_data\_I \rightarrow$
$get\_control\_cache.id?cache \rightarrow$
$port\_env.id.1.retrieve\_data\_O?cache \rightarrow Read(id))$
$\Box$
...
$Read(id) =$
$(internal.3 \rightarrow set\_control\_size.id!0 \rightarrow$
$Init\_Control(id))$
$\Box$
...
$Write(id) =$
...
$\Box$
$(internal.6 \rightarrow get\_control\_vl\_env.id?vl\_env \rightarrow$
$set\_control\_cache.id!vl\_env \rightarrow$
$set\_control\_size.id!1 \rightarrow Init\_Control(id))$

The other component of the Ring Buffer is the *CELL* and its behaviour is modelled in Figure 19. 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:

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

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.

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

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

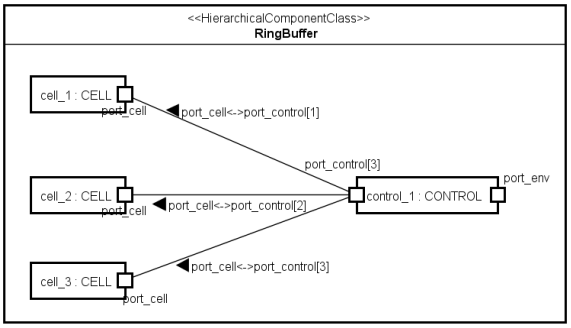


Fig. 20 Ring Bufffer Component

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 20 shows an example configuration of the hierarchical component *Ring Buffer* that is composed by 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].

Some errors can be introduced when designing a component. For instance, Figure 21 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. 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 record 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 22. 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 maximum 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 23 the fourth instance of *CELL*, *cell_4*, is connected to the port *port_control* in the index 3 that is also connected to the *cell_3*.

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 20, 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 24. It is also exposed from another perspective through composite structure diagram, Figure 25. A dotted line between instances *control_1* and *cell_2* evidences the connection in which the deadlock occurs.

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 B&O [27]. 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 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.

Figure 26 shows the state machine with the behaviour 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 for a node a timeout information by *bus_receive.receive_pack(TIMEOUT)*, and another transition that is triggered to identify an online status from a node by *bus_sender.send_status(ISON)*; 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, if it is not empty ($data\ !=\ EMPTY$), by *bus_receiver.receive_pack(TIMEOUT)*.

Figure 27 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, it is able to receive ($node\_receiver.receive\_pack$) and send ($men\_node.men\_pack(pack)$) information. When

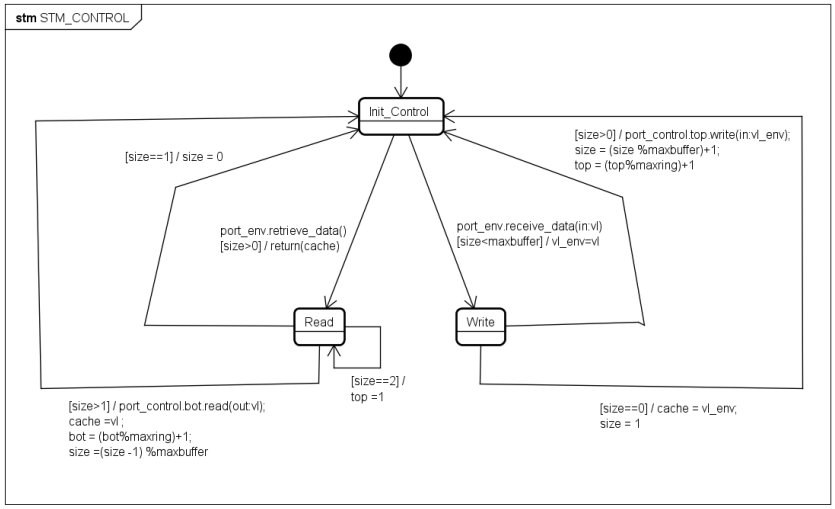


Fig. 21 State Machine of Control Component.

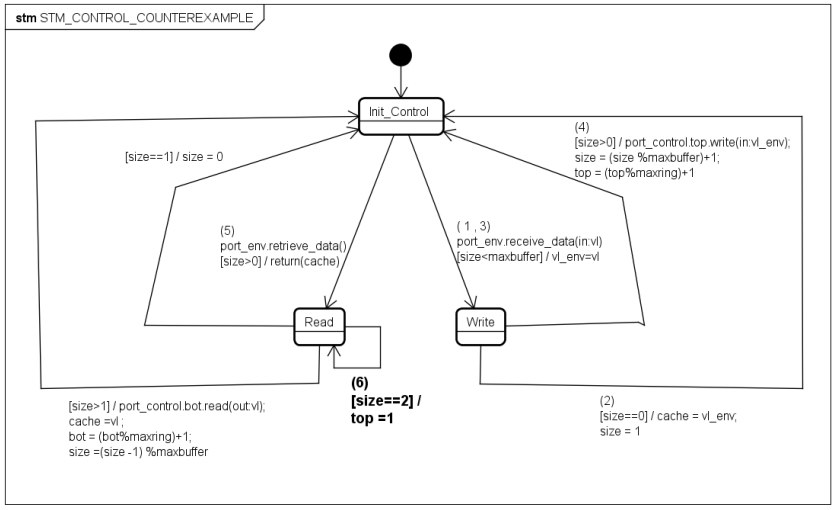


Fig. 22 State Machine with Divergence Counterexample

in the state *Choice*, the role of the node is determined: *undecided*, *follower* or *leader*. This decision is given by the priority of a node and its claim. For instance, if *myclaim* = *leader* and the number of leaders in the other nodes is greater than zero ($nrLeaders > 0$), the current node change its claim to *undecided*.

Figure 28 shows the *LeaderShip* System composed by the basic components *NODE* and *BUSCELL*. As participants interact sending messages from one participant's transmitter to another participant's receiver for a 2-Node configuration the number of instances of *NODE* and *BUSCELL* is the same; in our example, two. *NODE* and *BUSCELL* realise two interfaces: *interface_receiver* and *interface_sender*. The former provides an operation that is responsible for receiving the status of its peers, *receive_status*; this operation is asynchronous. This interface also has 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.

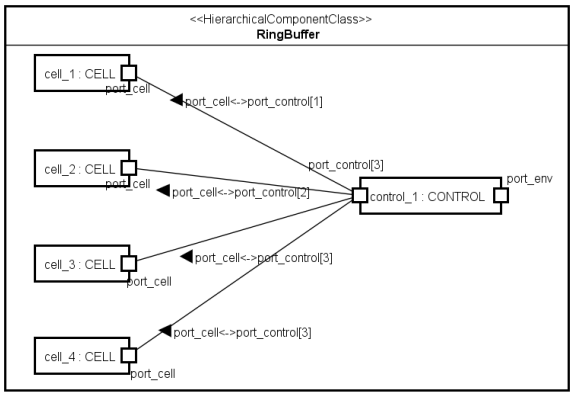


Fig. 23 Ring Bufffer Component with 4 cells

The communication among the instances of these components is illustrated in Figure 29. In this diagram, there are two instances of the *NODE* component that require the interfaces *interface_receiver* and *interface_sender*, and two instances of the *BUSCELL* component that provide the same interfaces. Connections are made between conjugated ports. For example, the 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*.

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 guard, *(data != EMPTY)*, in the self-transition of *CellIdle* state that can be triggered by *bus_receiver.receive_pack*. This transition has its trigger, guard and action marked in boldface in Figure 30. In this context, a node can send information to the *buscell*, and then the control moves from *CellIdle* to the *CellOn* state. However, in the *CellOn* state, the *buscell* is not allowed to receive information from a node since the guard will never hold.

This malfunction is traced back as a sequence diagram, as shown in Figure 31.

6 Related Work

There are several approaches to define component models and verification strategies, which are based on a variety of formalisms.

For instance, the Foundational UML Subset (fUML) [21] provides a precise semantics for UML classes, activities and actions. The operational semantics of fUML maps UML activities to 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) [28]. Precise Semantics of UML State Machine (PSSM) [22] is an extension of fUML that defines a foundational semantics for UML state machines. Precise Semantics of UML Composite Structure (PSCS) [23] 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 Section 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 [4, 29] 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) [5, 6] 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

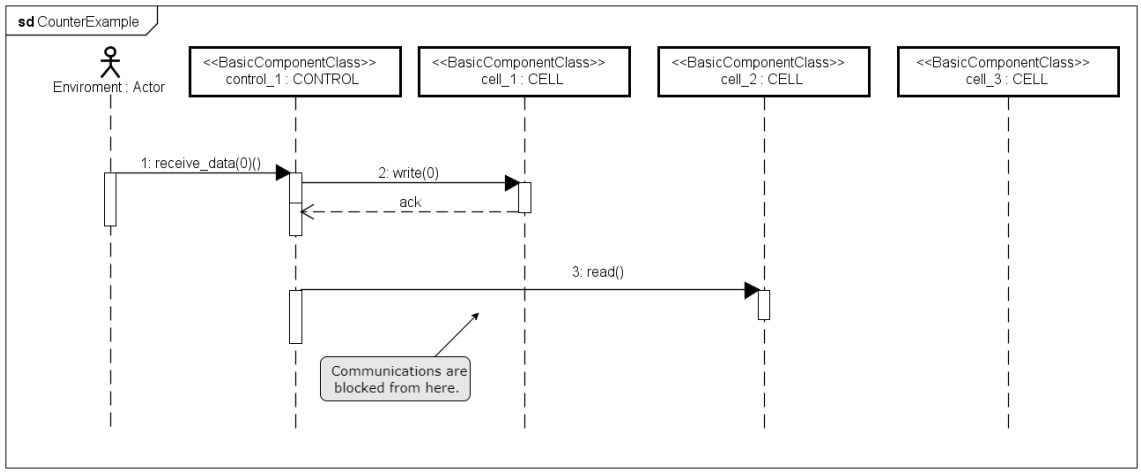


Fig. 24 Automatically generated Sequence Diagram with deadlock

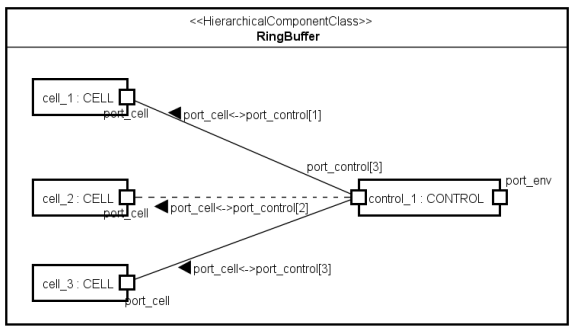


Fig. 25 Ring Buffer Deadlock Traceability

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 [30] 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 the 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 [31] provides a formalisation of the Gamma Composition Language that is part of the Gamma Framework [32]. 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 [7], which is an extension of [31, 32], the authors proposed a cloud-based, end-to-end verification workflow for SysML [8] 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 to component design and verification are compositional.

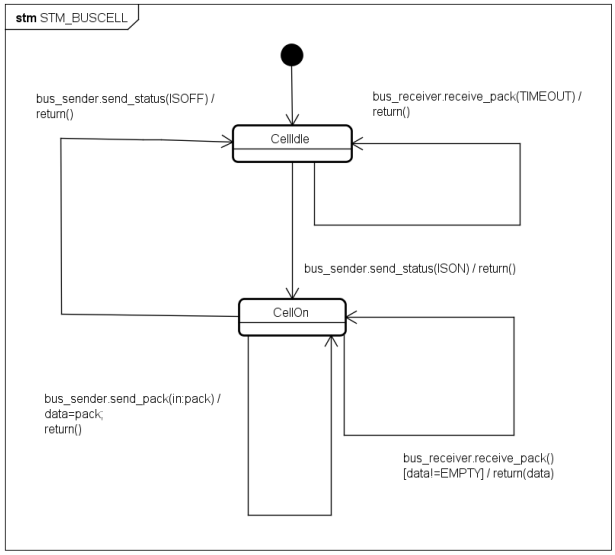


Fig. 26 BUSCELL Component

	Component Based	Approach to Verification	Modeling	Hierarchical Composition	Well-Formedness Conditions	Visual	Formalism	Property
fUML [21]	N/A	N/A	UML	N/A	N/A	N/A	PSL	N/A
rCOs [4]	x	A Posteriori	UML	x			CSP	refinement and deadlock
BIP [6] [30]	x	Compositional	Petri Net extend UML	x			Invariants	Invariance, temporal properties and deadlock
Gamma [31]	x	A Posteriori	UML	x	x		UPPAAL	liveness
Horváth et al. [7]		A Posteriori	SysML		x	x	UPPAAL	liveness
UML-RT [33]	N/A	N/A	UML-RT	x	N/A	N/A	N/A	N/A
BTS [34]		Construtive					CSP	deadlock
Lima et al. [35]		Posteriori	SysML		x	x	CSP	refinement
Gibson et al. [36]		A Posteriori	SysML			x	JPF	fault protection
ProMoBox [37]		A Posteriori	DSML			x	SPIN and Promela	temporal properties
Our Work	x	Compositional	UML	x	x	x	CSP	deadlock

Table 1 Summary of related works.

UML-RT [33] is a UML profile that facilitates the modular development of software systems, following a model based approach. 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 (a reactive class with its own control flow) 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; thus, 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 the one in [38], where the authors present a mapping from UML-RT to the Circus process algebra. As far as we are aware, UML-RT offers no verification strategy to ensure desired properties, such as deadlock freedom. Verification techniques similar to the ones we propose here would also be applicable to UML-RT. Unfortunately, tool support for UML-RT seems to be out-of-date, and the notation is not widely used anymore.

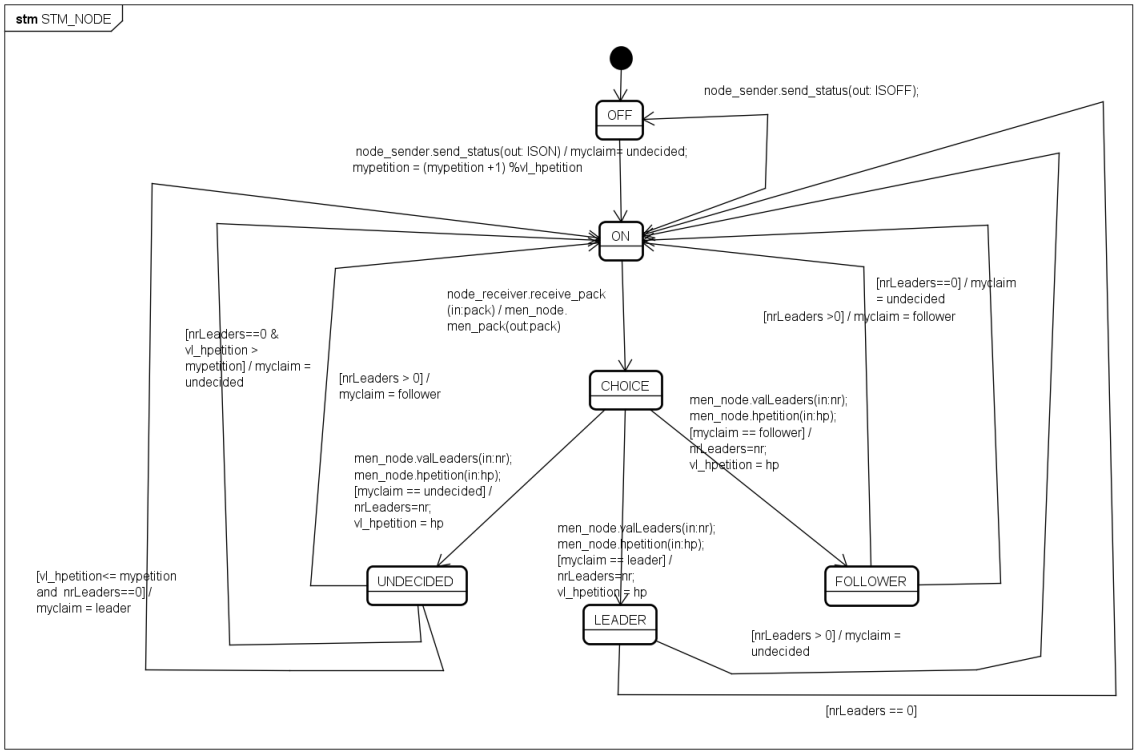


Fig. 27 NODE Component

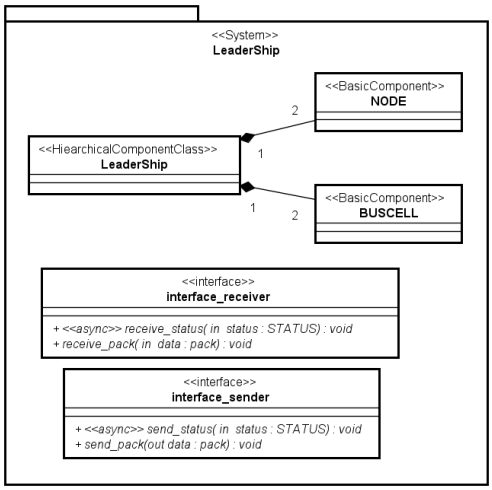


Fig. 28 Leadership Election Component

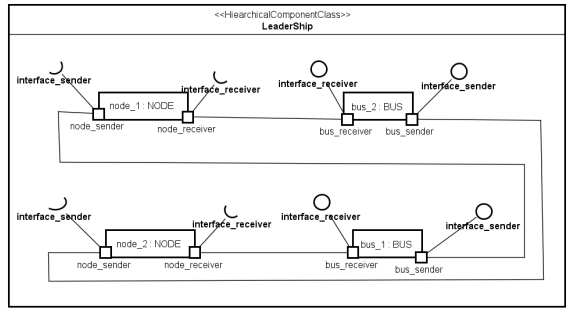


Fig. 29 Leadership Election Composition

In [39], the authors present a formal semantics for a comprehensive subset of SysML [8] via a mapping into CML [40], a formalism that combines CSP and VDM [41]. 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 [39] is not on component-based design nor on ensuring property preservation compositionally.

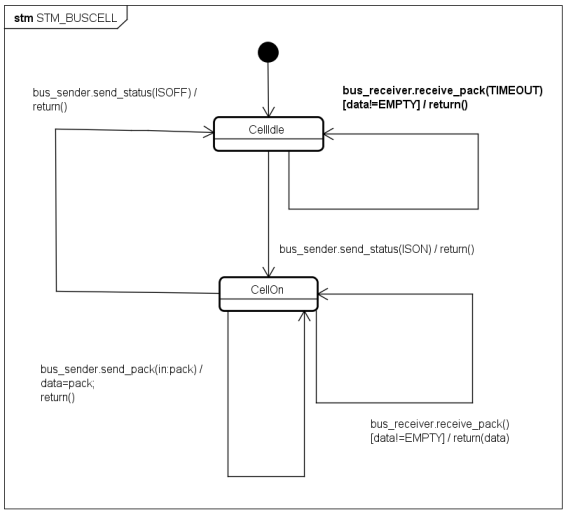


Fig. 30 Component with deadlock

Collaborating SysML state machines are used to model systems in [36], 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 [37] 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 [34] 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.

Table 1 summarises the related work described in this section, 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 *Visual 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 deciseveness, 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.

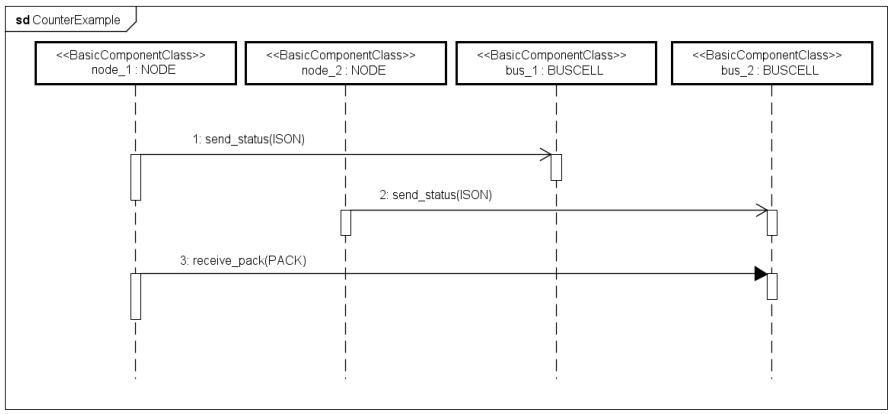


Fig. 31 Sequence Diagram with deadlock

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 for analysis, 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, but they are necessary to ensure the preservation of desired proprieties like deadlock freedom. 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 Section 5.

Our view in this paper 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 have more expressiveness with the addition of advanced constructors in state machines 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 [42] 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; 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. We have already developed a theoretical infrastructure for compositional livelock analysis [43] as well as for analysis of determinism [44]. We plan to integrate these theoretical results into the proposed component model and related verification strategy.

References

[1] Booch, G., Rumbaugh, J. & Jacobson, I. *The unified modeling language user guide* (Addison-Wesley, Upper Saddle River, NJ).

[2] Oliveira, M. V. M. *et al.* Rigorous development of component-based systems using component metadata and patterns. *Formal Aspects of Computing* **28** (6), 937–1004 (2016) .

[3] Clarke, E. M. & Wing, J. M. Formal methods: State of the art and future directions. *ACM Computing Surveys* **28**, 626–643 (1996) .

[4] Chen, Z., Liu, Z., Ravn, A. P., Stolz, V. & Zhan, N. Refinement and verification in component-based model-driven design. *Sci. Comput. Program.* **74** (4), 168–196 (2009) .

[5] Bonakdarpour, B., Bozga, M., Jaber, M., Quilbeuf, J. & Sifakis, J. A framework for automated distributed implementation of component-based models. *Distributed Computing* **25** (5), 383–409 (2012) .

[6] Basu, A., Bozga, M. & Sifakis, J. *Modeling heterogeneous real-time components in bip*, SEFM '06, 3–12 (2006).

[7] Horváth, B. *et al.* *Model checking as a service: Towards pragmatic hidden formal methods* (2020).

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

[9] Ramos, R. T. *Systematic Development of Trustworthy Component-based Systems*. Ph.D. thesis, Brazil (2011).

[10] Ramos, R., Sampaio, A. & Mota, A. Cavalcanti, D. R., Anaand Dams (ed.) *Systematic development of trustworthy component systems.* (ed.Cavalcanti, D. R., Anaand Dams) *FM 2009: Formal Methods*, 140–156 (Springer Berlin Heidelberg, Berlin, Heidelberg, 2009).

[11] Roscoe, A. W. *The Theory and Practice of Concurrency* (Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997).

[12] Falcão, F., Lima, L. & Sampaio, A. Massoni, T. & Mousavi, M. R. (eds) *Safe and constructive design with uml components.* (eds Massoni, T. & Mousavi, M. R.) *Formal Methods: Foundations and Applications*, 234–251 (Springer International Publishing, Cham, 2018).

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

[14] Vision, C. Astah - premier diagramming, modeling software tools (2022). URL https://astah.net/.

[15] Schneider, S. *Concurrent and Real Time Systems: The CSP Approach* 1st edn (John Wiley and Sons, Inc., New York, NY, USA, 1999).

[16] Gorrieri, R. & Versari, C. *Introduction to Concurrency Theory: Transition Systems and CCS* 1st edn (Springer Publishing Company, Incorporated, 2015).

[17] Heineman, G. T. & Councill, W. T. (eds) *Component-based Software Engineering: Putting the Pieces Together* (Addison-Wesley Longman Publishing Co., Inc.,

Boston, MA, USA, 2001).

[18] kiu Lau, K. & Wang, Z. A survey of software component models. Tech. Rep., in Software Engineering and Advanced Applications. 2005. 31 st EUROMICRO Conference: IEEE Computer Socity (2005).

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

[20] Falcão, F., Lima, L., Sampaio, A. UML Component-Based Design Using a Safe Stepwise Strategy - Extend Report. (https://github.com/flaviafalcao/ExtendedReport) (2022).

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

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

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

[24] Gibson-Robinson, T., Armstrong, P., Boulgakov, A. & Roscoe, A. in *Fdr3: A modern refinement checker for csp* (eds Abraham, E. & Havelund, K.) *Tools and Algorithms for the Construction and Analysis of Systems*, Vol. 8413 of *Lecture Notes in Computer Science* 187–201 (Springer Berlin Heidelberg, 2014).

[25] Boulgakov, A., Gibson-Robinson, T. & Roscoe, A. W. Merz, S. & Pang, J. (eds) *Computing maximal bisimulations.* (eds Merz, S. & Pang, J.) *Formal Methods and Software Engineering*, 11–26 (Springer International Publishing, Cham, 2014).

[26] Sampaio, A., Nogueira, S., Mota, A. & Isobe, Y. Sound and mechanised compositional verification of input-output conformance. *Software Testing, Verification and Reliability* **24** (4), 289–319 .

[27] Company, B. B&o softwares (2022). URL http://www.bang-olufsen.com/.

[28] Grüninger, M. & Menzel, C. The process specification language (psl) theory and applications. *AI Magazine* **24**, 63–74 (2003) .

[29] Chen, Z., Morisset, C. & Stolz, V. *Specification and validation of behavioural protocols in the rcos modeler*, FSEN'09, 387–401 (Springer-Verlag, Berlin, Heidelberg, 2010).

[30] Chehida, S., Baouya, A. & Bensalem, S. Salaün, G. & Wijs, A. (eds) *Component-based approach combining uml and bip for rigorous system design.* (eds Salaün, G. & Wijs, A.) *Formal Aspects of Component Software*, 27–43 (Springer International Publishing, Cham, 2021).

[31] Graics, B., Molnár, V., Vörös, A., Majzik, I. & Varró, D. Mixed-semantics composition of statecharts for the component-based design of reactive systems. *Software and Systems Modeling* **19**, 1–35 (2020) .

[32] 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*, 113–116 (2018).

[33] Selic, B., Gullekson, G. & Ward, P. T. *Real-Time Object-Oriented Modeling* (John Wiley and Sons, Inc., USA, 1994).

[34] Pereira, D. I. A., Oliveira, M. V. M. & Silva, S. R. R. Tool support for formal component-based development (2016).

[35] Lima, L. *et al.* An integrated semantics for reasoning about sysml design models using refinement. *Softw. Syst. Model.* **16** (3) (2017) .

[36] Abstractions for executable and checkable fault management models. *Procedia Computer Science* **28**, 146–154 (2014). 2014 Conference on Systems Engineering Research .

[37] Meyers, B. *et al.* Combemale, B., Pearce, D. J., Barais, O. & Vinju, J. J. (eds) *Promobox: A framework for generating domain-specific property languages.* (eds Combemale, B., Pearce, D. J., Barais, O. & Vinju, J. J.) *Software Language Engineering*, 1–20 (Springer International Publishing, Cham, 2014).

[38] Ramos, R., Sampaio, A. & Mota, A. Steffen, M. & Zavattaro, G. (eds) *A semantics for uml-rt active classes via mapping into circus.* (eds Steffen, M. & Zavattaro, G.) *Formal Methods for Open Object-Based Distributed Systems*, 99–114 (Springer Berlin Heidelberg, Berlin, Heidelberg, 2005).

[39] Lima, L. *et al.* An integrated semantics for reasoning about sysml design models using refinement. *Software & Systems Modeling* **16** (3), 875–902 (2017) .

[40] Woodcock, J. *et al. Features of cml: A formal modelling language for systems of systems*, 1–6 (2012).

[41] Fitzgerald, J. & Larsen, P. G. *Modelling Systems: Practical Tools and Techniques in Software Development* (Cambridge University Press, New York, NY, USA, 2009).

[42] Cavalcanti, A. *et al. Modelling and verification for swarm robotics* (2018). To appear.

[43] Filho, M. C., Oliveira, M. V. M., Sampaio, A. & Cavalcanti, A. Compositional and local livelock analysis for CSP. *Inf. Process. Lett.* **133**, 21–25 (2018) .

[44] Otoni, R., Cavalcanti, A. & Sampaio, A. da Costa Cavalheiro, S. A. & Fiadeiro, J. L. (eds) *Local analysis of determinism for CSP.* (eds da Costa Cavalheiro, S. A. & Fiadeiro, J. L.) *Formal Methods: Foundations and Applications - 20th Brazilian Symposium, SBMF 2017, Recife, Brazil, November 29 - December 1, 2017, Proceedings*, Vol. 10623 of *Lecture Notes in Computer Science*, 107–124 (Springer, 2017).

Acknowledgments. We would like to thanks Sidney Nogueira, Dalay Almeida and Marcel Oliveira for important feedback in earlier versions of this paper. This work is partially funded by CNPq, Grant 432198/2018-0.

Appendix A Well-formedness conditions

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

The first 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 AbstractComponent

inv STMBasicComponent:
 self.oclIsTypeOf(BasicComponent)
  implies
   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
```

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* contains 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 )
```