Grant Agreement: 287829
Comprehensive Modelling for Advanced Systems of Systems

Compositional Analysis and Design of CML Models

Deliverable Number: D2.4.4
Version: 2.0
Date: September, 2014
Public Document

http://www.compass-research.eu
Contributors:

Marcel Oliveira, UFPE  
Augusto Sampaio, UFPE  
Pedro Antonino, UFPE  
Jose Dihego Oliveira, UFPE  
Madiel Conserva Filho, UY  
Jeremy Bryans, UNEW

Editors:

Marcel Oliveira, UFPE  
Augusto Sampaio, UFPE

Reviewers:

Peter Gorm Larsen, IHA  
John Fitzgerald, UNEW
## Document History

<table>
<thead>
<tr>
<th>Ver</th>
<th>Date</th>
<th>Author</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>01-07-2014</td>
<td>Marcel Oliveira</td>
<td>Initial version of Introduction section</td>
</tr>
<tr>
<td>0.2</td>
<td>01-07-2014</td>
<td>Jeremy Brans</td>
<td>Initial version of section describing the Leadership Election formalisation</td>
</tr>
<tr>
<td>0.3</td>
<td>01-07-2014</td>
<td>Pedro Antonino</td>
<td>Initial version of section describing the Dining Philosophers formalisation</td>
</tr>
<tr>
<td>0.4</td>
<td>01-07-2014</td>
<td>Jose Dihego Oliveira</td>
<td>Initial version of section on Substitutability</td>
</tr>
<tr>
<td>0.5</td>
<td>01-07-2014</td>
<td>Madiel Conserva Filho</td>
<td>Initial version of section on Livelock Analysis</td>
</tr>
<tr>
<td>0.6</td>
<td>01-07-2014</td>
<td>Augusto Sampaio</td>
<td>Initial version of section on Timed Behaviour</td>
</tr>
<tr>
<td>0.7</td>
<td>01-08-2014</td>
<td>Marcel Oliveira</td>
<td>Initial version of the background section</td>
</tr>
<tr>
<td>0.8</td>
<td>01-08-2014</td>
<td>Pedro Antonino</td>
<td>Initial version of the section on Behavioural Patterns for Deadlock Analysis</td>
</tr>
<tr>
<td>0.9</td>
<td>15-08-2014</td>
<td>Marcel Oliveira</td>
<td>Initial version of the Service Conformance</td>
</tr>
<tr>
<td>1.0</td>
<td>16-08-2014</td>
<td>Marcel Oliveira</td>
<td>First Full Version for Local Review</td>
</tr>
<tr>
<td>1.1</td>
<td>30-08-2014</td>
<td>All Authors</td>
<td>Corrections based on the results of the local review</td>
</tr>
<tr>
<td>1.2</td>
<td>09-09-2014</td>
<td>Marcel Oliveira</td>
<td>Version for Compass Internal Review</td>
</tr>
<tr>
<td>2.0</td>
<td>30-09-2014</td>
<td>Marcel Oliveira</td>
<td>Final Version</td>
</tr>
</tbody>
</table>
Abstract

In our previous Deliverable D24.1 on Compositional Analysis and Design of CML Models [OSA13] we proposed a correct-by-construction approach, BRIC, for building CML models of trustworthy SoS. This approach contained a comprehensive set of composition rules, which can be used to systematically develop a wide variety of trustworthy systems, guaranteeing, by construction, the absence of deadlock. The approach covered not only tree-topologies, but also topologies with cycles in a compositional method, without being aware of needing to know the overall structure of the system. Our approach was improved using enriched components with metadata, which extended our approach for arbitrary components with improved and lightweight side conditions. As a result, the effort of verification is significantly reduced (particularly for noncyclic networks).

The approach proposed in [OSA13] had some limitations. First, the benefits of using metadata were limited to the application of composition and feedback composition rules. In other words, although systematic, our approach was not local for cyclic communicating systems, potentially presenting a state explosion in the verification of such systems.

In this deliverable, besides further improving BRICK, we provide further scientific contributions, which can be summarised as follows.

Section 4 A strategy for local deadlock analysis of cyclic networks based on adherence to some formalised behavioural patterns.

Section 5 A compositional strategy for livelock analysis.

Section 6 A notion of service conformance that ensures the preservation of the constituent functional behaviour that does not involve interaction with other constituents.

Section 7 A notion of substitutability at the BRICK level, allowing replacing constituents with valid refinements.

Section 8 An approach to handle analysis involving timed behaviour.
Contents

1 Introduction 9

2 Background 14
   2.1 CSP ................................................. 14
      2.1.1 CSP semantic models ....................... 17
   2.2 CML .............................................. 24
   2.3 BRICK: Systematic Development of Trustworthy Component Systems .............................................. 30
      2.3.1 Component Model ......................... 30
      2.3.2 Extended Component Model ............... 37

3 Case Studies 44
   3.1 Dining Philosophers ............................. 44
      3.1.1 The BRIC approach ....................... 48
   3.2 Leadership Election ............................. 55
      3.2.1 Leadership Election Formal Model ......... 56
      3.2.2 The BRIC approach ....................... 66

4 Behavioural Patterns for Local Deadlock Analysis 76
   4.1 Behavioural Patterns ......................... 77
   4.2 Client/Server pattern ......................... 82
   4.3 Resource allocation pattern ................. 90
   4.4 Async Dynamic pattern ....................... 95
   4.5 Experimenting on the case studies ........... 106
      4.5.1 Asymmetric dining philosophers .......... 107
      4.5.2 Leadership election algorithm .......... 110
   4.6 Automating the extended strategy .......... 114

5 Preserving Livelock 115
   5.1 Compositional Livelock Analysis ............. 115
   5.2 A General Investigation on Livelock ......... 117
   5.3 Livelock in BRIC ............................... 119
   5.4 Livelock Analysis ............................. 120
   5.5 Local Livelock Analysis ...................... 121
      5.5.1 Conditions for Livelock Freedom ........ 126
      5.5.2 Safe Simple Composition ................. 127
      5.5.3 Safe Complex Composition ............... 128
      5.5.4 Dealing with Metadata .................. 131
   5.6 Experimental analysis ...................... 134
List of Figures

1. FDR GUI .................................................. 15
2. A Simple CSP Example ................................. 16
3. Result of Interleaving forks and philosophers ........ 51
4. Result of Communication composition ................. 52
5. Result of Feedback compositions ....................... 54
6. Result of final reflexive composition .................. 54
7. Representation of supernode 0 ......................... 71
8. Representation of supernode 0_2 ....................... 73
9. Representation of the Core ............................. 74
10. Representation of the System ........................ 75
11. Results of experiments for the dining philosophers .... 109
12. Results of experiments for the leadership election .. 113
13. Example of 3 philosophers ........................... 135
14. FDR Results for the dining philosophers .............. 143
15. A heterogeneous framework for analysis of timed programs in *Circus* .......................... 161
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Summary of the Experiment’s Results (livelock-free version)</td>
<td>136</td>
</tr>
<tr>
<td>2</td>
<td>Summary of the Experiment’s Results (version with livelock)</td>
<td>137</td>
</tr>
<tr>
<td>3</td>
<td>Deadlocks semantic clauses</td>
<td>169</td>
</tr>
<tr>
<td>4</td>
<td>Revivals semantic clauses</td>
<td>170</td>
</tr>
</tbody>
</table>
1 Introduction

Compositional approaches (both for development and analysis) have been around for a long time [Mah90]. Over the last decade, however, they have re-emerged as a promising paradigm to deal with the ever increasing need for mastering complexity, evolution and reuse in the design of computer based systems. The basic motivation for this paradigm is to replace conventional programming with the composition and configuration of reusable and independent constituents, in such a way that the resulting system exhibits some desired properties (like deadlock and livelock freedom) by construction.

In the context of the overall scope of the COMPASS project, the current deliverable contributes (on top of the results presented by Deliverable D24.1) with a constructive approach to compositional analysis and design of CML models. The main challenge is to make static analysis (particularly via model checking) feasible for SoS, which tend to be larger and more complex than ordinary component systems. In D24.1, we focused on deadlock verification. Here, we further develop the deadlock analysis strategy to consider some communication patterns (with proven significant gain in efficiency) and extend the approach to livelock analysis, as another classical property. We also define notions and corresponding analysis mechanisms for service conformance and substitutability via refinement. Verification including time is briefly addressed. The remain of the introduction details what has been achieved in the previous and in the current deliverable.

In Deliverable D24.1 on Compositional Analysis and Design of CML Models [OSA13] we propose a correct-by-construction approach for building CML models of trustworthy SoS. The BRIC approach focuses on performing analyses that are intended to address engineering concerns on compositional development. The entire approach is based on the original approach from [Ram11] that is underpinned by the CSP process algebra, which offers rich semantic models that support a wide range of process verifications, and comparisons. The strategy for lifting the entire approach from CSP to CML (via Circus) provided a general theoretical link between these three formal languages that fosters the reuse of practical results achieved in any of them.

The CML BRIC model is aligned to other models with behaviour descriptions [ADG98, BCD02, HLL06, BHP06, Arb04, Sif10, CZ07]. It focuses on (re)active components that are input deterministic and output decisive. It considers not only compositions between two distinct constituents, but also the assembly between ports of the same constituent. This brings more flexibility to design decisions at development. An operation for hiding informa-
tion to pack components into black-boxes is also presented.

The correct-by-construction approach for building trustworthy CML SoS [OSA13], contained a comprehensive set of composition rules that can be regarded as safe steps in the development. The application of the rules can be used to systematically develop a wide variety of trustworthy systems, and guarantees, by construction, the absence of deadlock. The approach covers not only tree-topologies, but also topologies with cycles in a compositional method, without being aware of needing to know the overall structure of the system. Port protocols play an important role in the approach, and, in conjunction with other properties, help to alleviate verifications by supporting local analyses.

The verification of our approach [OSA13] was improved using enriched components with metadata. We proposed an integrated correct-by-construction approach for component contracts using metadata, BRICK, which extended our approach for arbitrary components with improved and lightweight side conditions. Metadata are derived from component-contract elements and are used in substitution to heavier verifications in the version without metadata. Additionally, metadata of compositions can be easily derived from the metadata of its constituting components. As a result, the effort of verification is significantly reduced (particularly for noncyclic networks).

The approach proposed in [OSA13] had some limitations. First, the benefits of using metadata were limited to the application of composition and feedback composition rules. Although this corresponds to two of the four basic proposed composition rules, the application of the other composition rules is compatible with our strategy with metadata. Moreover, one of these composition rules, the interleave one, is already very simple, and does not need further improvements. In other words, although systematic, our approach was not local for cyclic communicating systems, potentially presenting a state explosion in the verification of such systems. This drawback can make BRICK inapplicable to complex cyclic systems.

This and other new scientific contributions of this deliverable are summarised as follows.

- A strategy for local deadlock analysis of cyclic networks based on adherence to some formalised behavioural patterns.
- A compositional strategy for livelock analysis.
- A notion of service conformance that ensures the preservation of the constituent functional behaviour that does not involve interaction with
other constituents.

- A notion of substitutability at the BRICK level, allowing replacing constituents with valid refinements.
- An approach to handle analysis involving timed behaviour.

More details on each of these topics are given in the reminder of the introduction; each topic is the subject of a separate section of this deliverable.

Our strategy to do local deadlock analysis of cyclic models through adherence to communication patterns is presented in Section 4. This work is based on Roscoe’s solution based on architectural patterns that reduce the verification effort, by allowing a local analysis of deadlock, even for cyclic communication topologies [Ros10]. However, some of the existing architectural patterns in the literature (including those in [Ros10]) do not directly match the structure of our case studies presented in Section 3. As a further contribution, we formalise a new pattern that allows local deadlock analysis of the leadership election case study. We use CSP to specify the pattern behaviour and its stable failures refinement to formalise a conformance relation. We compare the application of our local analysis with a standard global analysis using FDR.

By considering particular topologies, based on these patterns, we obtain a very efficient analysis strategy. In the particular case of deadlock freedom, considering the dining philosophers classical example, we achieved an efficiency comparable to tools like the deadlock checker [Mar95] and strategies as proposed in [RGG+95], dramatically increasing the size of networks we can analyse.

In the previous deliverable [OSA13], we focused on assuring only deadlock freedom by construction. There, livelock was not an issue because compositions do not hide the composed channels from the behaviour; they are just removed from the interface avoiding further compositions on them. This gives us a grey-box style abstraction. In this deliverable, we consider a black box composition style, which, due to hiding, can lead to divergences, even considering that the constituents to be composed are livelock free. In Section 5, we present a strategy for local livelock analysis, based on constructive rules similar to those that ensure deadlock freedom, but with additional side conditions.

The composition rules were defined in [OSA13] on the top of the notion of service conformance, which guarantees that unused services of a component are still available after composition. In [Ram11], the degree of satisfaction
of this notion varied from preserving all services (strong conformance) to at least one (weak conformance). In Section 6 we discuss the kind of quality attributes we are concerned with: unused services of a constituent should be still available after composition. Hence, we are interested in preserving behavioural properties. In particular, we support the argument that the preservation of basic properties helps in the preservation of more complex ones. Here, we consider only the strong conformance notion from [Ram11]. This notion of conformance is concerned with the preservation of the behaviour of each individual port-protocol implementation [OSA13]. Here, we named this as protocol conformance. This notion of conformance is concerned only with the maintenance of the behaviour of individual port-protocols. In Section 6 we also introduce a stronger notion of conformance, named behavioural conformance, which guarantees that the overall behaviour of a component is maintained on a given set of communication channels $cs$. Further conformance notions might be interesting for more specific coordination purposes, but are not in the scope of this document.

Besides composition, substitution is another important aspect in the development of SoSs. Most works on substitutability are based on the notions of behavioural subtyping [LW94] [Weh03], which is a strong form of relationship between two (component) types. It requires instances of a subtype and of a supertype to fulfil the principle of type substitutability [LW94]:

An instance of the subtype should be usable wherever an instance of the supertype is expected, without a client being able to tell the difference.

This suggests the use of some form of refinement [Ros98] to formalise behavioural subtyping. Refinement guarantees substitutability of an even stronger form: a system can always be replaced by its refinement without any noticeable difference. For subtyping, we want only a replacement to be unnoticeable at places where a supertype is expected. This is a weaker form of substitutability, but that nevertheless can be characterised in terms of refinement [Weh03]. Different substitutability relations can be defined if we are aware of the context in which the component is. Here we consider substitutability as refinement.

In contrast with composition, substitutability relates constituents that are not currently present in the system (it relates a present configuration with a future one). As a consequence, besides the definition of a substitutability notion, in Section 7 we show that the operators defined in the composition rules are monotonic with respect to the refinement relation.
In this deliverable, we consider two different case studies, which are presented in Section 3. First, in Section 3.1 we present the dining philosophers, a classical concurrency problem that constitutes a small case study that perfectly illustrates the problems we are trying to solve in this deliverable. Next, in Section 3.2 we formalise and analyse a more complex and SoS related case study, a version of the leadership election algorithm developed as case study with B&O\(^1\) which is yet another example of a class of cyclic networks. This kind of algorithm can be used in networks of Audio and Video (AV) systems with up to 32 systems. The sections that follows (Section 4–7) use either the dining philosophers or the leadership election as a case study, or both.

Finally, in Section 8 we take time into consideration. Concerning analysis techniques involving time, our approach is based on that presented in [SCHS10a]. In this technique, a syntactic transformation strategy is used to split a timed program into two parallel components: an untimed program that uses timer events, and a collection of timers. We show that, with the timer events, it is possible to reason about time properties in the untimed language, and so the untimed analysis techniques is directly applicable.

Our results on optimising the compositional approach to deadlock analysis (Section 4), livelock analysis (Section 5), service conformance (Section 6), and substitutability (Section 7) were all developed based on the CSP process algebra [Ros98] because the CML model-checker was still under development. However, an important theoretical link from CSP to CML for processes and refinement presented in the previous deliverable [OSA13], allows us to lift the results to provide a similar systematic approach to build trustworthy CML SoS. Envisioning the SoS context in which our work is inserted, our approach supports asynchronous communications. The main principle for lifting the approach from CSP to CML is to keep the main structure of the definitions and rules. Our link, and hence, our results, are limited to a subset of untimed feasible divergence free CML processes without object-oriented constructs and without undefined expressions with a limited use of predicative specifications.

\(^1\)http://www.bang-olufsen.com/
2 Background

In this section, we provide the technical background of the document. In Section 2.1, we present the process algebra on which the original approach is underpinned, CSP [Hoa85, Ros98]. Next, Section 2.2 presents our extension to CSP, CML [WCF12], a formal specification language that integrates a state based notation (VDM++ [FL09]) and CSP, as well as Dijkstra’s language of guarded commands and the refinement calculus. Finally, we present our component model, BRICK: we present the basic definitions and the composition rules in Section 2.3.1 and the extended counterparts of the definitions and the composition rules in Section 2.3.2.

2.1 CSP

The language of CSP was first described by Hoare [Hoa85]. It is a process algebra that can be used to describe systems composed by interacting components, which are independent self-contained entities (called processes) with particular interfaces that are used to interact with the environment. In [Ros98], a new version of CSP was presented: it differs from Hoare’s version only on the treatment of alphabets. It is the later version that forms the basis of FDR, a tool that model-checks a machine-processable subset of CSP, called CSP\textsubscript{M}, which is a combination of an ASCII version of CSP with an expression language inspired on functional languages. A link between the CSP and CSP\textsubscript{M} syntaxes can be found in [Ros98]. In what follows, we briefly describe the most important CSP constructs.

The two most basic CSP processes are \textit{STOP} and \textit{SKIP}; the former deadlocks, and the latter does nothing and terminates. If \( P \) is a CSP process, and \( a \) an event, then the prefixing \( a \rightarrow P \) is initially able to perform only \( a \), and after performing \( a \) it behaves as \( P \). A boolean guard may be associated with a process: given a predicate \( g \), if the condition \( g \) is true, the process \( g \& P \) behaves like \( P \); it deadlocks otherwise. Processes \( P_1 \) and \( P_2 \) can be combined in sequence using the sequence operator: \( P_1; P_2 \). This process executes the process \( P_2 \) after the execution of \( P_1 \) terminates. The external choice \( P_1 \parallel P_2 \) initially offers events of both processes. The performance of the first event resolves the choice in favour of the process that performs it. Differently from the external choice, the environment has no control over the internal choice \( P_1 \cap P_2 \), in which the process internally (nondeterministically) resolves the choice. The sharing parallel composition \( P_1 \parallel [cs] P_2 \) synchronises \( P_1 \) and \( P_2 \) on the channels in the set \( cs \); events that are not
listed occur independently. Differently, in the alphabetised parallel composition $P_1 \parallel [cs_1 | cs_2] P_2$, the processes $P_1$ and $P_2$ synchronise on the channels that are in the intersection between $cs_1$ and $cs_2$; events that are not in this intersection occur independently. Processes can also be composed in interleaving: in $P_1 || P_2$, both processes run independently. The event hiding operator $P \setminus cs$ is used to encapsulate the events that are in the channel set $cs$. This removes these events from the interface of $P$, which become no longer visible to the environment. CSP also provides finite iterated operators that can be used to generalise the binary operators of sequence, external and internal choice, parallel composition, and interleaving. A few other process constructors are available in CSP but omitted here for conciseness. Furthermore, we also omit the syntax of the expression language accepted in CSP, which can be found in [Ros98].

By way of illustration, we consider the development of a parking spot presented in Figure 2. In CSP, every channel used in the specification must be declared. For instance, channel enter, leave declares the channels enter and leave, which indicate that a customer has entered the parking spot and left it, respectively. Our abstract specification of a parking spot, PARKING_SPOT only requires that two customers cannot enter in sequence; the first one must leave before the next one enters. Using FDR’s assertion commands, we can verify that the abstract specification of the parking spot is deadlock free, divergence free, and deterministic. This is indicated in FDR with a ✓ on the
channel enter, leave

\[ \text{PARKING\_SPOT} = \text{enter} \rightarrow \text{leave} \rightarrow \text{PARKING\_SPOT} \]

datatype \( \text{ALPHA} = a \mid b \)
datatype \( \text{ID} = \text{Letter(ALPHA)} \mid \text{unknown} \)

channel cash, ticket, change : ID

\[ \text{MACHINE} = \text{cash}!id \rightarrow \text{ticket}.id \rightarrow \text{change}.id \rightarrow \text{MACHINE} \]

\[ \text{CUSTOMER}(id) = \]
\[ (\text{enter} \rightarrow \text{cash}!id \rightarrow \]
\[ (\text{ticket}.id \rightarrow \text{change}.id \rightarrow \text{SKIP} \]
\[ \square \text{change}.id \rightarrow \text{ticket}.id \rightarrow \text{SKIP})]; \]
\[ \text{leave} \rightarrow \text{CUSTOMER}(id) \]

\[ \text{PAID\_PARKING\_SPOT} = \]
\[ (\text{CUSTOMER}(\text{Letter}.a) \]
\[ [\{\text{cash, ticket, change}\}] \]
\[ \text{MACHINE}) \setminus \{\{\text{cash, ticket, change}\} \]

Figure 2: A Simple CSP Example

left of the assertion in Figure 1.

Our concrete parking spot, \( \text{PAID\_PARKING\_SPOT} \) is a paid version of a public parking spot with a pay and display parking permit machine that accepts cash, and issues tickets and change. First, we declare a datatype that represents simple identifications. Datatypes can be divided into two groups: basic datatypes and complex datatypes. The former is defined in terms of simple constants and the latter uses constructors that are applied to types. In Figure 2, \( \text{datatype ALPHA} = a \mid b \) defines a datatype \( \text{ALPHA} \): variables of type \( \text{ALPHA} \) can assume either value \( a \) or \( b \). On the other hand, \( \text{datatype ID} = \text{Letter(ALPHA)} \mid \text{unknown} \) defines a \( \text{ID} \) that represents identifications. The constructors \( \text{Letter} \) receives an \( \text{ALPHA} \) value and returns a value of type \( \text{ID} \) (for example, \( \text{Letter}.a \)); another possibility is the \( \text{unknown} \) \( \text{ID} \).

Next, we declare all the new channels that are used in the concrete specification. Then, we declare the \( \text{MACHINE} \) process, which implements the functions of issuing tickets and change, after receiving the cash. After entering the parking spot, a \( \text{CUSTOMER} \) must interact with the ticket \( \text{MACHINE} \).
by inserting the \textit{cash} into it. The \textit{CUSTOMER} can then pick the \textit{ticket} and the \textit{change} in any order, and finally, \textit{leave} the parking spot. In order to uniquely identify each customer, we parameterise the process \textit{CUSTOMER} with an identification, which is used to identify this customer while interacting with the machine via channels \textit{cash}, \textit{ticket}, and \textit{change}. This guarantees that the machine will only issue the ticket and the change to the customer who inserted the cash.

The paid parking spot is modelled by the process \textit{PAID\_PARKING\_SPOT}. It is a parallel composition of the processes \textit{CUSTOMER} and \textit{MACHINE}; they synchronise on \textit{cash}, \textit{ticket} and \textit{change}, which are hidden from the environment. The specification \textit{PAID\_PARKING\_SPOT} must always allow only one customer to enter, and then to leave the parking spot. The assertion \textit{assert PARKING\_SPOT} [$FD = PAID\_PARKING\_SPOT$] captures the failures divergence refinement check to be carried out.

### 2.1.1 CSP semantic models

CSP offers a number of semantical approaches. A process written in CSP may be understood in terms of operational semantics (where the process is transformed to a labelled transition system, with transitions representing communications); or in terms of algebraic semantics (where properties of a process such as equivalence to some other process may be deduced by syntactic transformations on the process text following a set of algebraic laws); or in terms of denotational semantics (where the process corresponds to a value in some mathematical model, typically a complete partial order or a complete metric space). The latter is the one of particular interest for our work.

In what follows we briefly describe the three denotational models: traces, failures and failures-divergences \[\text{[Ros98]}\].

**Traces model.** The traces model $\mathcal{T}$ denotes a CSP process according to its traces, which are the set of sequences of communications in which the process may engage. Let $\mathcal{A}^\triangledown = \Sigma^* \cup \{s \triangledown \langle \triangledown \rangle \mid s \in \Sigma^\triangledown\}$ be the alphabet of communications. Formally in the traces model each process is identified by a set $T \subseteq \mathcal{A}^\triangledown$ that satisfies the following healthiness condition:

\begin{itemize}
  \item \textbf{T1.} $T$ is nonempty and prefix-closed. This means that it always contains the empty trace $\langle \rangle$ and if $s \triangledown t \in T$ then $s \in T$.
\end{itemize}
Given a CSP process $P$, the traces of $P$ are denoted as $\text{traces}(P)$. For example, $\text{STOP}$ never communicates anything: its set of traces consists only of the empty trace $\text{traces}(\text{STOP}) = \{\langle \rangle \}$. Furthermore, the traces of a prefix process are the traces of the prefixed process $P$, each prefixed with the event $a$ first communicated and the empty trace added ($\text{traces}(a \rightarrow P) = \{\langle \rangle \} \cup \{\langle a \rangle \triangleleft s \mid s \in \text{traces}(P)\}$). Details about the other constructors are presented in [Ros98].

A process $C$ is a trace refinement of $A$ if, and only if, it contains all traces within $A$.

**Definition 2.1 (Traces refinement)** Let $P$, $Q$ be CSP processes. $P$ is a trace refinement of $Q$, written as $Q \sqsubseteq_T P$, if and only if, $\text{traces}(P) \subseteq \text{traces}(Q)$.

Two processes $P$ and $Q$ are traces-equivalent, $P \equiv_T Q$, if $P \sqsubseteq_T Q$ and $Q \sqsubseteq_T P$, i.e., $\text{traces}(P) = \text{traces}(Q)$. The process $\text{STOP}$ is the most refined process in the traces model, i.e., $P \sqsubseteq_T \text{STOP}$ for all processes $P$.

The traces model is the weakest of the three denotational models of CSP that we consider. In fact, the traces of internal and external choice are indistinguishable. This indicates that $\text{traces}(P)$ does not give a complete description of $P$, since we would like to be able to distinguish between $P \sqcap Q$ and $P \sqcap Q$. For example, the process $a \rightarrow \text{SKIP}$ guarantees that if the environment is prepared to engage in the event $a$ and then terminate, then it can engage in the event $a$ and terminate successfully. However, $a \rightarrow \text{SKIP} \sqcap a \rightarrow \text{STOP}$ does not guarantee that it can engage in the event $a$ and terminate successfully if the environment is ready to engage in the event $a$ and terminates. The traces model identifies both processes as they have the same traces. However, one of them guarantees that it will terminate successfully, but the other does not guarantee.

In terms of verification, the traces model can be deployed for the verification of safety conditions. That is, a process $Q$ which is a trace refinement of a process $P$, will perform traces already defined in $P$ and nothing more, i.e., $\text{traces}(Q) \subseteq \text{traces}(P)$. Safety conditions are concerned with the exclusion of traces only.

**Stable failures model.** The stable failures model $\mathcal{F}$ gives finer information about processes. For instance, it allows us to distinguish between internal and external choice (and much more). In particular, it allows us 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 $s$. The set $X$ is called the refusal set; the process cannot perform any event in the set $X$ no matter for how long it is offered.

The 'stable' in the model name means that the sequences represented by $s$ are those that reach a stable state where no transition is chosen nondeterministically. In other words, stable states are those in which there are no choices between external and internal actions.

As an example, let us consider the following processes over the alphabet \( \{a, b\} \):

\[
P = a \rightarrow \text{STOP} \n b \rightarrow \text{STOP}
\]

\[
Q = a \rightarrow \text{STOP} \n \top \n b \rightarrow \text{STOP}
\]

The stable failure set of $P$ and $Q$, denoted by $\text{failures}(P)$ and $\text{failures}(Q)$, are given by:

\[
\text{failures}(P) = \{ (\langle \rangle, \{\checkmark\}) \} \\
\cup \{ (\langle a \rangle, X) \mid X \subseteq \{a, b, \checkmark\} \} \\
\cup \{ (\langle b \rangle, X) \mid X \subseteq \{a, b, \checkmark\} \}
\]

\[
\text{failures}(Q) = \{ (\langle \rangle, X) \mid X \subseteq \{a, \checkmark\} \} \\
\cup \{ (\langle a \rangle, X) \mid X \subseteq \{a, b, \checkmark\} \} \\
\cup \{ (\langle b \rangle, X) \mid X \subseteq \{a, b, \checkmark\} \}
\]

Here, $P$ and $Q$ have different failures, i.e., the stable failures model $\mathcal{F}$ can distinguish between internal and external choice. The failures of $P$ records that initially (after the trace $s = \langle \rangle$) the process cannot refuse either $a$ or $b$. The process $Q$ has two initial invisible actions $\tau$ to choose from. After performing them, it reaches stable states, where it can perform either $a$ or $b$ separately, and refuse $b$ or $a$ respectively. The failure of $Q$ does not record any information about the initial state, but only information about the stable states.

Observe that it is by no means inevitable that every trace of a process has failure: it may never stop performing $\tau$ actions. So, as not all traces of a process are present in its failures, a process in the $\mathcal{F}$ model is represented not only by its stable failures, but also by its traces. Formally, in the stable failures model, each process $P$ is modelled by a pair $(T, F)$, denoting $T = \text{traces}(P)$ and $F = \text{failures}(P)$, where $T \subseteq \Sigma^\ast$ and $F \subseteq \Sigma^\ast \times \mathcal{P}(\Sigma^\ast)$, satisfying the following healthiness conditions (where $s, t$ range over $\Sigma^\ast$ and $X, Y$ over $\mathcal{P}(\Sigma^\ast)$):
T1. \( T \) is non-empty and prefix closed.

T2. \((s, X) \in F \Rightarrow s \in T\). This asserts that all traces performed by the failures should be recorded in the traces component \( T \). In other words it establishes consistency between the traces component and the failures component.

T3. \( s \upharpoonright \langle \checkmark \rangle \in T \Rightarrow (s \upharpoonright \langle \checkmark \rangle, X) \in F \). If a trace terminates successfully by producing \( \checkmark \), then it should refuse all events in \( \Sigma' \) at the stable state after \( s \upharpoonright \langle \checkmark \rangle \).

F2. \((s, X) \in F \land Y \subseteq X \Rightarrow (s, Y) \in F\). This asserts that in a stable state if a set \( X \) is refused, then any subset \( Y \) of \( X \) should also be refused.

F3. \((s, X) \in F \land \forall a : Y \bullet s \upharpoonright \langle a \rangle \not\in T \Rightarrow (s, X \cup Y) \in F\). This asserts that if a process \( P \) can refuse the set \( X \) of events in some stable state, then the same state must also refuse any set of events \( Y \) that the process can never reach.

F4. \( s \upharpoonright \langle \checkmark \rangle \in T \Rightarrow (s, \Sigma) \in F\). This asserts that if we have any terminating trace \( s \upharpoonright \langle \checkmark \rangle \in F \), these should refuse \( \Sigma \) at the stable state after \( s \).

For example, \( \text{STOP} \) initially refuses to communicate anything.

\[
\text{failures}(\text{STOP}) = \{(\langle \rangle, X) \mid X \subseteq \Sigma'\}
\]

Furthermore, initially the prefix process cannot refuse the prefixing event. Details about the other constructors are presented in [Ros98].

\[
\text{failures}(a \to P) = \{(\langle \rangle, X) \mid a \notin X\} \\
\cup \{(\langle a \rangle \upharpoonright s, X) \mid (s, X) \in \text{failures}(P)\}
\]

A process \( C \) is a stable failures refinement of \( A \) if, and only if, it contains all traces within \( A \) and presents less stable failures; it refuses less communications.

**Definition 2.2 (Stable failure refinement)** Let \( P, Q \) be CSP processes. \( P \) is a stable failure refinement of \( Q \), written as \( Q \sqsubseteq F P \), if, and only if: \( \text{traces}(P) \subseteq \text{traces}(Q) \land \text{failures}(P) \subseteq \text{failures}(Q) \).

In other words, if every trace \( s \) of \( Q \) is possible for \( P \) and every refusal after this trace is possible for \( P \), then \( Q \) can neither accept an event nor refuse unless \( P \) does. Two processes \( P \) and \( Q \) are stable failure-equivalent, \( P \equiv F Q \), if \( \sqsubseteq F Q \) and \( Q \sqsubseteq F Q \), i.e., \( \text{traces}(P) = \text{traces}(Q) \) and \( \text{failures}(P) = \text{failures}(Q) \). The bottom element in \( \sqsubseteq F \) is \( \Sigma^*, \Sigma^* \times P(\Sigma^*) \), while its top element is \( (\langle \rangle, \emptyset) \).
An important phenomenon captured by $F$ is deadlock. Deadlock is a phenomenon pertaining to networks of communicating processes which occur when two processes cannot agree to communicate with each other, thus the whole system becomes permanently frozen. This is potentially catastrophic in safety-critical computing applications. A network that can never exhibit deadlock is said to be deadlock-free.

In CSP deadlock is represented by the process $STOP$, which can perform only the empty trace, and after the empty trace the process $STOP$ refuses to engage in any event. In CSP, a process $P$ is considered to be deadlockfree, if the process $P$ after performing a trace $s$ never becomes equivalent to the process $STOP$.

**Definition 2.3 (Deadlock-free process)** A process $P$ is deadlock-free in CSP if, and only if, $\forall s : \Sigma^* \bullet (s, \Sigma^c) \notin \text{failures}(P)$

This definition is justified, as in the model $F$ the set of stable failures is required to be closed under the subset-relation ($F2$). In other words: Before termination, the process $P$ can never refuse all events; there is always some event that $P$ can perform. Moreover, the stable failure refinement notion preserves the deadlock-freedom of a process. That is, if $P$ is deadlock free and $P \sqsubseteq F Q$, then $Q$ is deadlock free.

From the definition of deadlock-free, an interesting lemma about deadlock-freedom in parallel synchronisations is described below.

**Lemma 2.1** Let $P$ and $Q$ be divergence-free CSP processes. Then $P \parallel Q$ deadlocks if, and only if:

$$\exists (t, X) : \text{failures}(P) \bullet (t, \Sigma \setminus X) \in \text{failures}(Q)$$

From the lemma above, it is possible to formulate an important observation about how process should communicate in order to preserve deadlock-freedom: one process can never refuses all events that the other can perform. For instance, consider that $X$ is a maximum refusal of $P$, then $P$ can perform events within $\Sigma^c \setminus X$. From the lemma above, in order to avoid deadlock, $Q$ cannot refuse such events.

**Failures/divergences Model.** The failures/divergence model gives us the most satisfactory representation for analysing liveness and safety properties of a CSP process; it allows us to detect not only deadlocked, but also live-locked processes. Furthermore, it has long been taken as the ‘standard’ model for CSP.
A process diverges, if it reaches a state from which it may forever compute internally through an infinite sequence of invisible actions. This is clearly a highly undesirable feature of the process, described by as ‘even worse than deadlock’ [Hoa85]. Livelock may invalidate certain analysis methodologies, and is often caused by a bug in the modelling. However the possibility of writing down a divergent process arises from the presence of two crucial constructs: hiding and ill-formed recursive processes. For instance, consider the processes $P = P$ and $Q = (a \rightarrow Q) \setminus \{a\}$. $Q$ converts the external event $a$ into an internal action $\tau$. 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 $\text{div}$ (the same of $Q$, in our example) represents the livelock phenomenon: immediately, 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. The divergences of a process are the finite traces on which the process can perform an infinite sequence of internal (invisible) actions. So, each process $P$ is modelled by the pair $(\text{failures}_{\perp}(P), \text{divergences}(P))$, where:

- $\text{divergences}(P)$ is the (extension-closed) set of traces $s$ on which a process can diverge. Thus, $\text{divergences}(P)$ contains not only the traces $s$ on which $P$ can diverge, but also all extensions $s \triangleleft t$ of such traces;

- $\text{failures}_{\perp}(P) = \text{failures}(P) \cup \{(s, X) \mid s \in \text{divergences}(P)\}$.

Formally the failures/divergences model $\mathcal{FD}$ is defined to be the pairs $(F_{\perp}, D)$ satisfying the following healthiness condition, where $s, t$ range over $\Sigma^* \models$, and $X, Y$ range over $\mathcal{P}(\Sigma^*)$:

**F.1.** $\text{traces}_{\perp}(P) = \text{traces}(P) \cup \text{divergences}(P)$ is non-empty and prefix closed.

**F.2.** $(s, X) \in F \land Y \subseteq X \Rightarrow (s, Y) \in F$.

**F.3.** $(s, X) \in F \land (\forall a \in Y \cdot s \triangleleft \langle a \rangle \notin \text{traces}_{\perp}(P)) \Rightarrow (s, X \cup Y) \in F$.

**F.4.** $s \triangleleft \langle \models \rangle \in \text{traces}_{\perp}(P) \Rightarrow (s, \Sigma) \in F$.

**D.1.** $s \in D \cap \Sigma^* \land t \in \Sigma^* \models \Rightarrow s \triangleleft t \in D$.

**D.2.** $s \in D \Rightarrow (s, X) \in F$. This adds all divergences-related failures of $F$.

**D.3.** $s \triangleleft \langle \models \rangle \in D \Rightarrow s \in D$. This ensures that we do not distinguish between how processes behave after successful termination.
A process $C$ is a failures/divergence refinement of $A$ if, and only if, it contains all failures and divergences of $A$: it refuses less communications and diverges in less occasions.

**Definition 2.4 (Failures/divergences refinement)** Let $P, Q$ be CSP processes. $P$ is a failures-divergences refinement of $Q$, written as $Q \preceq_{FD} P$, if and only if, $\text{failures}_\bot(P) \subseteq \text{failures}_\bot(Q) \land \text{divergences}(P) \subseteq \text{divergences}(Q)$.

Two processes $P$ and $Q$ are failures-divergences equivalent, $P \equiv_{FD} Q$, if $P \preceq_{FD} Q$ and $Q \preceq_{FD} P$, i.e., $\text{failures}_\bot(P) = \text{failures}_\bot(Q)$ and $\text{divergences}(P) = \text{divergences}(Q)$. The process $\text{div}$ is the least refined process in the failures/divergence model. Then, a process is said to be free of divergence (or livelock free) if after carrying out a sequence of events, its denotation is different from $\text{div}$.

It is consensual that the failures-divergences model gives us the most satisfactory representation for analysing liveness and safety properties of a CSP process. However, when we look into the mathematical theory of how divergences are calculated, it turns out that seeing accurately what a process can do after it has already been able to diverge is very difficult, and not really worth the effort [Ros98]. By combining traces with stable failures (which is in fact the failures part of the failures-divergences model), it is possible to see beyond any divergence by ignoring divergences altogether. Moreover, it is sometimes advantageous to analyse a divergence-free process $P$ by placing it in a context in which it may diverge as the result of hiding some set of actions; this only works when the traces and stable failures in this context are not influenced by these divergences.

For instance, the process $P = (a \rightarrow P \sqcap b \rightarrow P) \setminus \{b\}$ diverges in its initial state. The hiding operation converts the external choice into an internal choice. Therefore, the process internally chooses between the external event $a$ and an internal action resulted from hiding $b$. As a consequence, $P$ may indefinitely perform internal actions, which in the failures-divergences model leads to divergence.

As we will see in Section 2.3 in our formalisation of some notions, it is not convenient that certain hidden events result in divergence. For example, our intention is that the communication protocols of divergence-free components are also divergence-free processes, even after hiding all events not in the protocol interface.

Therefore, we assume in this work that basic components are divergence-free and deadlock-free, and use the semantic models presented here in verifications to ensure that such problems are not introduced in the system formed by
these components. The failures model is used in local analysis, in which the involved processes are divergent-free and the applied operators are known for not introducing such a problem. The failures/divergence model is used in verifications about the compositionality of strategy proposed here, checking theirs traces, failures and divergences.

2.2 CML

The COMPASS Modelling Language (CML) \cite{WCF12} is a formal specification language that integrates a state based notation (VDM++) and a process algebraic notation (CSP \cite{Hoa85}), as well as Dijkstra’s language of guarded commands and the refinement calculus. It supports the specification and analysis of state-rich distributed specifications. Additionally, CML supports step-wise development by means of algebraic refinement laws. The soundness of the refinement laws is established with respect to the formal semantics of CML, defined in Unifying Theories of Programming \cite{HJ98}. CML is still under development, with a COMPASS tool and several analysis plug-ins currently in production \cite{CML12}. In particular, tool support for CML will include a parser, a type-checker, a simulator, a theorem prover, a model-checker and a refinement editor.

In the remainder of this section, we introduce CML by means of a specification of a simple clock, and provide extensions to the clock example to illustrate features of the CML language. For more details on CML, refer to \cite{WCF12,WCC12}.

Initially, we specify a simple clock whose only observable behaviour is a synchronisation on a channel tick.

channels tick

Internally, the clock has a state variable \( s \) that records the number of seconds (marked by tick) that have elapsed, and has two operations defined: Init() and increment. The first simply initialises the state with 0, and the second adds one to the state component. The state is captured by the following class declaration.

```cml
class ClockSt =
begin
  state
    public s: nat
  initial
```
The frame keyword in the declaration of operations specifies the state components that can be read and written by the operation. In the case of the Init operation, the state component s can be written by Init. The post keyword specifies the post-condition of the operation. In the case of Init, the post-condition states that the state component s (after the operation) is equal to zero. The post-condition of the operation increment equates the state component s, after the operation, to the sum of its initial value (s\^\text{previous}) and one.

Our simple clock initialises its state, waits for one time unit, which we take to mean one second, increments its counter and synchronises on tick. This is specified by the following process declaration.

```
process SimpleClock =
begin
state
c: ClockSt
actions
  Ticking = Wait 1; c.increment(); tick -> Skip
  @ c.Init(); mu X @ Ticking; X
end
```

The simple clock is a process that declares a state and a number of actions. The state, in this case, is formed by a single state component c of type ClockSt. The actions include Ticking and the action started by @. The latter is a mandatory main action that defines the behaviour of the process; in this case, it simply initialises the state by calling the operation Init() of the state component c and recursively (mu) calls the action Ticking. This action waits for one time unit, increments the internal counter and synchronises on the channel tick.

Our initial specification of the clock is extremely simple, the only observable event is the synchronisation on tick. It might be interesting to have a clock that takes advantage of its internal counter and supplies information about
how many seconds, minutes, hours and days have elapsed.

We now extend our simple clock to include this additional functionality. First, we declare four additional channels that communicate a natural number. They are used to query the seconds, minutes, hours and days that have elapsed.

channels second, minute, hour, day: nat

The new clock specification is similar to the simple clock; it declares the state of the process as the component $c$ of type ClockSt, but additionally defines three functions: get_minute, get_hour and get_day. They take the number of seconds recorded in the state, and calculate, respectively, the equivalent number of minutes, hours and days.

process Clock =
begin
    state c: ClockSt
    functions
        get_minute(s: nat) m: nat
        post m = s/60

        get_hour(s: nat) h: nat
        post h = get_minute(s)/60

        get_day(s: nat) d: nat
        post d = get_hour(s)/24

The ticking action remains the same as before, but we add a new action, Interface, that provides the extra functionality.

actions
    Ticking = Wait 1; c.increment(); tick -> Skip
    Interface = second!(c.s) -> Interface
    [] minute!(get_minute(c.s)) -> Interface
    [] hour!(get_hour(c.s)) -> Interface
    [] day!(get_day(c.s)) -> Interface

This action simply offers a choice ([]) of communication over the channels second, minute, hour and day, and recurses. Each communication outputs (outputs are indicated by ! after a channel name) the appropriate value calculated using the functions previously defined.

Now, the main action of the new clock is slightly different. It first initialises the state as usual, but instead of offering Ticking alone, it composes Ticking...
in parallel with the recursive action Interface with the option of interrupting (\(\setminus\setminus\)) Interface with a synchronisation on tick. The parallel operator \([ \setminus \setminus \text{ns1} \setminus \setminus \text{cs} \setminus \setminus \text{ns2} \setminus \setminus \)] contains a set of events cs on which the two parallel actions synchronise, and two name sets ns1 and ns2 that partition the state of the process and indicate which state components can be updated by the left (ns1) and right (ns2) parallel actions. In our example, the action Ticking can update the state component c and the right parallel action does not update the state. The parallel actions synchronise on the channel tick.

The two parallel action synchronise on the channel tick.

\[
\text{\@ c.Init(); mu X @ (}
\text{Ticking}
\quad [\setminus \setminus \{c\} \setminus \setminus \{|tick|\} \setminus \setminus \{\} \setminus \setminus ]
\quad (\text{Interface/\setminus \setminus tick \setminus \setminus \rightarrow \setminus \setminus Skip})
\quad X
\text{\end)}
\]

While Ticking is waiting, the right hand side of the parallelism can offer any number of interactions over the channels in Interface. When Ticking finishes waiting, s is incremented, and the parallelism synchronises on tick. In this case, the action Interface is interrupted and both sides of the parallelism terminate. At this point, the recursive call (on X) takes place.

When the parallelism starts, both sides receive a copy of the state, and when the parallelism terminates, the state is updated based on the changes performed by the two sides (on their copies of the state) and the partition of the state. A consequence of this is that changes to the state performed by Ticking can only reflect in the behaviour of Interface when the parallelism terminates, the state is updated and Interface restarts (as part of the recursive call) with a copy of the updated state.

Now we have a clock that not only signals the passing of time, but can also output the time. However, we might also want to be able to restart the clock. For this, we define a channel restart and a new clock RestartableClock.

\[
\text{\begin{array}{c}
\text{channels}
\text{restart}
\text{\end{array}}}
\]

We define the restartable clock similarly to the process Clock defined above. The restartable clock process RestartableClock has a new action Cycle, and the altered main action offers the action Cycle and the possibility of interrupting it through the channel restart. If the interruption takes place,
the main action recurses and **Cycle** is called resetting the state.

**process RestartableClock =**

**begin**

**state c: ClockSt**

**functions**

get_minute(s: nat) m: nat

post m = s/60

get_hour(s: nat) h: nat

post h = get_minute(s)/60

get_day(s: nat) d: nat

post d = get_hour(s)/24

**actions**

Ticking = Wait 1; c.increment(); tick -> Skip

Interface = second!(c.s) -> Interface

[] minute!(get_minute(c.s)) -> Interface

[] hour!(get_hour(c.s)) -> Interface

[] day!(get_day(c.s)) -> Interface

**Cycle = c.Init(); mu X @ (**

Ticking

[] c | {} | tick | {} |

(Interface/\tick -> Skip)

)); X

@ mu X @ Cycle /\ restart -> X

**end**

We can further extend the functionality of the clock by specifying a multi-clock. A simple way of defining such a clock is to compose a number of restartable clocks (or any other variety of clock). This raises the question of how the clocks are composed. For instance, do all clocks synchronise on **tick**? Can they be restarted on a one by one basis? We present below two processes that model a multi-clock. Both of them assume that the clocks are synchronous, but the first allows independent restarting, while the second does not.

First, we define a number of channels that allow the environment to communicate with specific clocks. We assume that the clocks in the multi-clock are numbered by natural numbers, and are declared in an equivalent way to the ones already defined (except for **tick**), communicating a natural number
(the identifier of the clock) and the value originally communicated. We prefix the name of the channels with an i.

channels
  isecond, iminute, ihour, iday: nat * nat
  irestart: nat

Our first model of a multi-clock is specified by the process \texttt{NRestartableClocks1}. This is a parameterised process that takes the number \( n \) of clocks, and starts \( n \) copies of \texttt{RestartableClock} running in parallel and synchronising on \texttt{tick}. The channels in the \texttt{RestartableClock} process need to be renamed, otherwise we would not be able to distinguish one clock from another. We rename each channel (except \texttt{tick}) to its \texttt{i} version, communicating the identifier of the clock.

\begin{verbatim}
process NRestartableClocks1 = n: nat @
  [i{\{tick\}}] i: \{1,...,n\} @
  RestartableClock[[second <- isecond.i, minute <- iminute.i, hour <- ihour.i, day <- iday.i, restart <- irestart.i]]
\end{verbatim}

Our alternative process \texttt{NRestartableClocks2} is similar, except that the different clocks synchronise on \texttt{restart} as well, and this channel is not renamed. Thus, a synchronisation on \texttt{restart} restarts all the clocks simultaneously.

\begin{verbatim}
process NRestartableClocks2 = n: nat @
  [i{\{tick, restart\}}] i: \{1,...,n\} @
  RestartableClock[[second <- isecond.i, minute <- iminute.i, hour <- ihour.i, day <- iday.i]]
\end{verbatim}

One might consider that, whilst these definitions are reasonably intuitive, they are not the most efficient for implementation purposes. So, one might implement a multi-clock simply by associating each channel of a restartable clock with the equivalent \texttt{i} channel, but ranging over all the possible clocks. The next process models such an solution.

\begin{verbatim}
process NRestartableClocksImpl = n: nat @
  RestartableClock[[second <- isecond.i, minute <- iminute.i, 
\end{verbatim}
This process simply renames each channel of RestartableClock (except tick and restart) to a set of communications on the associated i channel communicating the identifiers of the clocks.

This process raises the question of which of our multi-clock processes is being implemented by NRestartableClocksImpl. This questions can be formulated as follows:

assert NRestartableClocks1 [= NRestartableClocksImpl
assert NRestartableClocks2 [= NRestartableClocksImpl

The first assertion states that NRestartableClocksImpl is a refinement of NRestartableClocks1, and the second asserts that the implementation is a refinement of NRestartableClocks2. For some models, this assertions can be checked using a model-checker, but for other, a theorem-prover may be necessary. The CML tools will help answer such questions.

2.3 **BRICK**: Systematic Development of Trustworthy Component Systems

In this section, we discuss the theoretical background of the report. We present the basic definitions and the composition rules in Section 2.3.1. The extended counterparts of the definitions and the composition rules are presented in Section 2.3.2. A full account on the theoretical background can be found in our previous deliverable Deliverable D24.1 [OSA13].

2.3.1 Component Model

Our approach is based on a component model that delimits the broad outline of what constitutes a component, exposing its necessary related technical concepts and constraints. Both, components and connectors, as well as their interaction semantics, are characterised in this component model that defines the building blocks of our systematic development approach. A component

\[\text{assert} \ P \ [= \ Q \text{ asks the model checker if the process P is refined by the process Q.} \]
contract\textsuperscript{3}, whose definition is presented below, encapsulates a component in our approach. They are defined in terms of their behaviour (represented as a CML process), ports (represented as channels) and respective interfaces (types).

**Definition 2.5 (Component contract)** A component contract $Ctr$ comprises an observational behaviour $B$, a set of communication channels $C$, a set of interfaces $I$, and a total function $R : C \rightarrow I$ between channels and interfaces of the contract:

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

such that $B$ is an I/O process (a CML process that satisfy the conditions described in Definition 2.6).

Intuitively, the component $R$ describes the component’s channels and their respective types.

Our approach follows approaches like that of [All97] in which component models have a higher-level granularity by complementing the syntactical information of a component with behaviour. In our case, we explicitly separated inputs and outputs.

The behaviour of these components are represented by I/O processes.

**Definition 2.6 (I/O process)** We say $P$ is a CML I/O process if it satisfies the following five conditions, which are formally presented in [Ram11]:

- **I/O Channels.** Every channel in $P$ is either an input channel or an output channel. Formally, we say a channel $c$ is an I/O channel if there exists two functions, inputs($c$, $P$) and outputs($c$, $P$), for every process $P$, such that:
  - $\text{inputs}(c, P) \cup \text{outputs}(c, P) \subseteq \{c\}$, and
  - $\text{inputs}(c, P) \cap \text{outputs}(c, P) = \emptyset$.
  
  Formally, the two functions map pairs CHANNEL $\times$ PROCESS to a set of events.

- **Infinite Traces.** $P$ has an infinite set of traces (but finite state-space);

- **Divergence-freedom.** $P$ is divergence-free;

\textsuperscript{3}In the COMPASS project contracts are described in CML. Here, a contract is a tuple that includes a behavioural specification (described in CML), and other elements that describe the ports and their types.
• **Input Determinism.** If a set of input events in \( P \) are offered to the environment, none of them are refused. Formally, we say a process \( P \) is input deterministic if:

\[
\forall s \uparrow (c.a) : \text{traces}(P) \mid c.a \in \text{inputs}(c, P) \bullet \\
(s, \{c.a\}) \notin \text{failures}(P)
\]

• **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. Formally, we say a process is strong output decisive if:

\[
\forall s \uparrow (c.b) : \text{traces}(P) \mid c.b \in \text{outputs}(c, P) \bullet \\
(s, \text{outputs}(c, P)) \notin \text{failures}(P) \\
\land (s, \text{outputs}(c, P) \setminus \{c.b\}) \in \text{failures}(P)
\]

These conditions lay the foundations of our composition rules for contracts whenever every two components are compatible to interoperate. The application of the composition rules and the characterisation constraints in the component model impose side conditions that, if satisfied, ensure deadlock freedom in the composition result. Hence, in our approach, problems are anticipated before all parts are integrated.

In [Ram11], we present four composition rules; each one focuses on a specific scenario at composition. The rules provide asynchronous pairwise compositions and focus on the preservation of deadlock freedom in the resulting component. The preservation of livelock-freedom is not in the scope of this report but also discussed in [Ram11]. Using the rules, developers may synchronise two channels of two components, or even of the same component. The four rules are interleave, communication, feedback and reflexive compositions. The first three rules have also been presented in [RSM09].

The interleave composition rule is the simplest form of composition. It aggregates two independent entities such that, after composition, these entities still do not communicate between themselves. They directly communicate with the environment as before, with no interference from each other. The only proviso states that they do not share any communication channel.

**Definition 2.7 (Interleave composition)** Let \( P \) and \( Q \) be two component contracts, such that:

- \( P \) and \( Q \) have disjoint channels, and;
- \( C_P \cap C_Q = \emptyset \).

32
Then, the interleave composition of $P$ and $Q$ (namely $P[|||]Q$) is given by:

$$P[|||]Q = P \langle \rangle \times \langle \rangle Q$$

This definition and others that follow use the direct composition operator $\times$, which provides an asynchronous interaction, mediated by infinite buffers, between corresponding channels from two lists. In this rule, no channel participates in the operation.

The result of an application of a composition rule is a new component. In many cases, it is necessary to provide a means to connect to two channels of a same component. This, however, is not possible using CML as it does not provide any constructor for a reflexive direct connection. For that, we use a buffered communication as means to permit such reflexive communication in a component. By providing an asynchronous interaction, we also offer a more generic approach that allows its use in both asynchronous and synchronous systems. On the other hand, the costs of verification are knowingly higher for buffered asynchronous specifications. This cost, however, is alleviated by the use of metadata as we discuss in Section 2.3.2.

The next composition rule needs the properties below.

Prop. i. (I/O Confluence) Whenever a state has two alternative actions $\alpha$ and $\beta$, then performing either of them does not preclude the other, unless it is a choice among inputs or outputs of the same channel;

Prop. ii. (Finite Output Property) They always communicate a finite number of outputs. As I/O processes are divergence-free, the absence of divergence after hiding the outputs in the original protocol guarantees this property.

Prop. iii. (Strong Compatibility) There must always be an output event to be performed, and at least one of the processes must have all enabled outputs accepted by the other process.

The first two properties deal with buffering concerns in order to allow mechanical verifications on the system without state explosion [Ros05]. The third property guarantees the interoperability of the two components. Here, the formal definitions are omitted for the sake of conciseness. They can be found in [Ram11].

The second composition rule states the most common way for linking complementary channels of two different entities.
Definition 2.8 (Communication composition) Let $P$ and $Q$ be two component contracts, and $ic$ and $oc$ two communication channels, such that:

- $ic \in C_P \land oc \in C_Q$;
- $C_P \cap C_Q = \emptyset$, and;
- the port protocols $\text{Prot}_{\text{IMP}}(P, ic) \parallel R_{io}^{ic \rightarrow oc}$ and $\text{Prot}_{\text{IMP}}(Q, oc) \parallel R_{io}^{oc \rightarrow ic}$ are I/O confluent strong compatible and satisfy the finite output property.

Then, the communication composition of $P$ and $Q$ (namely $P[ic \leftrightarrow oc]Q$) via $ic$ and $oc$ is defined as follows:

$$P[ic \leftrightarrow oc]Q = P_{(ic)} \bowtie (oc)Q$$

Besides having disjoint channel sets, further restrictions apply to the divergent-free processes implementation protocols on the linked channels ($\text{Prot}_{\text{IMP}}$), which are given by the abstraction of their behaviour projection over these channels.

Definition 2.9 (Protocol implementation) Let $P$ be an I/O process, and $ch$ a communication channel. The communication protocol $\text{Prot}_{\text{IMP}}(P, ch)$ implemented by $P$ over $ch$ is a protocol that satisfies the following property:

$$\text{Prot}_{\text{IMP}}(P, ch) \equiv \mathcal{F} P \upharpoonright ch$$

where,

Definition 2.10 (Projection) Let $P$ be an I/O Process, and $C$ a set of communication channels. The projection of $P$ over $C$ (denoted by $P \upharpoonright C$) satisfies the following properties:

1. $P \upharpoonright C$ is an I/O Process
2. $\forall c : C \bullet \text{inputs}(P \upharpoonright C, c) \subseteq \text{inputs}(P, c)$
3. $\forall c : C \bullet \text{outputs}(P \upharpoonright C, c) \subseteq \text{outputs}(P, c)$
4. $\alpha(P \upharpoonright C) \subseteq \bigcup_{c : C} \{c\}$
5. $P \equiv T P \parallel \sum \parallel (P \upharpoonright C) \parallel \text{RUN}(\text{NOT}(C))$

Properties 1 - 3 guarantees that the communication direction (input and output) are not changed and that the properties of strong output decisiveness and input determinism are maintained. This ensures that we are neither removing nor introducing non-determinism. Property four ensures that the
projection process refers only to channels in $C$. Finally, together with the previous properties, property 5 guarantees that the process behaviour on the projected channels is not changed.

The restrictions on the prot-protocols, however, apply to a renamed version of these protocols: $[R]^{oc\rightarrow ic}_{IO}$ replaces outputs of $oc$ by inputs of $ic$.

$$P \parallel R^{oc\rightarrow ic}_{IO} = P\{a.x \mapsto b.x \mid a.x \in \text{outputs}(P)\}$$

This corresponds to the original intention (i.e. replaces outputs of $a$ by inputs of $b$) but uses the function outputs rather than the previously enforced channel structure.

There are also consequences to the definition of inputs and outputs of a renamed protocol.

$$\text{inputs}(P \parallel R^{oc\rightarrow ic}_{IO}) = \text{inputs}(P)$$

$$\text{outputs}(P \parallel R^{oc\rightarrow ic}_{IO}) = \text{outputs}(P)[a \mapsto b]$$

where $[a \mapsto b]$ replaces all references events on $a$ to events on $b$ in a given set of events.

$$S[a \mapsto b] = S \setminus \{a.x \mid a.x \in S\} \cup \{b.x \mid a.x \in S\}$$

Practical developments also present more complex systems with cycles of dependencies in the topology of the system structure; undesirable cycles need to be avoided. The feedback composition provides the possibility of creating safe cycles.

**Definition 2.11 (Feedback composition)** Let $P$ be a component contract, and $ic$ and $oc$ two communication channels, such that:

- the protocols $\text{Prot}_{IMP}(P, ic) \parallel R^{oc\rightarrow ic}_{IO}$ and $\text{Prot}_{IMP}(P, oc) \parallel R^{ic\rightarrow oc}_{IO}$ are I/O confluent strong compatible and satisfy the finite output property, and;

- $\{ic, oc\} \subseteq CP$ and decoupled in $P$.

Then, the feedback composition $P (P[oc \leftrightarrow ic])$ hooking $oc$ to $ic$ is defined as follows:

$$P[oc \leftrightarrow ic] = P \triangleright_{\{ic\}} \triangleright_{\{oc\}}$$
This rule imposes some conditions that are similar to those in the communication composition rule (relative to protocol compatibility and buffer tolerance), except that it additionally imposes that channels are decoupled.

Prop. iv. (Decoupled Channels) Communication on one channel does not interfere on communications through the other (their communications are interleaved). Formally, the channels within Ch are decoupled in P if, and only, if $P \upharpoonright \text{Ch} \equiv_{\mathcal{F}} \prod_{z \in \text{Ch}} \text{Prot}_{\text{IMP}}(P, z)$.

The composition rules presented so far deal with systems with a tree topology. In practice, there are more complex systems that indeed present cycles of dependencies in the topology of the system structure. The last composition rule, reflexive composition, is more general than the feedback one. However, it is also more costly regarding verification.

Definition 2.12 (Reflexive composition) Let P be a component contract, and ic and oc two communication channels, such that:

- $\{ic, oc\} \subseteq C_P$, and;
- $P \upharpoonright \{ic, oc\}$ is buffering self-injection compatible and satisfies the finite output property.

Then, the reflexive composition $P$ (namely $P[oc \leftrightarrow ic]$) hooking oc to ic is defined as follows:

\[ P[ic \leftrightarrow oc] = P \upharpoonright_{\{oc\}}^{\{ic\}} \]

This rule requires that the projection on the two linked channels ($P \upharpoonright \{ic, oc\}$) satisfies the finite output property and the projection is buffering self-injection compatible.

Prop. v. (Buffering Self-injection Compatibility) allows the injection of information from one channel to the other via the implicit buffers of the composition. Formally, a buffering self-injection compatible process can establish a communication between its channels via a one-place buffer without deadlock.

From our proposed building block constructors (composition rules), any system $S$ can be structured as follows.

\[ S ::= P \mid S[\_\_\_ \leftrightarrow \_\_] S \mid S[c_1 \leftrightarrow c_2] S \mid S[c_1 \leftarrow c_2] S \mid S[c_1 \leftarrow c_2] \mid S[c_1 \leftarrow c_2] \mid S[c_1 \leftarrow c_2] \mid S[c_1 \leftarrow c_2] \]

where $P$ is a component contract whose behaviour is deadlock free. We say that any component system that follows this grammar is in normal form.
The following theorem from [Ram11] guarantees that components arising from the application of the rules to deadlock-free components are also deadlock-free.

**Theorem 2.1 (Deadlock-free Component Systems)** Any system $S$ in normal form, built from deadlock-free components, is deadlock-free.

In addition to the contract elements previously presented, we may also define an enriched component contract (*BRICK-components*). These components enrich the original contracts with metadata that record by construction information that can be used to alleviate some verification conditions during component composition. This enriched components contract and the corresponding composition rules are presented in the next section.

### 2.3.2 Extended Component Model

In our approach, metadata comprise information that can (at any moment) be derived from other component contract elements. Such metadata enriches component contracts with static information that assists the runtime environment with additional (validation) properties. The metadata information is: (1) dual protocols; (2) context protocols; (3) protocol implementations; and (4) decoupled channels. Informally, the behaviour of the dual protocol of a process $P$ after a trace $s$ is always an external choice of the outputs and one of the inputs of $P$, if it exists, after $s$. Furthermore, a context protocol of a process $P$ is a deadlock-free deterministic process that has the same traces as $P$. Both are used in protocol compatibility verifications. The main metadata information selected in our approach are decoupled channels and protocol implementations. These are important conditions in the communication and feedback compositions rules. Similarly to the composition rules presented before, we presented four composition rules for enriched component contracts. In particular, we use metadata to alleviate several verifications in our rigorous strategy for component compositions. The extended contracts specialise the notion of protocol oriented component and enrich their contract with metadata.

**Definition 2.13 (Enriched component contract)** Let $Ctr$ be a protocol oriented component contract, and $K$ a metadata derived from its elements. An enriched component contract that includes $Ctr$ is represented by:

$$\langle B_{Ctr}, R_{Ctr}, I_{Ctr}, C_{Ctr}, K \rangle$$
where $\mathcal{K}$ comprises the following information:

$$\mathcal{K} : \langle \text{Prot}^\mathcal{K}, \text{CTX}^\mathcal{K}, \text{DProt}^\mathcal{K}, \text{Dec}^\mathcal{K} \rangle$$

such that:

- $\text{dom } \text{Prot}^\mathcal{K} \subseteq \mathcal{C}_{\text{Ctr}} \land \forall c : \text{dom } \text{Prot}^\mathcal{K} \bullet \text{Prot}^\mathcal{K}(c) \sqsubseteq_{F} \text{Prot}_{\text{IMP}}(\mathcal{C}_{\text{Ctr}}, c)$

- $\text{dom } \text{DProt}^\mathcal{K} \subseteq \mathcal{C}_{\text{Ctr}} \land \forall c : \text{dom } \text{DProt}^\mathcal{K} \bullet \text{DProt}^\mathcal{K}(c)$ is the dual protocol of $\text{Prot}^\mathcal{K}(c)$

- $\text{dom } \text{CTX}^\mathcal{K} \subseteq \mathcal{C}_{\text{Ctr}} \land \forall c : \text{dom } \text{CTX}^\mathcal{K} \bullet \text{CTX}^\mathcal{K}(c)$ is the context process of $\text{Prot}^\mathcal{K}(c)$

- $\text{dom } \text{Dec}^\mathcal{K} \subseteq \mathcal{C}_{\text{Ctr}} \land \text{ran } \text{Dec}^\mathcal{K} \subseteq \mathcal{C}_{\text{Ctr}}$

- $\forall c_1, c_2 : \mathcal{C}_{\text{Ctr}} \bullet c_1 \text{ Dec}^\mathcal{K} c_2 \Rightarrow \{c_1, c_2\} \text{ DecoupledIn } \mathcal{C}_{\text{Ctr}} \land c_2 \text{ Dec}^\mathcal{K} c_1$

The element $\text{Prot}^\mathcal{K}$ is a relation from channels to protocols, which represent the actual port-protocol of the component on that channel. If a protocol within $\text{Prot}^\mathcal{K}$ satisfies a property, then, by refinement, it also holds for the protocol of the component. Similarly, the elements $\text{DProt}^\mathcal{K}$ and $\text{CTX}^\mathcal{K}$ map channels into context processes and dual protocols, respectively. They are used to support the use of the protocols within $\text{Prot}^\mathcal{K}$; these are used, for instance, in protocol compatibility verifications. Finally the element $\text{Dec}^\mathcal{K}$ is a relation among decoupled channels of the component.

Since these metadata comprise derived information, it can be ignored by a composition environment, and, furthermore, the component can still be used in environments unaware of them. As a consequence, despite the use of metadata can be considered a powerful tool during the integration phase, its use is optional.

To increase the value of our compositional approach, we derive composition metadata from the metadata of the original components, without always building them from scratch. After each composition rule is applied, the metadata are updated using simple formulae that consider the semantics of such composition rule.

Similarly to the composition rules presented before, we present four composition rules for enriched component contracts. In order to preserve protocols behaviours after each composition and to store them in metadata, enriched components require a stronger verification of protocol compatibility, which we call matching compatible.
Similarly to the rules presented before, we present four new composition rules for enriched component contracts. In order to preserve protocol behaviours after each composition and to store them in metadata, the new rules require a stronger notion of protocol compatibility, which we call matching compatibility.

Prop. vi. (Matching Compatibility) Two protocols $R$ and $S$ are compatible if the dual protocol of $R$ is failure equivalent to $S$. Formally, two port-protocols $P$ and $Q$ are matching compatible if, and only if, $DProt(P) \equiv_F Q$.

This kind of compatibility is subtly different from strong compatibility. The former is even stronger than the latter. The advantage of compositions in which the protocols are matching compatible is that it preserves local progress and, furthermore, other protocols (not involved in the composition) are preserved.

The simplest composition of enriched component contracts is the one formed by the interleaving of its components.

Definition 2.14 (Enriched interleaving composition) Let $P$ and $Q$ be two enriched component contracts, such that $P$ and $Q$ have disjoint channels, $C_P \cap C_Q = \emptyset$. Then, the enriched interleaving composition of $P$ and $Q$ (namely $P [\|\|] Q$) is given by:

$$P [\|\|] Q = Enrich(\langle B_P, R_P, I_P, C_P \rangle_0 \bowtie_0 \langle B_Q, R_Q, I_Q, C_Q \rangle, \langle Prot^K_{PQ}, CTX^K_{PQ}, DProt^K_{PQ}, Dec^K_{PQ} \rangle)$$

where

(i) $Prot^K_{PQ} = Prot^K_P \cup Prot^K_Q$

(ii) $CTX^K_{PQ} = CTX^K_P \cup CTX^K_Q(c)$

(iii) $DProt^K_{PQ} = DProt^K_P \cup DProt^K_Q$

(iv) $Dec^K_{PQ} = Dec^K_P \cup Dec^K_Q \cup \{(c_1, c_2) \mid (c_1 \in C_Q \wedge c_2 \in C_P) \vee (c_1 \in C_P \wedge c_2 \in C_Q)\}$

The result of this composition is similar to the one from Definition 2.7. In addition, we show here the metadata associated to the interleaving. At this moment, no benefit is obtained from the metadata; they are maintained for more complex compositions. However, the calculation of metadata is very simple. It basically includes all information of the metadata of $P$ and $Q$. 

39
except that it also states that all channels of one component are decoupled from the other; this is a direct result of the interleaved behaviour of the composition.

Similarly, we define communication compositions of enriched component contracts in the following way.

**Definition 2.15 (Enriched communication composition)** Let $P$ and $Q$ be two enriched component contracts, and $ic$ and $oc$ two channels, such that:

- $ic \in C_P \land oc \in C_Q$;
- $C_P \cap C_Q = \emptyset$, and;
- the port protocols $Prot^K_P(ic) \parallel R_{IO}^\rightarrow oc$ and $Prot^K_Q(oc) \parallel R_{IO}^\rightarrow ic$ are I/O confluent matching compatible and satisfies the finite output property.

Then, the communication composition $P[ic \leftrightarrow oc]Q$ is defined as follows:

$$P[ic \leftrightarrow oc]Q = \text{Enrich}\left(\left\langle B_P, R_P, I_P, C_P \right\rangle_{(ic)} \bowtie_{(oc)} \left\langle B_Q, R_Q, I_Q, C_Q \right\rangle, \left\langle Prot^K_{PQ}, CTX^K_{PQ}, Dec^K_{PQ} \right\rangle\right)$$

where

$$Prot^K_{PQ} = \{(c, Prot^K_P(c)) \mid c \in \text{dom } Prot^K_P \setminus \{ic\}\} \cup \{(c, Prot^K_Q(c)) \mid c \in \text{dom } Prot^K_Q \setminus \{oc\}\}$$

$$DProt^K_{PQ} = \{(c, DProt^K_P(c)) \mid c \in \text{dom } DProt^K_P \setminus \{ic\}\} \cup \{(c, DProt^K_Q(c)) \mid c \in \text{dom } DProt^K_Q \setminus \{oc\}\}$$

$$CTX^K_{PQ} = \{(c, CTX^K_P(c)) \mid c \in \text{dom } CTX^K_P \setminus \{ic\}\} \cup \{(c, CTX^K_Q(c)) \mid c \in \text{dom } CTX^K_Q \setminus \{oc\}\}$$

$$Dec^K_{PQ} = \left\{\begin{array}{l}
\left\{(c_1, c_2) \mid \{c_1, c_2\} \cap \{ic, oc\} = \emptyset \right. \\
\wedge \left. \left(\begin{array}{l}
\left(c_1 \ Dec^K_P ic \lor ic \ Dec^K_P c_1\right) \\
\wedge \left(c_2 \in C_Q \lor c_1 \ Dec^K_P c_2\right) \\
\lor \left(oc \ Dec^K_Q c_2 \lor c_2 \ Dec^K_Q oc\right) \\
\wedge \left(c_1 \in C_P \lor c_1 \ Dec^K_Q c_2\right)
\end{array}\right) \right\}
\end{array}\right\}$$

The result of this composition is similar to the one from Definition 2.8, except for: instead of checking compatibility among port protocols of the original components, we check it on port protocols within their metadata. Furthermore, the composition does not have to take into account the complexity of its components, since no port-protocol has to be derived from the component.
behaviours. In addition, we show here the metadata associated to the composition, which can be used in further compositions. Again, the calculation of metadata is very simple. They include all information of the metadata of $P$ and $Q$, excluding information about $ic$ and $oc$, which does not belong to the new composition contract. There are also new relations identified among channels of one component and channels of the other, requiring that these channels are decoupled with the channels involved in the composition ($ic$ and $oc$). This results from the semantics of the parallel operator being used in the composition. Observe that $Dec^K$ is a symmetric relation, and, furthermore, this has to be handled in its calculation.

Now we define the feedback composition of an enriched component contract.

**Definition 2.16 (Enriched feedback composition)** Let $P$ be an enriched component contract, and $ic$ and $oc$ two communication channels, such that:

- $\{ic, oc\} \subseteq C_P$;
- the port protocols $Prot^K_P(ic)$ and $Prot^K_P(oc)$ are I/O confluent matching compatible and satisfies the finite output property, and;
- $ic Dec^K_P oc$.

Then, the feedback composition $P$ (namely $P[oc \leftarrow ic]$) hooking $oc$ to $ic$ is defined as follows:

$$P[oc \leftarrow ic]^e = \text{Enrich}((\langle B_P, R_P, I_P, C_P \rangle \times|^{\langle ic, oc\rangle},$$

$$\langle Prot^K_S, CTX^K_S, DProt^K_S, Dec^K_S \rangle)$$

where

$$Prot^K_{PQ} = \{(c, Prot^K_P(c)) | c \in \text{dom} Prot^K_P \setminus \{ic, oc\}\}$$
$$DProt^K_{PQ} = \{(c, DProt^K_P(c)) | c \in \text{dom} DProt^K_P \setminus \{ic, oc\}\}$$
$$CTX^K_{PQ} = \{(c, CTX^K_P(c)) | c \in \text{dom} CTX^K_P \setminus \{ic, oc\}\}$$
$$Dec^K_{PQ} = \left\{(c_1, c_2) \mid \{c_1, c_2\} \cap \{ic, oc\} = \emptyset \wedge c_1 Dec^K_P c_2 \wedge \left( (c_1 Dec^K_P ic \wedge c_1 Dec^K_P oc) \vee (ic Dec^K_P c_2 \wedge oc Dec^K_P c_2) \right) \right\}$$

The result of this composition is similar to the one from Definition 2.11, except that most provisos use the metadata of its original components directly. Instead of having to check compatibility among port protocols of $P$, there are new relations identified among channels of one component and channels of the other, requiring that these channels are decoupled with the channels involved in the composition ($ic$ and $oc$). This results from the semantics of the parallel operator being used in the composition. Observe that $Dec^K$ is a symmetric relation, and, furthermore, this has to be handled in its calculation.
we check this on port protocols within the metadata. Instead of verifying that two channels are decoupled in \( P \), we verify it directly on relations within the metadata. In this way, we perform lightweight verifications. Moreover, the composition does not have to take into account the complexity of \( P \). In addition, we show here the metadata associated to the composition, which can be used in further compositions. Again, the calculation of metadata is very simple. The new metadata include all information of the metadata of \( P \), excluding information about \( ic \) and \( oc \), which does not belong to the composition contract. Some other channels are also removed from the decoupled relation \( Dec^K_S \), since after the composition new communications are established.

The last rule is the reflexive composition of enriched compositions.

**Definition 2.17 (Enriched reflexive composition)** Let \( P \) be a component contract, and \( ic \) and \( oc \) two communication channels, such that:

- \( \{ ic, oc \} \subseteq C_P \), and;
- \( P \upharpoonright \{ ic, oc \} \) is buffering self-injection compatible and satisfies the finite output property.

Then, the reflexive composition \( P \) (namely \( P[oc \Rightarrow ic] \)) hooking \( oc \) to \( ic \) is defined as follows:

\[
P[ic \leftrightarrow oc]^e = \text{Enrich}(\langle B_P, R_P, I_P, C_P \rangle \rhd (ic), \langle Prot^K_S, CTX^K_S, DProt^K_S, Dec^K_S \rangle)
\]

where

\[
\begin{align*}
\text{Prot}^K_{PQ} &= \{(c, \text{Prot}^K_P(c)) \mid c \in \text{dom Prot}^K_P \setminus \{ ic, oc \}\} \\
\text{DProt}^K_{PQ} &= \{(c, \text{DProt}^K_P(c)) \mid c \in \text{dom DProt}^K_P \setminus \{ ic, oc \}\} \\
\text{CTX}^K_{PQ} &= \{(c, \text{CTX}^K_P(c)) \mid c \in \text{dom CTX}^K_P \setminus \{ ic, oc \}\} \\
\text{Dec}^K_{PQ} &= \left\{ (c_1, c_2) \mid \{ c_1, c_2 \} \cap \{ ic, oc \} = \emptyset \right. \\
&\quad \wedge \ c_1 \ Dec^K_P c_2 \wedge \\
&\quad \left. \left( (c_1 \ Dec^K_P ic \wedge c_1 \ Dec^K_P oc) \vee (ic \ Dec^K_P c_2 \wedge oc \ Dec^K_P c_2) \right) \right\}
\end{align*}
\]

The result of this composition is similar to the one from Definition 2.12. It does not benefit from the metadata of its original components. This is because to check buffering self-injection compatibility we cannot solely use port
protocols, but the entire component behaviour; it checks the behaviour concerning two communication channels. In addition, we show here the metadata associated to the composition, which can be used in further compositions. The structure of the metadata is identical to the one of a feedback composition of enriched components, since both are unary compositions.

In [Ram11], we provide proofs that guarantee that the result of the application of the extended composition rules are themselves extended component contracts. Observe that all rules presented here also guarantee deadlock freedom because the behaviour of their compositions is equivalent to the behaviour of the general rules used to create them.
3 Case Studies

In this section, we describe the two case studies that were developed for this deliverable. First, in Section 3.1, we informally describe and formalise a classical concurrency problem, the dining philosophers. This case study, although rather simple, perfectly illustrates the problems we are trying to solve in this deliverable. In Section 3.2, we informally describe and formalise a more complex and SoS related case study, a version of the leadership election algorithm developed in partnership with B&O [AOS+14a].

3.1 Dining Philosophers

This case study is the asymmetric dining philosophers. In this case study, philosophers try to acquire a pair of shared forks in order to eat. The philosophers are sat in a table and there is a fork between each pair of philosophers. The random acquisition of the forks by the philosophers might lead to a deadlock, but in the asymmetric version, the order of acquisition of forks prevents deadlocks. In this section, we introduce a CML model for the forks and philosophers, and we demonstrate how to use \texttt{BRIC} to compose these components so as to create the asymmetric dining philosophers settings.

Firstly, we introduce some global variable, or, as the CML terminology defines, values, that are useful for parametrising our model. The value $N$ gives the number of philosophers, or forks, in our model. Hence, if $N = 3$ then we have three forks and three philosophers in our model. The value \texttt{RANGE\_SET} gives a set of natural numbers that distinguishes the philosophers one from the other. The same set is used for distinguishing forks.

\begin{verbatim}
values
  N : nat = 3
  RANGE\_SET : set of nat = \{0,...,(N-1)\}
\end{verbatim}

Next, we introduce the types used in our model. The \texttt{RANGE} type consists of the set of natural numbers such that its elements belong to the \texttt{RANGE\_SET} value. The enumerated type \texttt{REQUISITION} gives the possible uses of a communication channel. The \texttt{<req>} states that an action is being requested by the channel, whereas the \texttt{<ack>} is used to acknowledge that an action has been performed. The enumerated type \texttt{S\_ACTION} (S stands for Shared) gives the actions performed by forks and philosophers, which require interaction between them. The enumerated type \texttt{I\_ACTION} (I stands for Individual)
gives the actions performed by a philosopher that do not require interaction, i.e. individual tasks performed by the philosopher.

types
RANGE = nat
   inv n == n in set RANGE_SET
REQUISITION = <req> | <ack>
S_ACTION = <picksup> | <putsdown>
I_ACTION = <eats> | <getsup> | <sitsdown>

Moving forward, we introduce the channels used in our model. The channel pfk is used by the philosophers to perform interacting actions with forks. The channel fk is used by the forks to perform actions that require interaction with philosophers. Lastly, the life channel is used by the philosophers to state which individual action they are doing.

channels
pfk : RANGE * RANGE * S_ACTION * REQUISITION
fk : RANGE * RANGE * S_ACTION * REQUISITION
life : RANGE * I_ACTION

Additionally, some functions that are useful in the description of the behaviour of forks and philosophers are described. The next gives the increment by one of a natural number i, passed as a parameter, modulo the constant N. Similarly, the prev gives the decrement by one of a natural number i, passed as a parameter, modulo the constant N.

functions
next: nat -> nat
   next(i) == ( (i+1) mod N)

prev: nat -> nat
   prev(i) == ( (i-1) mod N)

After the description of these preliminary constructs, we start describing the basic processes, which model the behaviour of forks and philosophers. We begin by the introduction of the philosopher. The philosopher process called Phil is parametrised by a natural number i. This number is used to distinguish the behaviour of the various philosophers a model can have. The process philosopher has only one main action describing its behaviour. A philosopher is initially up, then he sits down, then it acquires its left-hand side fork, identified by i, then he acquires the fork on its right-hand side, identified by next(i). After these acquisitions, he can eat. After eating, he releases the left and then the right-hand side fork. Finally, he gets up again,
process Phil = i : nat @
begin
actions
  Main =
  ( life.i.<sitsdown> ->
    pfk.i.i.<picksup>.<req> ->
    pfk.i.i.<picksup>.<ack> ->
    pfk.(next(i)).i.<picksup>.<req> ->
    pfk.(next(i)).i.<picksup>.<ack> ->
    life.i.<eats> ->
    pfk.i.i.<putsdown>.<req> ->
    pfk.i.i.<putsdown>.<ack> ->
    pfk.(next(i)).i.<putsdown>.<req> ->
    pfk.(next(i)).i.<putsdown>.<ack> ->
    life.i.<getsup> -> Main
  ) @ Main
end

when this cycles restart. This behaviour is given by the action \textbf{Main}.

The model also requires an asymmetric philosopher, which is modelled by the process \textbf{APhil}. The asymmetric philosopher is also parametrised by a natural number, which is used to parametrise the behaviour of the philosopher. The behaviour of this process is given by the action \textbf{Main}. Its behaviour is very similar to the regular philosopher, the only difference being the order in which it acquires its forks: he acquires the fork on the right-hand side before acquiring the fork on the left-hand side.

Finally, we introduce the process describing a fork, which is given by process \textbf{Fork}. This process is, similarly, parametrised by a natural number, which is used to distinguish the forks. The behaviour of a fork is given by an action called \textbf{Main}. This action states that the fork is initially released and can be acquired by the philosophers on its left-hand side, or by its right-hand side philosopher. After being acquired, only the philosopher which acquired it can release it.

process Fork = i : nat @
begin
actions
process APhil = i : nat @
begin
actions
  Main =
    ( life.i.<sitsdown> ->
      pfk.(next(i)).i.<picksup>.<req> ->
      pfk.(next(i)).i.<picksup>.<ack> ->
      pfk.i.i.<picksup>.<req> ->
      pfk.i.i.<picksup>.<ack> ->
      life.i.<eats> ->
      pfk.(next(i)).i.<putsdown>.<req> ->
      pfk.(next(i)).i.<putsdown>.<ack> ->
      pfk.i.i.<putsdown>.<req> ->
      pfk.i.i.<putsdown>.<ack> ->
      life.i.<getsup> -> Main )
) @
Main
end

Main =
  ( fk.i.i.<picksup>.<req> ->
    fk.i.i.<picksup>.<ack> ->
    fk.i.i.<putsdown>.<req> ->
   fk.i.i.<putsdown>.<ack> -> Main []
    fk.i.(prev(i)).<picksup>.<req> ->
    fk.i.(prev(i)).<picksup>.<ack> ->
    fk.i.(prev(i)).<putsdown>.<req> ->
    fk.i.(prev(i)).<putsdown>.<ack> -> Main )
) @
Main
end
3.1.1 The BRIC approach

After describing these CML processes, we are able to describe the contracts of our constituent systems to build our model using the BRIC strategy. We define which are the initial constituent contracts, describing also the inputs and outputs function for these contracts. Besides the definition, we claim why the contracts defined are I/O contracts, as required by BRIC. After this, we apply the rules of composition to create the asymmetric dining philosophers from these initial contracts. For each application of the rules, we also claim why the side conditions are satisfied and what is the resulting contract.

To begin with, we introduce the contract describing a philosopher. We also use a natural number $i$ to parametrise a philosopher’s contract. The behaviour of this constituent is given by the process $\text{Phil}(i)$. The channels used for interaction are the $\text{pfk}.i.i, \text{pfk}.next(i).i$. The type communicated by this channel is $\text{S}_\text{ACTION} \ast \text{REQUISITION}$. The channel $\text{life}.i$ is used to signal internal actions of the philosopher, and has type $\text{I}_\text{ACTION}$.

**Definition 3.1 (Regular Philosopher contract)**

$$ Ctr_{\text{Phil}}(i) \cong \langle \begin{array}{l}
\text{Phil}(i), \\
\{ \text{pfk}.i.i \mapsto \text{S}_\text{ACTION} \ast \text{REQUISITION}, \\
\{ \text{pfk}.next(i).i \mapsto \text{S}_\text{ACTION} \ast \text{REQUISITION}, \\
\{ \text{life}.i \mapsto \text{I}_\text{ACTION} \\
\{ \text{S}_\text{ACTION} \ast \text{REQUISITION} \\
\} \{ \text{pfk}.i.i, \text{pfk}.next(i).i, \text{life}.i \} \\
\rangle \end{array} \rangle $$

The input and output functions for this contract are defined as follows. The inputs of a philosopher are the acknowledgement events, whereas the outputs are the events of request.

$$ \text{inputs}(\text{pfk}.i.i, Ctr_{\text{Phil}}(i)) = \\
\{ \text{pfk}.i.i.\text{action}.<\text{ack}> \mid \text{action} : \text{S}_\text{ACTION} \} $$

$$ \text{inputs}(\text{pfk}.next(i).i, Ctr_{\text{Phil}}(i)) = \\
\{ \text{pfk}.next(i).i.\text{action}.<\text{ack}> \mid \text{action} : \text{S}_\text{ACTION} \} $$

$$ \text{outputs}(\text{pfk}.i.i, Ctr_{\text{Phil}}(i)) = \\
\{ \text{pfk}.i.i.\text{action}.<\text{req}> \mid \text{action} : \text{S}_\text{ACTION} \} $$

$$ \text{outputs}(\text{pfk}.next(i).i, Ctr_{\text{Phil}}(i)) = \\
\{ \text{pfk}.next(i).i.\text{action}.<\text{req}> \mid \text{action} : \text{S}_\text{ACTION} \} $$

48
In order to be a valid contract, the contract’s behaviour must be an I/O process. Hence, it must satisfy the following five conditions:

1. I/O Channels: It satisfies this condition since inputs and outputs are disjoint for each channel, inputs have as \texttt{REQUISITION} the value \texttt{<ack>}, whereas outputs have the value \texttt{<req>}. Additionally, the inputs and outputs are events of the appropriate channels, as they are obviously extensions of it.

2. Non-terminating: the process \texttt{Phil} does not terminate, since it recurses and it makes no use of either the \texttt{Stop} or \texttt{Skip} process.

3. Divergence freedom: The process is guarded and make no use of hiding, hence it is non-divergent.

4. Input Determinism: The process does not use choices operator, hence the inputs offered are a full choice.

5. Strong Output Decisive: The process does use choices operator, hence it leaves no choice to the environment, as required by this property.

The contract of an asymmetric philosopher is very similar to the one of the regular philosopher. The only difference between this contract and the previous one is that the behaviour of this contract is given by the \texttt{APhil(i)} process, instead of the regular \texttt{Phil(i)} one.

**Definition 3.2 (Asymmetric Philosopher contract)**

\[
Ctr_{APhil(i)} \equiv \left\langle \begin{array}{l}
APhil(i), \\
pfk.i.i \mapsto S\_ACTION \ast \text{REQUISITION}, \\
pfk.next(i).i \mapsto S\_ACTION \ast \text{REQUISITION}, \\
life.i \mapsto I\_ACTION \\
\{S\_ACTION \ast \text{REQUISITION}\}, \\
\{pfk.i.i, pfk.next(i).i, life.i\} \\
\end{array} \rightangle
\]

The inputs and outputs function for an asymmetric philosopher are exactly the same as the ones for the regular philosopher.

This contract also needs to be an I/O one to be used in the \texttt{BRIC} strategy. It is the case, as it satisfies the five I/O contract conditions. We do not demonstrate their validity here, because the claim is identical to the one presented for the regular philosopher.

Considering the forks, their contract is quite similar to the previous two. The contract, as usual, is parametrised by a natural number \textit{i}. The behaviour of a
fork is given by the process $\text{Fork}(i)$, previously described. It uses for communication, the channel $\text{fk}.i.i$ and $\text{fk}.i.prev(i)$. Similarly to the previous contracts, the type of these channels is given by $\text{S}\_\text{ACTION} \ast \text{REQUISITION}$.

**Definition 3.3 (Fork contract)**

$$\text{Ctr}_{\text{Fork}}(i) \triangleq \langle \text{Fork}(i), \{ \text{fk}.i.i \mapsto \text{S}\_\text{ACTION} \ast \text{REQUISITION}, \text{fk}.i.prev(i) \mapsto \text{S}\_\text{ACTION} \ast \text{REQUISITION}, \} \rangle$$

The *inputs* and *outputs* of the forks are given by the following functions.

- **inputs**($\text{fk}.i.i, \text{Ctr}_{\text{Fork}}(i)$) = 
  $$\{ \text{fk}.i.i\_\text{action}.<\text{req}> \mid \text{action} : \text{S}\_\text{ACTION} \}$$

- **inputs**($\text{fk}.i.prev(i), \text{Ctr}_{\text{Fork}}(i)$) = 
  $$\{ \text{fk}.i.prev(i)\_\text{action}.<\text{req}> \mid \text{action} : \text{S}\_\text{ACTION} \}$$

- **outputs**($\text{fk}.i.i, \text{Ctr}_{\text{Fork}}(i)$) = 
  $$\{ \text{fk}.i.i\_\text{action}.<\text{ack}> \mid \text{action} : \text{S}\_\text{ACTION} \}$$

- **outputs**($\text{fk}.i.prev(i), \text{Ctr}_{\text{Fork}}(i)$) = 
  $$\{ \text{fk}.i.prev(i)\_\text{action}.<\text{ack}> \mid \text{action} : \text{S}\_\text{ACTION} \}$$

Again, the behaviour of this contract must be an I/O process. It is fortunately the case, as it satisfies the five I/O contract conditions. The justifications are very similar to those for the philosophers contracts and are omitted here for conciseness. The only difference is the justification for I/O channels: It satisfies this condition since, for each channel, inputs and outputs are disjoint(8,8),(994,993). inputs have as $\text{REQUISITION}$ the value $<\text{req}>$, whereas outputs have the value $<\text{ack}>$. Additionally, the inputs and outputs are events of the appropriate channels, as they are obvious extensions of it.

After the definition of the initial contracts, we start composing them to build the dining philosophers system. Here, we describe the construction of an asymmetric dining philosophers with $N = 3$ (where $N$ is the number of philosophers, or forks). Hence, we have the following initial contracts available for composition, note that we need to have an asymmetric philosopher contract. As demonstrated, all of them are I/O contracts.

$$\text{Ctr}_{\text{Phil}}(0), \text{Ctr}_{\text{Phil}}(1), \text{Ctr}_{\text{APhil}}(2), \text{Ctr}_{\text{Fork}}(0), \text{Ctr}_{\text{Fork}}(1), \text{Ctr}_{\text{Fork}}(2)$$
Even though we only present the strategy of composition for a single instance of the problem, where $N = 3$, as we parametrise the contracts, one could use a strategy, similar to the one presented here, to build a system with an arbitrary $N$. We begin the application of the BRIC strategy by the composition of the philosophers processes. We start by composing the philosophers using the interleaving rule. First, we compose the $\text{Ctr}_{\text{Phil}}(0)$ with $\text{Ctr}_{\text{Phil}}(1)$, resulting in $\text{Ctr}_{\text{Phil}}_{01}$. Then, we compose this resulting contract $\text{Ctr}_{\text{Phil}}_{01}$ with $\text{Ctr}_{\text{APhil}}(2)$, resulting in the contract $\text{Ctr}_{\text{Phils}}$.

$$\text{Ctr}_{\text{Phils}} = \text{Ctr}_{\text{Phil}}_{01} || \text{Ctr}_{\text{APhil}}(2)$$

These compositions can be performed because their side conditions are fulfilled. For this rule, the channels of the contracts composed must be disjoint. This is the case since, the philosophers have channels of the form $\text{pfk}.x.i$ and $i$ is an unique natural number, identifying a philosopher.

After composing the philosophers in interleaving, we do this same process with the fork contracts. We start by composing $\text{Ctr}_{\text{Fork}}(0)$ and $\text{Ctr}_{\text{Fork}}(1)$ using the interleaving rule, creating the contract $\text{Ctr}_{\text{Fork}}_{01}$. Then, we compose this resulting contract with $\text{Ctr}_{\text{Fork}}(2)$, what results in the contract $\text{Ctr}_{\text{Forks}}$.

$$\text{Ctr}_{\text{Forks}} = \text{Ctr}_{\text{Fork}}_{01} || \text{Ctr}_{\text{Fork}}(2)$$

To help depicting the presented steps, we introduce a graphical notation to show the internal connections made between constituent systems. The blue boxes depict initial constituent systems, the black boxes channels of these constituents, and the black lines, the connections between channels.
Note that, in \textit{BRIC}, connections are made using buffers. Lastly, the grey boxes, involving some of the components and some of the channels represent a contract encompassing the involved constituents. The channels inside the grey boxes are no longer available for composition, whereas the ones outside are. The result of these initial interleavings is depicted in Figure 3.

\[
\begin{align*}
\text{Ctr}_{\text{Forks}} &= \text{Ctr}_{\text{Fork}}(0) \ || | \text{Ctr}_{\text{Fork}}(1) \\
\text{Ctr}_{\text{Forks}} &= \text{Ctr}_{\text{Fork}}(2)
\end{align*}
\]

These compositions can also take place, since their side conditions are satisfied. As mentioned, the channels of the contracts being composed must be disjoint, and that is the case for them. Note that the channels of the forks are of the form \(f_k.i.x\), where \(i\) is a unique natural number identifying the fork.

After this, we compose these two contracts (\(\text{Ctr}_{\text{Phils}}\) and \(\text{Ctr}_{\text{Forks}}\)) using the communication rule, in the channels \(pf_k.0.0\) and \(f_k.0.0\). The result of this composition is depicted in Figure 4.

\[
\begin{align*}
\text{Ctr}_{\text{Phils}_\oplus \text{Forks}} &= \text{Ctr}_{\text{Phils}}[pf_k.0.0 \leftrightarrow f_k.0.0] \text{Ctr}_{\text{Forks}}
\end{align*}
\]

This composition can only occur because the side conditions are met. The contracts work on different channels, one has only \(f_k\) channels and the other only \(pf_k\) ones, and the renamed version of their protocols, given next, are I/O confluent and strong compatible and satisfy the finite output property.

To illustrate the satisfaction of these properties, we present the functions giving the port protocols in the channel \(pf_k\), and in the channel \(f_k\). These functions are described using pattern matching.
D2.4.4 - Comp. Anal. and Des. of CML Models (Public) COMPASS

Prot.\_\_IMP(fk.\text{id.philId}) =
fk.\text{id.philId}.\text{picksup} \rightarrow fk.\text{id.philId}.\text{picksack} \rightarrow
fk.\text{id.philId}.\text{putsdown} \rightarrow fk.\text{id.philId}.\text{putsack} \rightarrow
Prot._\_IMP(fk.\text{id.philId})

Prot.\_\_IMP(pfk.\text{forkId.id}) =
pfk.\text{forkId.id}.\text{picksup} \rightarrow pfk.\text{forkId.id}.\text{picksack} \rightarrow
pfk.\text{forkId.id}.\text{putsdown} \rightarrow pfk.\text{forkId.id}.\text{putsack} \rightarrow
Prot._\_IMP(pfk.\text{forkId.id})

These protocols are I/O confluent, since no choice is offered, an environmental choice does not preclude another one. They are also strong compatible, since each output of one is input for the other one, and vice-versa. Lastly, both processes satisfy the finite output property, since, for both protocols, after an output there must be an input.

After this step, we proceed by linking all channels still available for composition using the feedback rule, as follows. Note that, after the last feedback composition, which yielded the contract $\text{Ctr}_{\text{Phils}\_\text{Forks}_0}^{2\_0}$, there is still a pair of channels to be composed, $\text{pfk.0}\_2$ and $\text{fk.0}\_2$. Nevertheless, these channels are not decoupled, i.e. the behaviour of process on these two channels is not independent. Therefore, a composition involving one of them might affect the behaviour on the other channel. Note that, the composition of a deadlocked component in one of these channels deadlocks the system, which clearly demonstrates the dependence between these channels. Hence, the feedback rule cannot be applied to compose these channels. The result of applying these feedback compositions is depicted in Figure 5.

\[
\begin{align*}
\text{Ctr}_{\text{Phils}\_\text{Forks}_0}^{2\_1} &= \text{Ctr}_{\text{Phils}\_\text{Forks}_0}^{2\_0}[\text{pfk.1}\_1 \leftrightarrow \text{fk.1}\_1] \\
\text{Ctr}_{\text{Phils}\_\text{Forks}_0}^{2\_2} &= \text{Ctr}_{\text{Phils}\_\text{Forks}_1}^{2\_1}[\text{pfk.2}\_2 \leftrightarrow \text{fk.2}\_2] \\
\text{Ctr}_{\text{Phils}\_\text{Forks}_0}^{2\_0} &= \text{Ctr}_{\text{Phils}\_\text{Forks}_2}^{2\_2}[\text{pfk.1}\_0 \leftrightarrow \text{fk.1}\_0] \\
\text{Ctr}_{\text{Phils}\_\text{Forks}_0}^{2\_1} &= \text{Ctr}_{\text{Phils}\_\text{Forks}_0}^{2\_1}[\text{pfk.2}\_1 \leftrightarrow \text{fk.2}\_1]
\end{align*}
\]

Note that these compositions are allowed because they satisfy the feedback rule side conditions. This means that the renamed protocols of the channels involved in these compositions are I/O confluent, strong compatible and respect the finite output property. Additionally, the channels composed must also be decoupled, and that is the case for these compositions. Note that, inside behaviour of $\text{Ctr}_{\text{Phils}\_\text{Forks}_0}^{2\_0}$, the processes Phil(1) and Fork(1), this is the reason why pfk.1\_1 and fk.1\_1 are decoupled. The same reasoning can be used to demonstrate why the other composed channels are
Finally, we finish the design of our system by using the reflexive rule to compose channels \( pfk.0.2 \) and channel \( fk.0.2 \). As described, the feedback rule is not applicable anymore, since these channels are not decoupled. Therefore, we must apply a rule which has more flexible constraints, which is the case of the reflexive rule. The result of this composition is shown in Figure 6.

\[
Ctr_{College} = Ctr_{Phil_Forks_2_1}[pfk.0.2 \Leftrightarrow fk.0.2]
\]

Again, this composition is only allowed because its side conditions are respected. The projection of the behaviour of the constituent \( Ctr_{Phil_Forks_2_1} \) on the channels \( pfk.0.2 \) and \( fk.0.2 \), which is given by \( B_{Phil_Forks_2_1} \mid \{pfk.0.2,fk.0.2\} \), is buffering self-injection compatible and satisfies the finite output property. The reason why buffer self injection compatibility is
achieved is because the addition of a buffer to link these channels does not cause a deadlock.

With this last step, we obtain the college contract, $\text{Ctr}_{\text{College}}$, which models the asymmetric dining philosophers for a $N = 3$. Again, a similar process can, simply, be employed to construct a system with an arbitrary $N$, as we use parametrised processes and contracts. We could use the interleaving to compose the philosophers and then the forks, regardless of the $N$ used. After, these contracts can be combined using the communication rule. Next, we can use the feedback rule to compose all the remaining pairs of communication channels, but the last one, since it is not decoupled. We then use the reflexive rule to finish the design of the system models. This case study (and its variations) is used later for the demonstration of how a local strategy can be combined to $\text{BRIC}$ so as to enable a more efficient deadlock analysis. In addition, similarly to what is done for deadlock analysis, this case study is used to demonstrate how an strategy based on static analysis can be used for efficient livelock analysis.

### 3.2 Leadership Election

A home Audio/Video (AV) network consists of several devices (such as audio, video, gateway and legacy audio products) which may be produced by competing manufacturers and distributed across a user’s home. The network is an SoS; it exhibits the dimensions typical of an SoS as described in [FFI+13].

- The individual Constituent Systems (CSs) exhibit a (potentially) wide variation in autonomy. They all operate at the behest of the user, but the fact that they may be legacy or well known systems means that they may only offer a limited degree of controllability from the point of view of the SoS.

- The CSs exhibit operational independence; they provide stand-alone streaming or content browsing experiences, e.g. watching TV or selecting music to play.

- The CSs are typically distributed in different zones/rooms, the AV content can be local or remote, and the location of content source is often transparent to the user.

- Geographical distribution leads to emergent behaviors such as making sound follow the user around, driven by contracts between streaming
and clock systems.

- The CSs undergo *evolutionary development*. The stakeholders will have an evolution vision that is not necessarily compliant with that of the system’s manufacturer.

- There is *dynamic reconfiguration behavior* in that products join or leave the SoS during streaming or browsing operations; products can be turned off by users or enter power-saving mode.

- While products have no interdependence, CSs rely on each other in order to deliver the emergent behavior that fulfills the SoS goal.

Constituent systems may join or leave the network at any time, but a consistent user experience (such as a playlist, current song, etc.) must be provided, and this requires availability and consistency of the system configuration data. In order to do this, a publish-subscribe architecture is employed. In this SoS, the chosen architecture requires that the underlying network is able to elect a leader from among the CSs, where the leader is responsible for distributing the global system configuration (containing e.g. network time and current playlist) to the followers in the SoS. As there is no centralised control, the ability to elect a leader is a required emergent property of the SoS: the way that the nodes interact must produce behaviour that individual nodes cannot produce on their own. In the following study, we present a CML model that captures this emergent behaviour.

### 3.2.1 Leadership Election Formal Model

In our model, products are represented by nodes, which have an internal memory used to store information about the current state of the network. The communication between these nodes is given by a transport layer, which is in turn composed of smaller entities called bus cells. These small entities provide a unidirectional point-to-point communication between two nodes in the network of products. Moreover, a relevant feature of this transport layer is the fact that it can detect whether nodes are on or off. The SoS consists of a fully connected network, where the nodes exchange messages via bus cells. These exchanged messages are formed by a priority (or petition), which is a natural number representing the eagerness of the node to become a leader, and a claim, which represents the state of the node in the election process (undecided, leader or follower). The leader is elected based on the priority of the nodes. After this brief introduction to the system and the model, we describe in details the CML model conceived.
To begin with, we describe the datatypes used in the model. The type \texttt{IO} is used to express the directionality of the event on a channel. The type \texttt{CLAIM} contains the possible claims that a node can make. It also contains a token \texttt{<minusone>}, used in the bus cell process to indicate it is empty. \texttt{PACK} is a record containing a claim and a priority, this represents the data exchanged between nodes. The \texttt{NODE\_COMM} type describes the messages exchanged between nodes and bus cells. The \texttt{MEM\_PACK} type is a record containing an \texttt{id} and a pack. This data informs the current state (pair of priority and claim), of the node identified by \texttt{id}. \texttt{LEADERS, HPETITION} and \texttt{HPETITIONID} are types representing the data exchanged between a node’s memory and the node itself for determining the number of leaders, the highest petition and the \texttt{id} of the node having the highest petition, respectively. \texttt{MEM\_COMM} represents the type describing the possible interaction between a node and its memory. \texttt{NODES\_IDS} is the set of node identifiers, and \texttt{DIST} is the distribution of the priority values. \texttt{N\_RANGE} is a type representing the interval of the number of leaders of an election round.

\texttt{types}
\begin{verbatim}
IO = <inn>|<out>
CLAIM = <leader> | <follower> | <undecided>
    | <off> | <minusone>
PACK :: c : CLAIM
    p : DIST
NODE\_COMM = <isOn> | <isOff> | <req> | <ack>
    | PACK | <timeout>
MEM\_PACK :: id : NODES\_IDS
    p : PACK
LEADERS :: n : N\_RANGE
HPETITION :: n : DIST
HPETITIONID :: n : NODES\_IDS
MEM\_COMM = MEM\_PACK | LEADERS | HPETITION | HPETITIONID
    | <reqLeaders> | <reqHpetition> | <reqHpetitionid>

NODES\_IDS = nat
    inv n == n in set NODE\_SET
DIST = nat
    inv d == d in set DIST\_SET
N\_RANGE = nat
    inv n == n in set \{0,...,NODES\}
\end{verbatim}
Values used throughout the specification are below. Note that some of the values are, simply, a set representing a type. The reason is that one cannot use a type as a set expression in CML. NODES is the number of nodes; N is the upper bound of the identifiers of nodes; LOWER_LIMIT_PET and UPPER_LIMIT_PET are the lower and upper limit of the petition values respectively. NODE_SET is a set containing the ids of the nodes, used for replication. CLAIM_SET is the set of values of type CLAIM. Similarly, the set DIST_SET has the values of the type DIST. The following sets are used to restrict the communication on some channels. The set PACKS, MEM_PACKS and TPACKS are composed of the values of type PACK, MEM_PACK and PACK plus the timeout event, respectively. Similarly, LEADERS_SET, HPETITION_SET and HPETITIONID_SET are composed of the values of types LEADERS, HPETITION and HPETITIONID, respectively.

values

NODES : nat = 2
N : nat = NODES-1
LOWER_LIMIT_PET : nat = 0
UPPER_LIMIT_PET : nat = 1

NODE_SET : set of nat = {0,...,N}
CLAIM_SET = { c | c : CLAIM}
DIST_SET = {0,...,UPPER_LIMIT_PET}

PACKS = { p | p : PACK}
MEM_PACKS = { mp | mp : MEM_PACK}
TPACKS = {<timeout>} union PACKS

LEADERS_SET = {l | l : LEADERS}
HPETITION_SET = {e | e : HPETITION}
HPETITIONID_SET = {e | e : HPETITIONID}

The channels used in the model are below. The sender channel is used by a node to interact with the bus cell when behaving as a sender. The counterpart of this sender channel used by the bus cell for this interaction is the sender_bus. In a similar way, the receiver channel is used by the node to interact with a bus cell, as a receiver. Its counterpart used by the bus cell is the receiver_bus channel. As we present later, the two initial arguments of this channels (two node identifiers) convey the source and destination of the data being transmitted. For instance, the channel sender.0.1, is used by node 0, to send data to node 1. Finally, the channels node_mem is used by the node to communicate with its memory, and its counterpart used by
the memory to realise this communication is the `mem_node`.

channels

sender : NODES_IDS * NODES_IDS * IO * NODE_COMM
sender_bus : NODES_IDS * NODES_IDS * IO * NODE_COMM
receiver : NODES_IDS * NODES_IDS * IO * NODE_COMM
receiver_bus : NODES_IDS * NODES_IDS * IO * NODE_COMM

node_mem : NODES_IDS * IO * MEM_COMM
mem_node : NODES_IDS * IO * MEM_COMM
setCellPack, getCellPack : NODES_IDS * NODES_IDS * CLAIM * DIST

After introducing this preliminary definitions, we start presenting the processes describing the elements of this system. For all the processes in the sequel, we only present their actions and their main action. We disregard the other elements for the sake of readability. When a function, or other relevant element, appears, we simply introduce it informally. The full CML specification can be found in [LEM]. The first constituent introduced is the memory, which is given by the `Ram` process. The parametrised action `CellPack` represents a memory cell, holding the claim and petition of a node of the network. It may receive a new claim/petition pair (through `setCellPack`) or give out the current value (through `getCellPack`).

process Ram =
begin
... actions
CellPack = idN : NODES_IDS, idd : NODES_IDS,
c : CLAIM, d : DIST @
(
  setCellPack.idN.idd?valc?vald ->
    CellPack(idN,idd,valc,vald)
   []
  getCellPack.idN.idd!c!d -> CellPack(idN,idd,c,d)
)

The `StatePack` is an interleaving of NODES-1 `CellPack`s: one for each of the neighbouring Nodes. This represent the memory cells used to store information about each other node of the network.

StatePack = idN : NODES_IDS, cells_ids : set of NODES_IDS @
ControllerPack manages the memory cells, by controlling the interactions with the outside world. The parameter \(idN\) is the identity of the “owning” Ram process. ControllerPack keeps track of the highest petition, the highest petition id and the number of leaders in the environment. It can receive a request to update the current state of a node (a triple containing the identifier, the claim and the petition of the node in question), via the channel mem_node. In this case, it first retrieves the old values (via getCellPack, sets the new values (channel setCellPack) then evaluates new values for the highest petition, the identity of the node with the highest petition, and the new value for the number of leaders. The ControllerPack is also always prepared to transmit the current values of these variables, on channels nleaders, hpetition and hpetitionid. The overall memory of a node is given by the action MemePack, which combines ControllerPack and StatePack and hides the interaction between them.

\[
\text{MemePack} = \text{id: NODES_IDS,cells_ids : set of NODES_IDS} @ \begin{cases} 
\text{ControllerPack(id,cells_ids, 0, 0, 0)} \\
\text{getCellPack, setCellPack} \\
\text{StatePack(id,cells_ids)} \\
\text{getCellPack, setCellPack}
\end{cases}
\]

The RAM main action is an initialisation of the MemePack processes with a node \(id\) and the ids of the rest of the nodes in the network. The function setdiff stands for set difference.

\[
\text{let diff = setdiff(i) in MemePack(i,diff)}
\]

The BusCell process is relatively simple. When the bus cell detects that its sender node is on, it behaves as the BusCellOn action. In such state, it may receive a new claim and petition at any time on channel sender_bus, which is stored in the bus. If the bus cell has some data, and a new pair arrives the data is overwritten. Provided it has data to transmit, it may transmit it on the channel receiver_bus. Moreover, it may receive a turning off event from its sender, where it starts behaving as the process BusCellIdle.

When it has detected that its sender is turned off, it starts behaving as the action BusCellIdle. In this state, it offers a receiving behaviour where a timeout is offered to its receiver node. This timeout event signals that the sender node is turned off and, consequently, it will not offer any data to the receiver node. If the sender node sends a turning on event, then the bus
ControllerPack = idN : NODES_IDS, cells_ids : (set of NODES_IDS),
  highest_petition : DIST, highest_petition_id : NODES_IDS,
  leaders : N_RANGE @
    [leaders < NODES and leaders >= 0] &
    (mem_node.idN.<inn>?x:(x in set MEM_PACKS) ->
      let v = getValue(x), id = v.id, pack = v.p,
        valNewClaim = pack.c, valNewPet = pack.p in
        getCellPack.idN.id?valOldClaim?valOldPet ->
        setCellPack.idN.id.valNewClaim.valNewPet ->
        (let newHighestPetition : DIST =
          getNewHighestPetition(highest_petition, valNewPet),
          newHighestPetitionId : NODES_IDS =
          getNewHighestPetitionId(highest_petition, valNewPet, id, highest_petition_id),
          newLeaders : nat =
          getNewLeaders(valOldClaim, leaders, valNewClaim)
        in
          ControllerPack(idN, cells_ids, newHighestPetition,
          newHighestPetitionId, newLeaders)
    )
  )}
[]
  [leaders < NODES and leaders >= 0] &
  mem_node.idN.<inn>.<reqLeaders> ->
  mem_node.idN.<out>.mk_LEADERS(leaders) ->
  ControllerPack(idN, cells_ids, highest_petition, highest_petition_id, leaders)
[]
  [leaders < NODES and leaders >= 0] &
  mem_node.idN.<inn>.<reqHpetition> ->
  mem_node.idN.<out>.mk_HPETITION(highest_petition) ->
  ControllerPack(idN, cells_ids, highest_petition, highest_petition_id, leaders)
[]
  [leaders < NODES and leaders >= 0] &
  mem_node.idN.<inn>.<reqHpetitionid> ->
  mem_node.idN.<out>.mk_HPETITIONID(highest_petition_id) ->
  ControllerPack(idN, cells_ids, highest_petition, highest_petition_id, leaders)

61
process BusCell = idSource : NODES_IDS, idTarget : NODES_IDS @
begin
  actions
    BusCellOn = c:CLAIM , d:DIST @
      sender_bus.idSource.idTarget.<inn>?x:(x in set PACKS) ->
        sender_bus.idSource.idTarget.<out>.<ack> ->
          BusCellOn(c,d)
        []
        [c <> <minusone>] &
          receiver_bus.idSource.idTarget.<inn>.<req> ->
            receiver_bus.idSource.idTarget.<out>.mk_PACK(c,d) ->
              BusCellOn(<minusone>,d)
        []
    sender_bus.idSource.idTarget.<inn>.<isOff> -> BusCellIdle
      []
    receiver_bus.idSource.idTarget.<inn>.<isOff> ->
      BusCellOn(c,d)

    BusCellIdle =
      receiver_bus.idSource.idTarget.<inn>.<req> ->
        receiver_bus.idSource.idTarget.<out>.<timeout> ->
          BusCellIdle
        []
      sender_bus.idSource.idTarget.<inn>.<isOn> ->
        BusCellOn(<minusone>,0)
      []
    receiver_bus.idSource.idTarget.<inn>.<isOff> ->
      BusCellIdle
  end

  cell detects that the sender has turned on and starts behaving as the action
  BusCellOn.

  The overall behaviour of the BusCell given by the action BusCellIdle.
    @
      BusCellIdle
  end

Before introducing the behaviour of the main constituent of our system,
we introduce an important sequence in our model. SEQ_NEIGHBOURS is
a sequence of all ids of the nodes, which guides the interaction between
The behaviour of the node is more complex than the behaviour of the previous constituents. The node is initially turned off, where it behaves as the action OffNode. In this state, it can only turn on. When it does, it signals to all the bus cells that have this node as a sender (behaviour given by the action BroadcastControl(<isOn>, SEQ_NEIGHBOURS)), that it has turned on. After this, it starts behaving as the OnNode action. Note that this node stores a priority which is decremented when it turns on, this is a strategy used to give the more stable nodes the highest priorities. When behaving as OnNode, the behaviour of the node is given by action Node’ composed sequentially with the Fail process. This means that the main behaviour of an turned on node is given by Node’ and, at some point, this action may behave as Skip, in which case the node has failed, and will start behaving as Fail. When behaving as Fail, the node sends turn off messages as a mechanism to abstract the fact that the transport layer detects this turning off action. Hence, it broadcasts to the bus cells having it as a sender, a turn off event. This broadcast behaviour is given by BroadcastControl(<isOff>, SEQ_NEIGHBOURS). After this, the node behaves as the OffNode action.

process Node = i : NODES_IDS @
begin
actions
    OffNode = prior: DIST @
    (BroadcastControl(<isOn>, SEQ_NEIGHBOURS);
     (let mx =
      getNewHighestPetition(LOWER_LIMIT_PET, prior-1)
     in OnNode(mx)
    )
    )

    OnNode = prior: DIST @
    Node’([], <undecided>, prior); Fail(prior)

    Fail = prior: DIST @
    BroadcastControl(<isOff>, SEQ_NEIGHBOURS); OffNode(prior)

The behaviour of the node when is ready to communicate is given by the action Node’. At this point its behaviour is cyclic, and controlled by the
Node′ = s : seq of NODES_IDS, c : CLAIM, p : DIST @
if (s = [ ]) then BroadCastData(SEQ_NEIGHBOURS, c, p)
elseif (not (hd(s)=i)) then
  ((receiver.(hd(s)).i.<out>.<req> ->
    receiver.(hd(s)).i.<inn>?x:(x in set TPACKS) ->
    let pk = getValue(x)
    in node_mem.i.<out>.mk_MEM_PACK(hd(s),pk) ->
      Choice(tl(s), c, p))
|~|
  (receiver.(hd(s)).i.<out>.<isOff> -> Skip)
) else Choice(tl(s), c, p)

BroadCastData = s : seq of NODES_IDS, c: CLAIM, p: DIST @
if (s = [ ]) then Node′(SEQ_NEIGHBOURS, c, p)
elseif (hd(s)=i) then BroadCastData(tl(s),c,p)
else (sender.i.(hd(s)).<out>.mk_PACK(c,p) ->
  sender.i.(hd(s)).<inn>.<ack> ->
  BroadCastData(tl(s),c,p))
|~|
  Skip

sequence of nodes in the parameter s. When this sequence is empty, the node
is ready to broadcast data. It broadcasts its claim and petition to each of
the neighbouring nodes (via action BroadCastData). Note that, as specified
by this action, the node might behave as Skip, instead of sending data to
another peer. When the node behaves as Skip, a failure has occurred and
the node starts behaving as the action Fail. Moreover, this sending process
is made in a two-steps interaction, where the node sends the data first and
then wait for an acknowledgement. This two-steps interaction is atomic,
meaning that the node might not fail between sending data and receiving
an acknowledgment. After sending data to all its peers, the node starts
listening to its peers, waiting for data to be received. The receiving process
is also made with a two-steps interaction. First the node request some data
from a bus cell, then it receives either some actual data, or a timeout event.
Observe again, that the node, when receiving data from each of the other
nodes, may start behaving as Skip, signalling a failure. Furthermore, like for
sending, this two step interaction for receiving is atomic, meaning that the
node cannot fail between a request and receiving data. After receiving data
from a peer, the election process is triggered, via the action Choice.
Choice = s : seq of NODES_IDS, c : CLAIM, p : DIST @  
  if (c=<undecided>) then Undecided(s,p)  
  elseif (c=<leader>) then Leader(s,p)  
  elseif (c=<follower>) then Follower(s,p)  
  else Stop  

Follower = s : seq of NODES_IDS, p:DIST @  
  node_mem.i.<out>.<reqLeaders> ->  
  node_mem.i.<inn>?leaders:(leaders in set LEADERS_SET) ->  
    if (leaders.n = 0) then Node'(s,<undecided>,p)  
    else Node'(s,<follower>,p)  

The action Choice is responsible for choosing the subsequent behaviour according to the claim of the node.  

As a Follower, a node requests the number of leaders from its memory. If this is zero the leader of the system is no longer available, therefore, the node becomes Undecided. If a leader exists, the node remains as a follower of this leader.  

A leader also looks at the number of leaders. It becomes Undecided if this is less than 0, otherwise it remains a leader. Additionally, if this node is the leader at the end of a cycle (when \( s = [] \)), then it has its petition increased. This is a strategy to elect the most stable leader, as it is likely to have the highest priority.  

The Undecided action first collects the number of leaders, the highest leadership priority and the identifier of the node with the highest petition, then calculates its claim according to a simple algorithm based on these param-

Leader = s : seq of NODES_IDS, p:DIST @  
  node_mem.i.<out>.<reqLeaders> ->  
  node_mem.i.<inn>?leaders:(leaders in set LEADERS_SET) ->  
    if (leaders.n > 0) then Node'(s,<undecided>,p)  
    else (  
      if (s = []) then  
        (  
          let mn = getMinPetition(UPPER_LIMIT_PET,p+1)  
          in Node'(tl(s),<leader>,mn)  
        )  
      else Node'(s,<leader>,p)  
    )
Undecided = s : seq of NODES_IDS, p: DIST @
node_mem.i.in <out>. <reqLeaders> ->
node_mem.i.in <inn>.leaders: (leaders in set LEADERS_SET) ->
node_mem.i.in <out>. <reqHpetition> ->
node_mem.i.in <inn>.hppetition:
 (hppetition in set HPETITION_SET) ->
node_mem.i.in <out>. <reqHpetitionid> ->
node_mem.i.in <inn>.hppetitionid:
 (hppetitionid in set HPETITIONID_SET) ->
( dcl c: CLAIM @
  (if leaders.n > 0 then c := <follower>
   else if (s = []) then
    (if (((hppetition.n = p) and (hppetitionid.n < i))
      or (hppetition.n < p))
      then c := <leader>
      else c := <follower>)
    else c := <undecided>
  ); Node'(s, c, p)
); Node'(s, c, p)
@ OffNode(LOWER_LIMIT_PET)
end

eters. If a leader has been elected already, it becomes a follower. Otherwise, if it has reached the end of a cycle, which means that the node has contacted all its peers, then the election is finished, and the leader is the node having the highest petition. In case of a tie between nodes, the leader is the node with the highest identifier among the tied ones.

As stated in the beginning of the description of this process, a node starts turned off. Hence, the main action of this node is given by the action OffNode.

3.2.2 The BRIC approach

Based on the processes above, we describe the contracts of our constituent systems used to build our BRIC model. We define the initial constituent
contracts, describing also the inputs and outputs function for them. Besides the definitions, we explain why the contracts defined are I/O contracts, as required by BRIC. After this, we systematically build a system-of-systems following the BRIC approach. For each application of the rules, we also informally explain why the side conditions are satisfied and what is the resulting contract.

To begin with, we introduce the contract describing a memory. The behaviour \( B \) of a memory is given by the process \( \text{RAM}(i) \), the memory has only one channel \( \text{mem_node}.i \), through which it is connected to the node. The type of this channel, as described in the channels sections of our model is given by \( \text{IO} \times \text{MEM_COMM} \). The entire BRIC contract is given next.

\[
\text{Ctr}_{\text{RAM}}(i) \cong \left( \text{RAM}(i), \left\{ \text{mem_node}.i \mapsto \text{IO} \times \text{MEM_COMM} \right\}, \left\{ \text{IO} \times \text{MEM_COMM} \right\}, \left\{ \text{mem_node}.i \right\} \right)
\]

The input and output events of the memory contract are given by the following pattern matching expression. Basically, the events of the memory channel with the extensions containing \(<\text{inn}>\) are inputs and the ones having \(<\text{out}>\) are the outputs.

\[
\begin{align*}
\text{inputs}(\text{mem_node}.i, \text{Ctr}_{\text{RAM}}(i)) &= \{|\text{mem_node}.i.<\text{inn}>|\} \\
\text{outputs}(\text{mem_node}.i, \text{Ctr}_{\text{RAM}}(i)) &= \{|\text{mem_node}.i.<\text{out}>|\}
\end{align*}
\]

To claim that the contract \( \text{Ctr}_{\text{RAM}}(i) \) is a I/O contract, we argue the 5 required subconditions.

1. All channels in \( C \) are I/O channel:
   - This is clear, since inputs and outputs have different extensions \(<\text{inn}>\) and \(<\text{out}>\)

2. \( \text{RAM}(i) \) has infinite traces, but the state space is finite because CLAIM and DIST are finite types.

3. It always perform a visible action before performing an internal ones, hence it does not diverge.

4. Input Determinism: The input choices are deterministic.

5. Strong Output Decisive, because when it offers an output then a single one is offered at a time.
Hence we conclude that $\text{Ctr}_{\text{RAM}}(i)$ is a valid I/O contract.

Moving on, we introduce the contract of a bus cell $\text{Ctr}_{\text{BusCell}}(i,j)$. This contract is also parametrised, but differently from the previous one, two natural numbers ($i$ and $j$) are used for parametrisation. These naturals represent the natural numbers identifying the sender node ($i$), and the receiver node ($j$). The behaviour of a bus cell, as stated before, is given by the process $\text{BusCell}(i,j)$. A bus cell has two channels, one to communicate with its sender ($\text{sender}_{\text{bus}.i.j}$), and another to communicate with its receiver ($\text{receiver}_{\text{bus}.i.j}$). Both channels have as type $\text{IO} \times \text{NODE\_COMM}$. The contract is presented next.

$$\text{Ctr}_{\text{BusCell}}(i,j) \triangleq \langle \text{BusCell}(i,j), \{ \text{sender}_{\text{bus}.i.j} \mapsto \text{IO} \times \text{NODE\_COMM}, \text{receiver}_{\text{bus}.i.j} \mapsto \text{IO} \times \text{NODE\_COMM} \} \rangle$$

The inputs and outputs are also given as pattern matching expressions. Once again, the inputs and outputs events of a channel are given by the events containing $\langle \text{inn} \rangle$ and $\langle \text{out} \rangle$ extensions, respectively.

$$\text{inputs}(\text{sender}_{\text{bus}.i.j}, \text{Ctr}_{\text{BusCell}}(i,j)) = \{ |\text{sender}_{\text{bus}.i.j}.\langle \text{inn} \rangle| \}$$
$$\text{inputs}(\text{receiver}_{\text{bus}.i.j}, \text{Ctr}_{\text{BusCell}}(i,j)) = \{ |\text{receiver}_{\text{bus}.i.j}.\langle \text{inn} \rangle| \}$$
$$\text{outputs}(\text{sender}_{\text{bus}.i.j}, \text{Ctr}_{\text{BusCell}}(i,j)) = \{ |\text{sender}_{\text{bus}.i.j}.\langle \text{out} \rangle| \}$$
$$\text{outputs}(\text{receiver}_{\text{bus}.i.j}, \text{Ctr}_{\text{BusCell}}(i,j)) = \{ |\text{receiver}_{\text{bus}.i.j}.\langle \text{out} \rangle| \}$$

To demonstrate that this is an I/O contract, we claim the validity of the five required subconditions.

1. Again the extensions of inputs and outputs for these channels are disjoint consequently they are disjoint.

2. $\text{BusCell}(i,j)$ has infinite traces. $\text{BusCell}(i,j)$ has a very simple structure, and can be seen to have infinite traces by inspection. Again, the state space is finite, because the channels have finite types.
3. The behaviour is constructed without the use of hiding, hence it is divergence free.

4. Input Deterministic: There is no internal choice among the input events: $BusCell(i,j)$ is never selective about the input events it is prepared to receive.

5. Strong Output Decisive: when an output is offered, a single one is offered.

Thus we conclude that $Ctr_{BusCell}(i,j)$ is a valid contract.

The contract of a node, which is given by $Ctr_{Node}(i)$ is by far the most complex one. The reason is that it implements the business logic of our system. Also, it has to communicate with each other node in the system, what makes its channel structure not as simple as the previous cases. This contract is also parametrised by a natural numbers, which differentiates the several nodes in our system. The behaviour of a node is given by the process $Node(i)$. A node has a channel to communicate with the memory, and as many channels as peer nodes to carry inter node communication. To make this set of channels generic, no matter the number of nodes in the system, we create a set comprehension expression, using the range of $SEQ_{NEIGHBOURS}$ sequence, to describe this set of channels used for inter node communication. The type of the channel used for communication with the memory is given by the type expression $IO \times MEM_{COMM}$, whereas the type used by a channel used for communication between nodes is given by the type expression $IO \times NODE_{COMM}$. The entire contract is given next.

$$Ctr_{Node}(i) \triangleq \langle \begin{array}{c}
\{ \text{Node}(i), \\
\{ \text{sender}.i.j \mapsto IO \times NODE_{COMM}, \\
\text{receiver}.i.j \mapsto IO \times NODE_{COMM} \\
\mid j \leftarrow SEQ_{NEIGHBOURS}, j \neq i \} \\
\cup \{ \text{node_mem}.i \mapsto IO \times MEM_{COMM}, \\
\{ IO \times NODE_{COMM}, \\
IO \times MEM_{COMM} \} \\
\cup \{ \text{node_mem}.i \} \\
\cup \{ \text{sender}.i.j, \text{receiver}.i.j \\
\mid j \leftarrow SEQ_{NEIGHBOURS}, j \neq i \} \end{array} \rangle$$

The inputs and outputs events for the channels of this contract are given next. We used again pattern matching expressions for this purpose. Note that, the events having $<\text{inn}>$ in their structure are inputs and the ones having $<\text{out}>$ are outputs.
To validate this contract as an I/O contract, we argue why the 5 required subconditions hold:

1. inputs and outputs event have different structures, hence they are disjoint.

2. Node(i) has infinite traces. The structure of Node(i) is complex, but by inspection it can be seen to have infinite traces. The state space of Node(i) is finite, because the channels have finite types.

3. The hiding operator is not used. Hence, there is no possibility of divergence.

4. Input Deterministic: There is no internal choice among the input events: Node(i) is never selective about the input events it is prepared to receive.

5. Strong Output Decisive: When a output is performed, no choice is given among outputs.

After describing what the contracts are for each of the constituents of our system, we start introducing our strategy to create our leader election system-of-systems. Our system is parametrised by the number of nodes NODES. Hence, we can vary this number creating different instances of our SoS. We present our generic strategy, where NODES can be an arbitrary number. Nevertheless, to make the use of the strategy more concrete, and, by consequence, easier to follow, as we introduce the steps to create our generic SoS, we present how the step is applied to create a 3-nodes leader election system.

The first step in order to use BRIC is to define the initial contracts available for composition. For our system, we have NODES number of node contracts. Hence, we have a node contracts $Ctr_{Node}(i)$ where $i$ ranges from 0 to $N$, where $N = NODES - 1$. The system must have a pair of bus cells to connect each pair of communicating nodes. Hence, we have contracts $Ctr_{BusCell}(i,j)$, where $i$ and $j$ ranges from 0 to $N$, but $i$ and $j$ must be different naturals. Concerning the contracts of the memory, we have a $Ctr_{RAM}(i)$ for each node.
D2.4.4 - Comp. Anal. and Des. of CML Models (Public)  COMPASS

Figure 7: Representation of supernode 0

in the system. Hence, \( i \) varies from 0 to \( N \). For our explanatory 3-node system, we have the following initial contracts.

\[
\begin{align*}
C_{\text{Node}}(0), & \quad C_{\text{Node}}(1), \quad C_{\text{Node}}(2), \\
C_{\text{BusCell}}(0, 1), & \quad C_{\text{BusCell}}(0, 2), \quad C_{\text{BusCell}}(1, 0), \\
C_{\text{BusCell}}(1, 2), & \quad C_{\text{BusCell}}(2, 0), \quad C_{\text{BusCell}}(2, 1), \\
C_{\text{RAM}}(0), & \quad C_{\text{RAM}}(1), \quad C_{\text{RAM}}(2)
\end{align*}
\]

Having defined the initial contracts, we can start composing our constituent systems to create the leadership election SoS. We begin our strategy by composing each node to the bus cells that have this node as a receiver. For these compositions, we use the communication rule, as the constituents being composed in this step are always independent ones. The final contract resulting from the composition of a node to all bus cells having it as a receiver are denominated supernodes (given by contract \( C_{\text{Supernode}}(i) \), where \( i \) is the identifier of the node). Hence, for our 3-nodes model, we start with node 0. We compose \( C_{\text{Node}}(0) \) with \( C_{\text{BusCell}}(1, 0) \), using the communication rule, giving rise to the contract \( C_{\text{Node}}\_\text{BusCell}\_1\_0 \). Then, we take the resulting contract and compose it with the \( C_{\text{BusCell}}(2, 0) \) resulting in the supernode contract \( C_{\text{Supernode}}(0) \). We repeat this process to create supernodes 1 and 2. The supernode 0 contract is depicted in Figure 7.

\[
\begin{align*}
C_{\text{Node}}\_\text{BusCell}\_1\_0 &= C_{\text{Node}}(0)[\text{receiver} \_1 \_0 \leftrightarrow \text{receiver} \_\text{bus} \_1 \_0] C_{\text{BusCell}}(1, 0) \\
C_{\text{Supernode}}(0) &= C_{\text{Node}}\_\text{BusCell}\_1\_0 \\
&[\text{receiver} \_2 \_0 \leftrightarrow \text{receiver} \_\text{bus} \_2 \_0] C_{\text{BusCell}}(2, 0)
\end{align*}
\]
These compositions can only take place because the side conditions of the communication rule are met. The channels of the contracts being composed are obviously disjoint, and the renamed protocols for the channels used for composition are strong compatible, I/O confluent, and satisfy the finite-output property.

After creating the supernodes, we start composing them. We initially compose the supernode 0 with all other supernodes in the appropriate channels. The supernode 0 is composed with the other supernodes using the channel $sender_{0.i}$, where $i$ is the identifier of the supernode being connected to the supernode 0, and $i$ ranges from 1 to $N$. The counterpart channel used by the other supernodes for composition is the $sender_{bus_{0.i}}$. The graphical representation of the contract $Ctr\_Supernode\_0\_2$, which results of the compositions expected in this step of composition, is given in Figure 8.

\[
\begin{align*}
Ctr\_Supernode\_0\_1 &= Ctr\_Supernode\_0[sender\_0.1 \leftrightarrow sender\_bus\_0.1]Ctr\_Supernode\_1 \\
Ctr\_Supernode\_0\_2 &= Ctr\_Supernode\_0\_1[sender\_0.2 \leftrightarrow sender\_bus\_0.2]Ctr\_Supernode\_2
\end{align*}
\]

Again, these compositions can only be made because the side conditions of the communication rule are met. The channels of the contracts being composed are obviously disjoint, and the renamed protocols for the channels used for composition are strong compatible, I/O confluent, and they satisfy the finite-output property.

Note that, the final contract resulting of the application of the last step contain all the nodes and the bus cells of the system, but they are not completely connected. The only node that is fully connected to its transport entities is the node 0. Hence, we need to connect the remaining nodes to their bus cells. We make this by using the reflexive composition to compose $sender\_i.j$ with $sender\_bus\_i.j$ where $i$ ranges from 1 to $N$ and $j$ from 0 to $N$, but $i$ and $j$ must be different. Note that node $i$ has the remaining $sender\_i.j$ channels to be connected, and this step will connect all these channels. Moreover, observe that, in this step, we connect channels of the same contract, hence, communication rule cannot be used. Also, the channels used in these composition are not decoupled, as their communication depends of each other. Therefore, the only rule that can be applied is the reflexive one. At the end of this step, the resulting contract is given by $Ctr\_Core$. For our 3-node example, we take contract $Ctr\_Supernode\_2\_0$ and start composing the appropriate
channels, these composition are given next. The graphical representation of the contract $\text{Ctr}_{\text{Core}}$, for this example, is given in Figure 8.

Figure 8: Representation of supernode 0\_2

\[
\begin{align*}
\text{Ctr}_{\text{Supernode} \_0 \_2} =
\text{Ctr}_{\text{Supernode} \_1 \_0}[\text{sender} \_0 \_1 \leftrightarrow \text{sender} \_\text{bus} \_0 \_1] \\
\text{Ctr}_{\text{Supernode} \_1 \_2} =
\text{Ctr}_{\text{Supernode} \_1 \_0}[\text{sender} \_0 \_2 \leftrightarrow \text{sender} \_\text{bus} \_0 \_2] \\
\text{Ctr}_{\text{Supernode} \_2 \_0} =
\text{Ctr}_{\text{Supernode} \_1 \_2}[\text{sender} \_0 \_1 \leftrightarrow \text{sender} \_\text{bus} \_0 \_1] \\
\text{Ctr}_{\text{Core}} =
\text{Ctr}_{\text{Supernode} \_2 \_0}[\text{sender} \_0 \_2 \leftrightarrow \text{sender} \_\text{bus} \_0 \_2]
\end{align*}
\]

The reason why we can compose these channels using the reflexive rule is because their side conditions are met. For all of these composition, the buffer-self injection clause is satisfied.
Figure 9: Representation of the Core

The $C_{\text{Core}}$ contract has all the bus cell and all the nodes, and they are completely connected. The only channels available for composition are the ones that will link the nodes to their memories. Hence, the last step in creating our SoS is the composition of this core contract with the memory contracts. We use the communication rule, to make this compositions as these contracts are independent. Hence, we start by composing the $C_{\text{Core}}$ to $C_{\text{RAM}}(0)$ using channels $\text{node}_{-}\text{mem}.0$ and $\text{mem}_{-}\text{node}.0$. Then, we take the resulting contract and compose in the same way with the $C_{\text{RAM}}(1)$ using channels $\text{node}_{-}\text{mem}.1$ and $\text{mem}_{-}\text{node}.1$. This steps are repeated until all memories have been composed. The resulting contract $C_{\text{System}}$ is the contract describing our deadlock free system-of-systems. For the 3-node example, we have the following compositions leading to our final system contract. The graphical representation of the final system contract $C_{\text{System}}$ is given in Figure 10.
With this approach, we are able to construct, systematically, a deadlock free system-of-systems.

These case studies give a taste of how to apply BRIC in the constructions of systems. In the construction of these designs, we were able to introduce the application and the main concepts used by this strategy. Moreover, these examples serve to clarify the notions introduced in Section 2.3 and to make the user acquainted with both the methodology of BRIC and with the systems built. As a matter of fact, these systems are used throughout this work for experimentation and to introduce new concepts built on top of the BRIC strategy.

\begin{align*}
Ctr_{Core_0} &= \text{node\_mem\_0} \leftrightarrow \text{mem\_node\_0} \cdot Ctr_{RAM}(0) \\
Ctr_{Core_1} &= \text{node\_mem\_1} \leftrightarrow \text{mem\_node\_1} \cdot Ctr_{RAM}(1) \\
Ctr_{System} &= \text{node\_mem\_2} \leftrightarrow \text{mem\_node\_2} \cdot Ctr_{RAM}(2)
\end{align*}
4 Behavioural Patterns for Local Deadlock Analysis

In this section, we provide a strategy to do local deadlock analysis of cyclic models through adherence to communication patterns. The contribution of this section is based on Roscoe’s solution that takes behavioural patterns, described in Section 4.1 that reduce the verification effort, by allowing a local analysis of deadlock, even for cyclic communication topologies. Section 4.1 also describes how we incorporate the verification of adherence to behavioural patterns into BRICK.

In this section, we also formalise in BRICK some of the existing architectural patterns in the literature. The client/server pattern, which is used for architectures where components interact in a client/server fashion, i.e. the server provides services that are requested by clients, is described in Section 4.2. In Section 4.3, we present the resource allocation pattern, which can be used to model systems where constituents are competing for some shared resources. The version that we introduce here is a slight modification from the one presented in [ASW14]. This pattern allows local deadlock analysis of one of our case studies, the leadership election, discussed in Section 3.2. As a further contribution, we also present the formalisation of a novel pattern. In Section 4.4, we present the async dynamic pattern, which can be applied to networks with systems with two types of entities: the participants and the transport layer. In this architecture, the participants of the system do not interact directly with each other, but exchange messages via the transport layer.

The benefits of using behavioural patterns for local deadlock analysis are demonstrated with the application of this strategy to the two case studies introduced in Section 4. Our experiments, whose description and results can be found in Section 4.5, considered the asymmetric dining philosophers and the leadership election examples and demonstrated that both specifications are deadlock free using our extended strategy.

Finally, Section 4.6 presents a discussion on issues and ideas that we consider important in the design of a tool that would automate the extended strategy, which is in our research agenda.
4.1 Behavioural Patterns

As explained, in Section 2.3, buffering self-injection compatibility is verified over the entire system. This global verification of a system is known to be inefficient due, mainly, to state-space exponential growth (the state space explosion problem), a common problem in analysing concurrent systems. One way to avoid this problem is to impose behavioural patterns that allow local analyses and still guarantee deadlock freedom [Mar96, Ros98, AOS+14b, ASW14].

Behavioural patterns have been introduced in the CSP context by Martin and Roscoe [Mar96, Ros98]. In [AOS+14b, ASW14], we formalised and systematised a set of behavioural patterns. These behavioural patterns consist of a set of elements of interest and a set of restrictions. The elements of interest of a pattern describe some relevant entities of the system, which have a restricted behaviour and structure. For instance, in the Resource Allocation pattern, later described, we assume that constituents of the system are divided into two distinct sets: Users and Resources. These sets represent some of the elements of interest of this pattern. More specifically, they identify which constituents behave as users and which constituents behave as resources. This classification differentiates the constituent systems and enables the application of specific restrictions, depending on whether it is a resource or a user.

Additionally, a behavioural pattern imposes some restrictions on the system model. In our formalisation of behavioural patterns, we divide these restrictions into behavioural and structural ones. The former is defined using refinement expressions. The left-hand side of these expressions corresponds to a specification of the expected behaviour of the constituent on the right-hand side of these expressions. The advantage of using these expressions is that they can be automatically verified by a refinement checker. For the structural restrictions, we introduce some predicates using first-order logic and set theory to constrain the structure of the system. These predicates usually constrain the types of the elements that a system might have, their communication channel and the connections that can be performed.

In this section, we present an extension of BRICK that allows adherence to these behavioural patterns. We present these extensions in a systematic way: we enumerate them as steps, each of which describes what the extension is and why it is needed.

Firstly, the metadata needs to be extended to store information on the el-
elements of the system that are relevant for pattern conformance. The behavioral restrictions impose that the behaviour of the initial constituents must satisfy a specification. This imposition is not applied on the entire behaviour of the constituent, but the behaviour related with interaction between constituents, i.e. the behaviour involving the channels available in C. The structural constraints restrict which constituents can be initially available, what are the role played by these initially available constituents, the relevant communication channels, and which connections can be made between these initial contracts throughout compositions. Note that, all these restrictions are imposed on the initial contracts, except the restriction on the connections allowed between initial constituent contracts. Hence, the additional information that needs to be kept throughout compositions so that the system remains compliant to a pattern is the allowed connections it can perform. Therefore, we extend the metadata of a contract, i.e. K, by adding an element, denoted by Con (Connections), which is a set of pairs of channels, representing the intended, or allowed, future compositions. This extended metadata is denoted by $K_{\text{Patt}}^+$, and is defined as follows.

**Definition 4.1 ($K_{\text{Patt}}^+$)** Let Con be a set of pairs of channels. The extended pattern-based metadata pattern, denoted by $K_{\text{Patt}}^+$, is defined as:

$$K_{\text{Patt}}^+ = \langle \text{Prot}^K, \text{CTX}^K, \text{DProt}^K, \text{Dec}^K, \text{Con}^K \rangle$$

where Prot$^K$, CTX$^K$, DProt$^K$ and Dec$^K$ are exactly as in the original contract metadata.

Next, in order to accommodate this extended metadata in the contracts, we introduce an enrichment function. This new function, $\text{Enrich}_{\text{Patt}}$, takes an enriched BRICK contract and enriches it further with this new extended metadata. This function can be seen as a variation of the Enrich function previously presented, which simply replaces the metadata element of the contract by the new metadata structure defined. This function also provides a useful and elegant connection between this pattern-based version of BRICK, and the original enriched version. This function is formally defined as follows.

**Definition 4.2 (Pattern based enriched constituent contract)** Let $C_{\text{tr}}^e$ be an enriched constituent contract, Con$^K$ and $K_{\text{Patt}}^+$ as previously defined. A resource allocation enriched component contract that includes $C_{\text{tr}}^e$ is defined by:

$$\text{Enrich}_{\text{Patt}}(C_{\text{tr}}^e, Con^K) = \langle B_{C_{\text{tr}}^e}, R_{C_{\text{tr}}^e}, I_{C_{\text{tr}}^e}, C_{C_{\text{tr}}^e}, K_{\text{Patt}}^+ \rangle$$
The next step to introduce patterns into BRICK consists of an initial validation. This validation step ensures that the initial constituents are behaviourally compliant to the chosen pattern, that the initial contracts are structurally compliant, and, additionally, that the Con structure, for each initial constituent, describes only the connections allowed by the pattern for the given constituent. As one might notice, this validation is particular to each pattern, since each one of them imposes its own behavioural and structural restrictions. Hence, this initial step, which is denoted by the predicate INIT, needs to be defined for each of the patterns integrated to BRICK. As described later, we use a subscript to denote the instantiation of this predicate to a particular pattern. For instance, in the sequel we introduce the resource allocation pattern, where we present the $INIT_{RA}$ predicate.

Finally, the rules are extended to deal with the pattern conformance throughout compositions. For this matter, the side conditions of the rules must be revised to guarantee that the compositions preserve pattern adherence. Also, as the metadata is extended to deal with the allowed connections, the clauses for calculating the metadata resulting from the composition must be modified to handle this new element. As the connections are the only aspect to be concerned with throughout compositions so as to maintain pattern adherence, the side conditions and metadata calculation deal exactly with properly handling these connections. The side conditions are modified to allow only expected compositions, i.e. involving pair of channels in Con, and the metadata are calculated so as to update the set of allowed connections of the constituent after composition. For the presentation of these extensions, we reintroduce each of the composition rules making the appropriate modifications.

We begin by extending the interleaving rule. As there is no actual connection being made in this rule, there is no need for a change in the side conditions. Nevertheless, the calculation of the metadata must be extended to deal with the calculation of the Con element. This extension is a mere union of the Con structures for each of the contracts participating in this composition. The reason for the union is that the resulting contract has the two constituents composed and no connection has been made, hence the resulting contract is able to make any connection that the two constituents are allowed to make before the composition. This extension is formalised as follows.

**Definition 4.3 (Pattern-based enriched interleaving composition)**

Let the constituent contracts $Ctr_1^e = Enrich_{Patt}(Ctr_1^e, Con_1^K)$ and $Ctr_2^e = Enrich_{Patt}(Ctr_2^e, Con_2^K)$ be two pattern-based enriched component contracts, such that $Ctr_1^e$ and $Ctr_2^e$ satisfy the condition of the enriched interleaving
composition rule. Then, the pattern-based enriched interleaving composition of $\text{Ctr}^e_1$ and $\text{Ctr}^e_2$ is given by:

$$\text{Ctr}^e_1 \parallel \text{Ctr}^e_2 = \text{EnrichPatt}(\text{Ctr}^e_1 \parallel \text{Ctr}^e_2, \text{Con}^K_1 \cup \text{Con}^K_2)$$

Note that we use the $\text{EnrichPatt}$ function to uniformly and concisely define the extension of the rules. It can be used to define the appropriate clauses for the calculation of the $\text{Con}$ for each rule, based on the enriched operators presented in Section 2.3.2.

Regarding the communication rule, both the side conditions and the metadata calculation must be modified. The extension of the side condition concerns the validity of the connections. A pair of components can only be composed in a given pair of channels if both components are allowed to engage in this connection, according to the restrictions imposed by the relevant pattern. As explained, this is expressed by the $\text{Con}$ metadata, so if component $\text{Ctr}_1$ is being composed using channel $ch_1$ with component $\text{Ctr}_2$ using channel $ch_2$, they are both willing to engage in this composition if $(ch_1, ch_2) \in \text{Con}_1$ and $(ch_2, ch_1) \in \text{Con}_2$. Concerning metadata calculation, after such a composition, the resulting component is no longer allowed to engage in the channels just composed. Hence, its metadata is the union of the $\text{Con}$ structures of each component but excluding the pair of channels used for composition.

**Definition 4.4 (Pattern-based communication composition)** Let the constituent system contracts $\text{Ctr}^e_1 = \text{EnrichPatt}(\text{Ctr}^e_1, \text{Con}^K_1)$ and $\text{Ctr}^e_2 = \text{EnrichPatt}(\text{Ctr}^e_2, \text{Con}^K_2)$ be two pattern-based enriched ones, and $ic$ and $oc$ two channels, such that $(ic, oc) \in \text{Con}^K_1 \land (oc, ic) \in \text{Con}^K_2$, and $\text{Ctr}^e_1$ and $\text{Ctr}^e_2$ satisfy the condition of the enriched communication composition rule for $ic$ and $oc$. Then, the pattern-based enriched communication composition of $\text{Ctr}^e_1$ and $\text{Ctr}^e_2$ via $ic$ and $oc$ is defined as:

$$\text{Ctr}^e_1[ic \leftrightarrow oc]^e \text{Ctr}^e_2 \triangleq \text{EnrichPatt}(\text{Ctr}^e_1[ic \leftrightarrow oc]^e \text{Ctr}^e_2, (\text{Con}^K_1 \setminus \{(ic, oc)\}) \cup (\text{Con}^K_2 \setminus \{(oc, ic)\}))$$

The extension of the feedback composition rule is the most interesting case. It has its side conditions modified (rather than just extended like the previous rules) and a clause for the calculation of the new metadata element defined. The reason for this change is the need for removing the decoupled channels clause from the side condition of this rule. This modification makes this rule more expressive, as it can be applied to the construction of cyclic systems, which are compliant to a pattern, avoiding the use of the reflexive
rule for this purpose. This is the motivation for the use of behavioural patterns. For this rule, the side condition becomes the following. The rule does not oblige the channels to be decoupled anymore, but the constituents must be willing to connect using the two channels designated for composition, according to the Con metadata element (this part is very similar to the other rules). After the composition takes place, the pairs of channels representing the intent of connection must be removed from the Con of the resulting contract. Therefore, thanks to the removal of the decoupled channels condition, we can use this rule to build systems with cyclic communication topology, provided they comply to a pattern. Hence, there is no need for using the reflexive composition to build cyclic systems which are compliant to a given behavioural pattern. Henceforth, by the use of this extended strategy, one needs only to verify whether the connections are valid and whether protocols are compatible. Therefore, no global analysis is needed; instead, only local verifications are carried out.

Definition 4.5 (Pattern-based feedback composition)
Let the system contract $\text{Ctr}^{e^+} = \text{Enrich}_{\text{Patt}}(\text{Ctr}^e, \text{Con}^K)$ be a pattern-based enriched one, and $ic$ and $oc$ two channels, such that $\{(ic, oc), (oc, ic)\} \subseteq \text{Con}^K$, and $\text{Ctr}^e$ satisfies the conditions of the enriched feedback composition rule via $ic$ and $oc$, except for the decoupled channel condition (which might or might not be satisfied). Then, the pattern-based enriched feedback composition of $\text{Ctr}^{e^+}$ via $ic$ and $oc$ is defined as:

$$\text{Ctr}^{e^+}[ic \leftrightarrow oc]^{e^+} \equiv \text{Enrich}_{\text{Patt}}(\text{Ctr}^e[ic \leftrightarrow oc]^e, \text{Con}^K \setminus \{(ic, oc), (oc, ic)\})$$

This BRICK extension provides a fully local analysis strategy for systems that are compliant with a behavioural pattern. This impacts the practical applicability of the rules, making the analysis of a larger set of systems feasible. The verification of the side conditions of the reflexive rule, which was intractable for a large set of systems using the original enriched rules, named the set of cyclic topology systems, becomes tractable to those adherent to the behavioural patterns. This is further discussed later based on a practical experiment conducted.

In the sequel, we introduce three patterns that we integrate to BRICK: the resource allocation pattern, the client/server pattern and the async dynamic pattern. The first two have been introduced in [Mar98, Ros98] and detailed formalised and automated via refinement checking assertions in [ASW14], and the third one has been originally proposed, formalised and automatized in [AOS+14b]. Additionally to presenting the patterns, we instantiate for
each of them the \textit{INIT} predicate. This instantiation together with the framework presented, represents the integration of the patterns into \texttt{BRICK}. The presentation of the patterns also help the reader to get a full understanding of the pattern-based framework.

4.2 Client/Server pattern

The client/server is used for architectures where components interact in a client/server fashion, i.e. the server provides services that are requested by clients. In such systems a constituent might be in a state where it acts as a client, when it is ready to interact with other server constituents, and it can also reach a state where it behaves as a server, offering services to a different set of client constituents. As previously, before introducing the initial step of validation, we introduce the pattern itself. Some minor modifications also have to be performed in this pattern conditions, as the settings in \texttt{BRICK} are slightly different from the ones presented in [ASW14].

The elements of the client/server pattern describe elements of the client/server interaction. The first element is the function $\text{serverRequests}(\text{Ctr}_1, \text{Ctr}_2)$ that given two contracts $\text{Ctr}_1$ and $\text{Ctr}_2$ yields the set of events used by constituent $\text{Ctr}_1$ to interact with constituent $\text{Ctr}_2$ behaving as a server. These events represent services provided by the constituent $\text{Ctr}_1$. The function $\text{clientRequests}(\text{Ctr}_1, \text{Ctr}_2)$ yields the set of events that can be offered by component $\text{Ctr}_1$, when behaving as a client, to interact with $\text{Ctr}_2$. These represent the possible requests that the component can perform. The $\text{responses}(ev)$ function yields all the possible responses events for the request event $ev$. Finally, the $\text{ack}(ev)$ function gives the acknowledgement event for the event $ev$. We assume that the events given by $\text{serverRequests}$, $\text{clientRequests}$, $\text{responses}$ and $\text{ack}$ are mutually disjoints, in the sense that there cannot be an event that belongs to two of those sets. To sum up we have:

- $\text{serverRequests}(\text{Ctr}_1, \text{Ctr}_2)$ gives the events used by $\text{Ctr}_1$ to interact with $\text{Ctr}_2$, behaving as a server.
- $\text{clientRequests}(\text{Ctr}_1, \text{Ctr}_2)$ gives the events used by $\text{Ctr}_1$ to interact with $\text{Ctr}_2$, behaving as a client.
- $\text{responses}(ev)$ gives the response events of the event $ev$.
- $\text{ack}(ev)$ gives the acknowledgement event for event $ev$.

After describing the elements of interest of the pattern, we are able to introduce the restrictions, behavioural and structural, that define the pattern.
We begin with the introduction of the behavioural conditions. The first behavioural condition restricts the behaviour of a system when it is behaving as a server. When the constituent system is in such a state, which is defined by the offer of a server request event, it must offer all its server requests to its clients. This behaviour is described by the following specification, which, additionally, allows the process to behave arbitrarily when performing non server request. Observe also that this specification mandates that after each event performed an acknowledgement event must be issued. This is a simple manner of making these specifications buffer tolerant, as required by \textit{BRICK}. The function $A(Ctr)$ gives the alphabet of constituent $Ctr$.

\textbf{Definition 4.6 (Behavioural server requests specification)} Let $Ctr$ be a contract describing a constituent system, and $Ctrs$ the set of constituent contracts initially available for composition.

$$\text{ServerRequestsSpec}(Ctr) =$$
\begin{verbatim}
let sEvs = Union({serverRequests(Ctr,Ctr') | Ctr' <- Ctrs})
otherEvs = diff(A(Ctr),sEvs)
Server = ((|~| ev : otherEvs @ ev -> ack(ev) -> SKIP)
          |~| ([] ev : sEvs @ ev -> ack(ev) -> SKIP)) ; Server
within
  if not empty(otherEvs) then
    Server
  else
    RUN(sEvs)
\end{verbatim}

\textit{where:}

$$\text{RUN(evs) = [] ev : evs @ ev -> RUN(evs)}$$

Note that, in the definition of the process \text{ServerRequestsSpec}, we check whether the set of non server request events is empty, since the replicated internal choice operator is not defined for an empty set of elements. In the case of this set being empty, the server request restriction mandates the constituent to always offer all its events, since they are all server requests. This behaviour is described by process \text{RUN(sEvs)}. Otherwise, the atom must behave according to the \text{Server} process.

The pattern also restricts the behaviour of constituent systems concerning how requests and responses are performed. This restriction is not imposed on the complete behaviour of the constituents, but in a particular subset of it. As deadlocks can only happen in \textit{BRICK} constituents due to some kind of ill-interaction the behaviour that needs to be restricted to avoid this
problem is the one related with interaction between constituents. Hence, we use an abstraction function in the behaviour of the constituents to conceal the events that are not linked with synchronisation and therefore cannot participate on a deadlock. This abstraction function is given by $\text{Abs}(\text{Ctr})$, which is defined below as the projection of the behaviour of $\text{Ctr}$ ($\mathcal{B}_{\text{Ctr}}$) on the channels that are used for interaction between constituents, i.e. the channels in $\mathcal{C}$.

**Definition 4.7 (Abstraction function)** Let $\text{Ctr}$ be an I/O contract. The abstraction of the behaviour of $\text{Ctr}$ considering the behaviour related to interaction between constituents is given as follows.

$$\text{Abs}(\text{Ctr}) \triangleq \mathcal{B}_{\text{Ctr}} \upharpoonright (\bigcup_{c \in \mathcal{C}_{\text{Ctr}}} \{c \})$$

The abstraction of a constituent, conforming to the client/server pattern, must be initially offering request events. Once a request is performed, it can behave in several ways, according to some conditions. If the request performed demands no response, then the process must offer, again, some request event. If the request demands a response, then there are two cases to consider, when this performed request is a server one and when it is a client one. In the case of a server request, the process must answer this request with at least one of the possible responses. In the case of a client request, the process must be able to accept all expected responses. The specification of this behaviour is given by the following process, which also has to deal with the replicate internal choice undefinedness for empty sets. Note again the use of acknowledgements after events for buffer tolerance purposes.

**Definition 4.8 (Behavioural server responses specification)** Let $\text{Ctr}$ be a contract defining a constituent system, and $\text{Ctrs}$ the set of constituent contracts initially available for composition.

$$\text{RequestsResponsesSpec}(\text{Ctr}) =$$

let

$$\begin{align*}
\text{cEvs} &= \text{Union}\{(\text{clientRequests}(\text{Ctr},\text{Ctr}')) \mid \text{Ctr}' \leftarrow \text{Ctrs}\} \\
\text{sEvs} &= \text{Union}\{(\text{serverRequests}(\text{Ctr},\text{Ctr}')) \mid \text{Ctr}' \leftarrow \text{Ctrs}\}
\end{align*}$$

$\text{ClientRequestsResponsesSpec} =$

$$\begin{align*}
\mid \sim \mid \text{ev} : \text{cEvs} @ \text{ev} \rightarrow \text{ack}(\text{ev}) \rightarrow \\
&\quad \text{if} \text{empty}(\text{responses}(\text{ev})) \text{ then } \text{SKIP} \\
&\quad \text{else} (\mid \sim \mid \text{res} : \text{responses}(\text{ev}) @ \text{res} \rightarrow \\
&\quad \quad \text{ack}(\text{res}) \rightarrow \text{SKIP}))
\end{align*}$$

$\text{ServerRequestsResponsesSpec} =$

$$\begin{align*}
\mid \sim \mid \text{ev} : \text{sEvs} @ \text{ev} \rightarrow \text{ack}(\text{ev}) \rightarrow \\
\end{align*}$$
(if empty(responses(ev)) then SKIP 
else (|~| res : responses(ev) @ res -> 
ack(res) -> SKIP)))

C = ClientRequestsResponsesSpec;C
S = ServerRequestsResponsesSpec;S
CS = (ClientRequestsResponsesSpec 
|~| ServerRequestsResponsesSpec);CS
within
if empty(cEvs) and empty(sEvs) then STOP 
else 
if empty(cEvs) then S 
else 
if empty(sEvs) then C 
else CS

As mentioned, the conformance relation between the behaviour of a constituent and the aforementioned specifications is given by a refinement relation. In the case of conformance to the ServerRequestsSpec, the refinement relation used for this purpose is the one from the stable revivals model, represented by \([V=\text{(the presentation and discussion about the relevance of}}\) this model is carried out in Appendix A]. On the other hand, considering conformance to the RequestsResponsesSpec specification, the stable failure refinement relation is used for this purpose. Hence, the behavioural restriction of this pattern is given by the following BehaviourCS predicate. Observe that, as mentioned for the resource allocation pattern, we are interested in the abstract behaviour of a system, this is the reason why we also employ the abstraction function, presented in Section 4.3, to the behaviour of constituent systems.

**Definition 4.9 (Client/server behavioural restriction)** Let Ctrs be the contracts initially available for composition.

\[
\text{BehaviourCS}(\text{Ctrs}) \equiv \\
\text{Behaviour}(\text{Ctrs, ServerRequestsSpec, } [V=]) \land \\
\text{Behaviour}(\text{Ctrs, RequestResponsesSpec, } [F=])
\]

where
\[
\text{Behaviour}(S, Spec, \oplus) = \forall Ctr : S \bullet Spec(Ctr) \oplus Abs(Ctr)
\]

Turning to the structural restriction, similarly to the resource allocation pattern, the client/server’s structural restriction is composed by a conjunction of smaller clauses. Before introducing the proper conditions, we introduce the notions of a client and a server channels. These notions help to
simplify the understanding and manipulation of the events involved in the interaction between a client and a server. A server channel used by the constituent $Ctr_1$ to interact with constituent $Ctr_2$ is given by the function $ServerChannel(Ctr_1, Ctr_2)$. The $ServerChannel(Ctr_1, Ctr_2)$ is composed of the server requests, the possible responses for these requests and the acknowledgements for these requests and responses.

**Definition 4.10 (Server channel definitions)**

$$ServerChannel(Ctr_1, Ctr_2) \triangleq serverRequests(Ctr_1, Ctr_2) \cup serverResponses(Ctr_1, Ctr_2) \cup serverAcks(Ctr_1, Ctr_2)$$

where

$$serverResponses(Ctr_1, Ctr_2) \triangleq \bigcup_{ev:\text{serverRequests}(Ctr_1, Ctr_2)} responses(ev)$$

$$serverAcks(Ctr_1, Ctr_2) \triangleq \{ ack(ev) \mid ev \in serverRequests(Ctr_1, Ctr_2) \cup serverResponses(Ctr_1, Ctr_2) \}$$

In a very similar way, we define that a client channel used by the constituent $Ctr_1$ to interact with constituent $Ctr_2$ is given by $ClientChannel(Ctr_1, Ctr_2)$. The channel $ClientChannel(Ctr_1, Ctr_2)$ is composed of the client requests, the possible responses for these requests and the acknowledgements for these requests and responses.

**Definition 4.11 (Client channel definitions)**

$$ClientChannel(Ctr_1, Ctr_2) \triangleq clientRequests(Ctr_1, Ctr_2) \cup clientResponses(Ctr_1, Ctr_2) \cup clientAcks(Ctr_1, Ctr_2)$$

where

$$clientResponses(Ctr_1, Ctr_2) \triangleq \bigcup_{ev:\text{serverRequests}(Ctr_1, Ctr_2)} responses(ev)$$

$$clientAcks(Ctr_1, Ctr_2) \triangleq \{ ack(ev) \mid ev \in (serverRequests(Ctr_1, Ctr_2) \cup serverResponses(Ctr_1, Ctr_2)) \}$$

After defining these channels, we start introducing the clauses of the structural restriction of this pattern. The first one, which is given by the predicate $disjointChannels(Ctrs)$, ensures that the server channels and the client channels of a given constituent are disjoint. This means that the constituent
cannot use a given communication channel to behave as a server and as a
client, and hence we can apply different restrictions on these disjoint chan-
nels. Note that, we assume that a channel is a set of events, and different
channels must be disjoint.

\[
\text{disjointChannels}(\text{Ctrs}) \triangleq \\
\forall \text{Ctr} : \text{Ctrs} \bullet \{\text{ServerChannel}(\text{Ctr}, \text{Ctr}') \mid \text{Ctr}' \in \text{Ctrs}\} \\
\cap \{\text{ClientChannel}(\text{Ctr}, \text{Ctr}') \mid \text{Ctr}' \in \text{Ctrs}\} = \emptyset
\]

The \textit{controlledChannels}(\text{Ctrs}) predicate guarantees that the channels used
for interaction, i.e. the channels in the set \(\mathcal{C}_{\text{Ctr}}\), must be either a client or a
server channel. This restriction makes the channels used for interaction the
ones which have their behaviour restricted.

\[
\text{controlledChannels}(\text{Ctrs}) \triangleq \\
\forall \text{Ctr} : \text{Ctrs} \bullet \mathcal{C}_{\text{Ctr}} = \{\text{ServerChannel}(\text{Ctr}, \text{Ctr}') \mid \text{Ctr}' \in \text{Ctrs}\} \\
\cup \{\text{ClientChannel}(\text{Ctr}, \text{Ctr}') \mid \text{Ctr}' \in \text{Ctrs}\}
\]

The \textit{paired}(\text{Ctrs}) guarantees that every server channel has a client counter-
part and vice-versa. This ensures that the compositions can be made properly
as there exists a client channel to be connected to a server one.

\[
\text{paired}(\text{Ctrs}) \triangleq \\
\forall \text{Ctr}_1, \text{Ctr}_2 : \text{Ctrs} \bullet (\text{Ctr}_1, \text{Ctr}_2) \in \text{dom ServerChannel} \iff \\
(\text{Ctr}_2, \text{Ctr}_1) \in \text{dom ClientChannel}
\]

Also, the \textit{strictOrder} predicate guarantees that the transitive closure of the
\(>_{CS}\) relation, given next, \((>^*_{CS})\), is a strict order.

\[
\text{strictOrder}(>^*_{CS})
\]

where

\[
\text{Ctr}_1 >_{CS} \text{Ctr}_2 \triangleq (\text{Ctr}_1, \text{Ctr}_2) \in \text{dom ClientChannel}
\]

A final condition is required to make sure that the compositions are properly
made according to what is described in the \textit{ServerChannel} and \textit{ClientChannel}
structures. As BRICK keeps no track of what connections are performed or have been performed to create a particular constituent, we need to resort to the connection structure introduced in the resource allocation pattern (Section 4.3) to record this information. This structure keeps track of what channels have been connected so far. This condition imposes that the client channels must be connected to their server counterparts, and vice-versa. This condition inhibits the connection of incompatible channels, that might cause a deadlock.

\[
\text{connected}(\text{connections}) \triangleq \text{connections} \subseteq \\
\{(\text{ServerChannel}(\text{Ctr}_1, \text{Ctr}_2), \text{ClientChannel}(\text{Ctr}_2, \text{Ctr}_1)), \\
(\text{ClientChannel}(\text{Ctr}_2, \text{Ctr}_1), \text{ServerChannel}(\text{Ctr}_1, \text{Ctr}_2)) \\
| (\text{Ctr}_1, \text{Ctr}_2) \in \text{dom ServerChannel}\}
\]

After detailing these smaller clauses, we are able to define the client/server structural restriction. This restrictions is a mere conjunction of the presented predicates, as presented next.

**Definition 4.12 (Client/server structural restriction)** Let Ctrs be the set of initially available contracts for composition, connections be the set of pairs of channels connected through BRICK compositions.

\[
\text{StructureCS} \triangleq \quad \text{disjointChannels}(\text{Ctrs}) \land \text{controlledChannels}(\text{Ctrs}) \land \\
\text{paired}(\text{Ctrs}) \land \text{strictOrder}(>^*_{CS}) \land \\
\text{connected}(\text{connections})
\]

A network conforms to this predicate if the conjunction of the structural and behavioural restrictions is satisfied.

The reason why this pattern preserves deadlock freedom is similar to the one presented to the resource allocation pattern. If a system conforms to this pattern, then a cycle of ungranted requests cannot arise. For this claim, we also consider two cases, one when the design is complete, i.e. all intended connections have been made, and the other when it is not. When the design is complete, if a given path contains a system behaving as clients and other behaving as server a cycle cannot occur. The reason is that, mandatorily in such case, we have a client willing to interact with a server behaving system, which offers the events expected by the client, preventing an ungranted request of arising. Hence, the only possible way of a cycle occurring is a path consisting of only server behaving systems or of client behaving systems. In both cases, the constituents in the cycle must comply to the strict
order $>_\text{CS}$ (or to its dual $<_\text{CS}$), what prevents such cycles. In the case of an unfinished design, considering a path of ungranted requests, either one of the constituents is not connected to the next one, which trivially avoids the cycle, or all the constituents of the path have a connection to the following one, which implies that they must respect the strict order $>_\text{CS}$ (or to its dual $<_\text{CS}$), avoiding the cycle.

After presenting the pattern and formally claiming why it preserves deadlock freedom, we proceed with the presentation of the initial validation required by this pattern. Firstly, all the restrictions defined by the pattern are imposed on the initial contracts but the connected clause of the structural restriction, which is imposed in the connection to be made. Therefore, all these conditions must be verified on the initial validation step. Furthermore, we must add a clause for ensuring that the $\text{Con}$ element of the initial contracts have the appropriate information about the connections that the components shall make. This is achieved by the addition of the connectionObligations clause. This initial step is given by the following predicate.

**Definition 4.13 ($\text{INIT}_\text{CS}$)** Let $\text{Ctrs}$ be the set of constituent contracts initially available for composition, and connections the set of pairs of channels that have been connected through $\text{BRICK}$ compositions.

\[
\text{INIT}_\text{CS} \equiv \text{Behaviour}_\text{CS}(\text{Ctrs}) \land \text{disjointChannels}(\text{Ctrs}) \land \\
\text{controlledChannels}(\text{Ctrs}) \land \text{paired}(\text{Ctrs}) \land \text{strictOrdered}(>_\text{CS}) \land \\
\text{connectionObligations}(\text{Ctrs})
\]

where

\[
\text{connectionObligations}(\text{Ctrs}) \equiv \\
\forall \text{Ctr} : \text{Ctrs} \bullet \text{Con}_\text{Ctr} = \\
\{(\text{ServerChannel}(\text{Ctr}, \text{Ctr}'), \text{ClientChannel}(\text{Ctr}', \text{Ctr})) \\
| (\text{Ctr}, \text{Ctr}') \in \text{dom} \text{ServerChannel}\} \cup \\
\{(\text{ClientChannel}(\text{Ctr}, \text{Ctr}'), \text{ServerChannel}(\text{Ctr}', \text{Ctr})) \\
| (\text{Ctr}, \text{Ctr}') \in \text{dom} \text{ClientChannel}\}
\]

Our extension of $\text{BRICK}$ to accommodate the client/server pattern does prevent deadlocks of arising. The claim is very similar to the one presented for the resource allocation pattern. Note that the initial validation predicate guarantees that the behavioural and structural restrictions are met. The initial step also ensures that the $\text{Con}$ of initial contracts has only allowed connections. Turning to compositions, the only condition that could be violated by composing constituents is the connected clause. The reason is that, as mentioned, all the other constraints are imposed on the initial contracts. Nevertheless, the side condition of the rules, together with the new meta-
data calculation, ensures that the connections made are only the allowed ones. This means that, after the initial validation, the systems created with the composition rules proposed are all compliant to the client/server pattern. Hence, as all the systems created with these rules are pattern compliant and a compliant system is deadlock free, as claimed, the systems created with this framework are, as a consequence, deadlock free.

4.3 Resource allocation pattern

The resource allocation pattern can be used to model systems where constituents are competing for some shared resources. The version that we introduce here is a slight modification from the one presented in [ASW14]. The reason for these modifications is that the BRICK strategy makes some assumptions that are not valid in the aforementioned work. For instance, communication between systems in the mentioned work is done by event sharing, nevertheless, in BRICK, event sharing between systems is not allowed. Hence, we perform a few minor changes to cope with some of the differences in the BRICK settings. To begin with, we present the elements of interest of the pattern, which must be identified by the user of our strategy. These are:

- **Users**: the set of components of the systems that behave as users
- **Resources**: the set of components of the system that behave as resources
- **acq(CtrSource, CtrTarget)**: the event used for signaling acquisition between components CtrSource and CtrTarget
- **rel(CtrSource, CtrTarget)**: the event used for signaling release between components CtrSource and CtrTarget
- **ack(event)**: the event used to acknowledge an event of either acquisition or a release
- **resources(CtrUser)**: the sequence of resources representing the order in which the user CtrUser acquires them
- **users(CtrResource)**: the set of users that can acquire the resource identified by CtrResource

After detailing these elements, we are able to describe the restrictions imposed by the pattern. The behavioural restriction constrains the behaviour of the user and resource constituents.
The restriction on the behaviour of users imposes that the abstracted behaviour of such a contract must be a recursive sequential combination of acquisition of resources and then release of the acquired resources. Both acquisition and release of resources must respect a strict order given by the sequence \( \text{resources}(\text{Ctr}) \). Note that, after the acquisition and release of resources, an acknowledgement is expected. This is made so this specification is also buffer tolerant. This specification is presented below.

\[
\text{UserSpec}(\text{Ctr}) = \\
\text{let Acquire}(s) = \\
\quad \text{if } s \neq <> \text{ then} \\
\quad \quad \text{acquire}(\text{Ctr},\text{head}(s)) \rightarrow \\
\quad \quad \quad \text{ack}(\text{acquire}(\text{Ctr},\text{head}(s))) \rightarrow \\
\quad \quad \quad \text{Acquire}(\text{tail}(s)) \\
\quad \text{else SKIP} \\
\text{Release}(s) = \\
\quad \text{if } s \neq <> \text{ then} \\
\quad \quad \text{release}(\text{Ctr},\text{head}(s)) \rightarrow \\
\quad \quad \quad \text{ack}(\text{release}(\text{Ctr},\text{head}(s))) \rightarrow \\
\quad \quad \quad \text{Release}(\text{tail}(s)) \\
\quad \text{else SKIP} \\
\text{User}(s) = \\
\quad \text{Acquire}(s) ; \text{Release}(s) ; \text{User}(s) \\
\text{within} \\
\quad \text{User}(\text{resources}(\text{Ctr}))
\]

Regarding the resource components, their expected behaviour is as follows. They must initially be released, where they can be acquired by any of their users. Once acquired, only the user that acquired this given resource can release it. Note that, after the events of acquisition and release, the resource must perform an acknowledgement event, so as to make this process buffer tolerant. This specification is as follows.

\[
\text{ResourceSpec}(\text{Ctr}) = \\
\text{let CtrUsers = users(Ctr) \\
\text{Resource =} \\
\quad [ ] \text{CtrU : CtrUsers @} \\
\quad \quad \text{acquire}(\text{Ctr},\text{CtrU}) \rightarrow \text{ack}(\text{acquire}(\text{Ctr},\text{CtrU})) \rightarrow \\
\quad \quad \quad \text{release}(\text{Ctr},\text{CtrU}) \rightarrow \text{ack}(\text{release}(\text{Ctr},\text{CtrU})) \rightarrow \\
\quad \quad \quad \text{Resource} \\
\text{within} \\
\quad \text{Resource}
\]
These two processes represent the behavioural specification that should be met by the user and resource processes, respectively. The actual compliance restriction is guaranteed by a refinement relation, which represents a notion of conformance of the behaviour of the components to their specification. This conformance notion is given by the stable failures refinement relation $\sqsubseteq_F$. The predicate used to represent the conformance relation applied to each component is given as follows. Note that, even though this restriction constrains the global behaviour of the system preventing deadlock, it is not applied on the global behaviour of the system. Conversely, each individual component is restricted. This implies that, for behavioural validation, one does not need to make a global analysis of the system, but rather a local analysis of each component of the system, a generally simpler validation.

**Definition 4.14 (Resource allocation behavioural restriction)** Let $\text{Users}$ and $\text{Resources}$ be the sets of user and resource contracts, respectively.

$$\text{Behaviour}_{RA}(\text{Users, Resources}) \triangleq$$

$$\text{Behaviour}(\text{Users, UserSpec, } \sqsubseteq_F) \land$$

$$\text{Behaviour}(\text{Resources, ResourceSpec, } \sqsubseteq_F)$$

The pattern also imposes a structural restriction, which is given by a conjunction of simpler conditions. The first condition, $\text{partitions}(S, T, U)$, ensures that two sets $T$ and $U$ are the only two disjoint partitions of $S$. Using $\text{partitions}(\text{Ctrs, Users, Resources})$ below, we ensure that users and resources are two disjoint partitions of the initial components contracts $\text{Ctrs}$.

$$\text{partitions}(S, T, U) \triangleq S = (T \cup U) \land (T \cap U) = \emptyset$$

In the conditions that follow, we constrain the channels of the initial contracts. For the sake of simplicity, as $\text{BRICK}$ is a strategy dealing with interaction on the channel level, we introduce the concept of a resource allocation channel. The resource allocation channel used by contract $\text{Ctr}_1$ to interact with contract $\text{Ctr}_2$ is denoted by the function $\text{RACH}(\text{Ctr}_1, \text{Ctr}_2)$, which yields all events used for interactions between components $\text{Ctr}_1$ and $\text{Ctr}_2$: the events from contract $\text{Ctr}_1$, used for acquisition, release, and their respective acknowledgement events in the interaction with $\text{Ctr}_2$.

The next condition, $\text{controlledChannels}$, imposes that the channels used for communication by a given set of contracts $\text{Ctrs}$ must be composed by resource allocation channels.

$$\text{controlledChannels}(\text{Ctrs}) \triangleq$$

$$\forall \text{Ctr} \cdot C_{\text{Ctr}} = \{\text{RACH}(\text{Ctr}, \text{Ctr'}) | (\text{Ctr}, \text{Ctr'}) \in \text{dom RACH}\}$$
The following three conditions are not present in the original resource allocation pattern. They are needed due to the fact that in the original model connections are made by event sharing and, in the BRICK model, connections are made by a composition, which links different channels via buffers.

The condition \textit{paired} guarantees a correspondence between resources and users in a given set of contracts $Ctrs$. In what follows, $users(Ctr)$ yields the set of users of the contract $Ctr$ and $resources(Ctr)$ yields the sequence of resources used by $Ctr$.

\[ paired(Ctrs) \triangleq \forall Ctr_1, Ctr_2 : Ctrs \bullet Ctr_1 \in users(Ctr_2) \Leftrightarrow Ctr_2 \in ran resources(Ctr_1) \]

The next condition, \textit{consistent}, ensures the existence of a resource allocation channel for each interacting pair of contracts in the sets of users and resources given.

\[ consistent(Users, Resources) \equiv \forall Ctr : Users \bullet ran resources(Ctr) \subseteq Resources \land ran resources(Ctr) = \{ Ctr' \mid (Ctr, Ctr') \in dom RACh \} \land \forall Ctr : Resources \bullet users(Ctr) \subseteq Users \land users(Ctr) = \{ Ctr' \mid (Ctr, Ctr') \in dom RACh \} \]

Next, the \textit{connected} condition guarantees that the compositions must be made preserving the intent of connecting interacting users and resources. As the BRICK model provides no way of keep tracking of the connections made, we introduce the set \textit{connections} to represent these compositions. It is a set of pairs of channels that have been connected.

\[ connected(connections) \triangleq \]

\[ \text{connections} \subseteq \{(RACh(Ctr_1, Ctr_2), RACh(Ctr_2, Ctr_1)) \mid (Ctr_1, Ctr_2) \in dom RACh\} \]

Finally, the condition \textit{strictOrder} ensures that for all given users, their $resources(Ctr)$, the sequence of acquisition of resources by users, must respect a strict order on resources.

\[ strictOrdered(Users, S, \preceq) \triangleq \forall Ctr : Users \bullet resources(Ctr) \preceq S \]

The conjunction of all these conditions is used in the definition of the resource allocation structural restriction presented below.

93
Definition 4.15 (Resource allocation structural restriction) Let $\text{Ctrs}$ be the set of contracts initially available for composition, $\text{Users}$ and $\text{Resources}$ the sets of contracts describing the user and resource components as described. Additionally, let $\text{connections}$ be the set of pairs of channels connected, $S$ a strict order over the resources of the network and $\preceq$ a relation between a sequence and a strict order that holds when the sequence respects the strict order.

$$\text{StructureRA} \triangleq \text{partitions}(\text{Ctrs}, \text{Users}, \text{Resources}) \land \text{controlledAlphabet}(\text{Ctrs}) \land \text{paired}(\text{Ctrs}) \land \text{consistent}(\text{Users}, \text{Resources}) \land \text{connected}(\text{connections}) \land \text{strictOrdered}(\text{Users}, S, \preceq)$$

The compliance with the resource allocation pattern is given by the conformance to both behavioural and structural constraints, i.e. the set of components must satisfy both the $\text{StructureRA}$ and $\text{ BehaviourRA}$ predicates.

This pattern prevents deadlock for a simple reason. The only way of attaining a deadlocked state is by a cycle of ungranted requests between constituents, and if a system meets the aforementioned restrictions such a cycle is not possible. An ungranted request arises between constituents when they reach a state, where they individually offer shared events, but they cannot agree on these offered events, hence they cannot interact. For this deadlock freedom claim we consider two cases. The first case is when the design is complete, i.e. all the expected connections have been made. In such case, the only path of ungranted requests that can arise is the one of users trying to acquire an already acquired resource. This implies that the resources on this path respect the strict order $S$. This means that this path cannot be a cycle, since if it could it would violate the requirement of $S$ being a strict order. In the case of an incomplete design, i.e. there are still connections to be made, the only way of a path of ungranted requests occurring between a subset of the constituents of a system is when these components are all connected; if a component is not connected to another, obviously an ungranted request cannot occur. In such a case, the resources of this path also follow the $S$ order, making the cycle impossible for the same reason as before.

After introducing the pattern, we present the initial validation required by the pattern and the instantiation of the $\text{INIT}$ predicate for it. First of all, the behavioural restriction is imposed on the abstract behaviour of the initial contracts, hence this verification must be performed in the initial step. Considering the structural restriction, all but the $\text{connected}$ clause are conditions imposed on the initial contracts. Therefore, all these conditions must also be
verified on the initial validation step. Furthermore, we must add a clause for ensuring that the Con element of the initial contracts have the appropriate information about the connections that the components shall make. This is achieved by the connectionObligations clause. This initial step is given by the following predicate.

**Definition 4.16 (INIT\(_{RA}\))**  Let Ctrs be the set of contracts initially available for composition, Users and Resources the sets of contracts describing the user and resource components as described. Additionally, let connections be the set of pairs of channels connected, \(S\) a strict order over the resources of the network and \(\preceq\) a relation between a sequence and a strict order that holds when the sequence respect the strict order.

\[
\text{INIT}_{RA} \equiv \\
\text{Behaviour}_{RA}(\text{Users, Resources}) \\
\text{partitions}(\text{Ctrs, Users, Resources}) \land \text{controlledChannels}(\text{Ctrs}) \land \\
\text{paired}(\text{Ctrs}) \land \text{consistent}(\text{Users, Resources}) \land \\
\text{strictOrdered}(\text{Users, } S, \preceq) \land \text{connectionObligations}(\text{Ctrs})
\]

where

\[
\text{connectionObligations}(\text{Ctrs}) \equiv \\
\forall \text{Ctr} : \text{Ctrs} \bullet \text{Con}_{\text{Ctr}} = \\
\{ (\text{RACH}(\text{Ctr, Ctr}'), \text{RACH}(\text{Ctr}', \text{Ctr})) \mid (\text{Ctr, Ctr}') \in \text{dom RACH} \}
\]

Our extension to **BRICK** to accommodate the resource allocation pattern indeed prevents deadlock of arising. Note that the initial validation predicate guarantees that the behavioural and structural restrictions are met. The initial step also ensures that the Con of initial contracts have only intended connections. Considering composition, the only condition that could be violated is the connected clause, since all the other constraints are imposed on the initial contracts. Nevertheless, the side condition of the rules, together with the new metadata calculation, ensures that the connections made are only the intended ones. This means that, after the initial validation, the systems created with the proposed composition rules are all compliant to the resource allocation pattern. Hence, as all the systems created with these rules are pattern compliant, they are deadlock free.

### 4.4 Async Dynamic pattern

The async dynamic pattern can be applied to networks with systems with two types of entities: the participants and the transport layer. The participants of the system do not interact directly with each other, but exchange
messages via the transport layer. A participant recursively sends messages to all its peer participants and receives messages from them. Both sending and receiving must follow an order. Furthermore, participants can turn on and off at any time. The transport layer, composed of a set of transport entities, provides communication point-to-point between participants of the network. It also has the ability to identify whether participants are on or off.

First, we identify the elements of interest of the pattern. As for the resource allocation, we partition the initial contracts into two sets. The set of participants, given by $\text{Participants}$, and the set of transport entities, given by $\text{TransportEntities}$. The $\text{source}(\text{Ctr})$ and $\text{target}(\text{Ctr})$ functions yield, respectively, the source participant of the transport entity $\text{Ctr}$, i.e. the participant feeding this transport entity with data, and the target participant, i.e. the participant which receives data from this transport entity. As mentioned, a transport entity provides a unidirectional point-to-point communication means between these two participants. The function $\text{transport}(\text{Ctr}_1, \text{Ctr}_2)$ gives the transport entity relaying data from participant $\text{Ctr}_1$ to participant $\text{Ctr}_2$. The $\text{sequence}(\text{Ctr})$ function yields a sequence of participants. This sequence states the order in which the participant $\text{Ctr}$ interacts with its peers via transport entities. The function $\text{onEvent}(\text{Ctr}_1, \text{Ctr}_2)$ gives the event used to signal to a transport entity that a participant has turned on. Hence, $(\text{Ctr}_1, \text{Ctr}_2)$ must be a pair composed of a transport entity and a participant. As $\text{BRICK}$ requires two different channels to make this interaction possible (one for each constituent), we convey that the function $\text{onEvent}(\text{Ctr}_1, \text{Ctr}_2)$ gives the event belonging to $\text{Ctr}_1$, whereas $\text{onEvent}(\text{Ctr}_2, \text{Ctr}_1)$ gives the event belonging to $\text{Ctr}_2$. This convention is used in all other functions yielding events or channels. Similarly, the function $\text{offEvent}(\text{Ctr}_1, \text{Ctr}_2)$ yield the event used for signalling to a transport entity that a participant has turned off. The function $\text{sendCh}(\text{Ctr}_1, \text{Ctr}_2)$ gives the channel used for a participant to send data, and the function $\text{sendAck}(\text{Ctr}_1, \text{Ctr}_2)$ yields the event used to acknowledge this sending process. The function $\text{receiveReq}(\text{Ctr}_1, \text{Ctr}_2)$ gives the event used to request the reception of data. The $\text{receiverOff}(\text{Ctr}_1, \text{Ctr}_2)$ function gives the event used to signal to a transport entity its receiver is off. $\text{receiveData}(\text{Ctr}_1, \text{Ctr}_2)$ yields the channel used for the reception of data by a participant, and $\text{timeoutEvent}(\text{Ctr}_1, \text{Ctr}_2)$ gives the event used for signalling that a timeout has occurred. We assume that the events given by these different functions are all mutually disjoint, what means that an event cannot be used, for instance, for signalling a turning on and turning off.

To sum up the elements of interest of this pattern are:

- $\text{Participants}$ the set of contracts describing participants
• *TransportEntities* the set of contracts describing transport entities

• *source*(Ctr) and *target*(Ctr) yield the source and the target participants of transport entity Ctr, respectively.

• *transport*(Ctr₁, Ctr₂) yields the transport entity contract having Ctr₁ as its source participant and Ctr₂ as its target participant.

• *sequence*(Ctr) yields the sequence in which the participant Ctr communicates with its peer participants.

• *onEvent*(Ctr₁, Ctr₂) gives the event used by Ctr₁ to communicate with Ctr₂ that a participant has turned on.

• *offEvent*(Ctr₁, Ctr₂) yields the event used by the constituent Ctr₁ to communicate with Ctr₂ that a participant has turned off.

• *sendCh*(Ctr₁, Ctr₂) yields the channel used by Ctr₁ in the transmission of data from a participant to a transport entity.

• *sendAck*(Ctr₁, Ctr₂) yields the event used by Ctr₁ to acknowledge the transmission of data from a participant to a transport entity.

• *receiveReq*(Ctr₁, Ctr₂) yields the event used by Ctr₁ for a participant to request the reception of data from a transport entity.

• *receiveData*(Ctr₁, Ctr₂) yields the channel used by Ctr₁ to receive data from a transport entity to a participant.

• *receiverOff*(Ctr₁, Ctr₂) yields the event used by Ctr₁ to signal to a transport entity that its receiver participant is off.

• *timeoutEvent*(Ctr₁, Ctr₂) gives the event used by Ctr₁ to communicate with Ctr₂ that a timeout has occurred.

The behavioural restriction of this pattern constrains the behaviour of both transport entities and participants. A transport entity is entitled to receive data from its sender participant and to pass this data on to its receiver participant. This communication is unidirectional from the sender to the receiver. It also detects whether its sender is switched on or off. When the sender is switched off, it offers a timeout to the target participant, so it will not wait indefinitely for data. The behavioural specification of a transport entity and a more detailed explanation of its behaviour is given next.

**Definition 4.17 (Transport entity specification)** Let Ctr be a transport entity contract.

\[ \text{TransportSpec}(Ctr) = \]
let

CtrR = target(Ctr)
On = offEvent(Ctr,CtrR) -> Off
[]
  ([] ev : sendCh(Ctr,CtrR) @ ev ->
    sendAck(Ctr,CtrR) -> OnF)
[] (receiverOff(Ctr,CtrR) -> On)
OnF = offEvent(Ctr,CtrR) -> Off
[] ([] ev : sendCh(Ctr,CtrR) @
    ev -> sendAck(Ctr,CtrR) -> OnF)
[] (receiveReq(Ctr,CtrR) ->
    [] d : receiveData(Ctr,CtrR) @ d -> On)
[] (receiverOff(Ctr,CtrR) -> OnF)
Off = onEvent(Ctr,CtrR) -> On
[] (receiveReq(Ctr,CtrR) ->
    timeoutEvent(Ctr,CtrR) -> Off)
[] (receiverOff(Ctr,CtrR) -> Off)
within

Off

The processes On, OnF (the “F” stands for full) and Off specify the expected behaviour of a transport entity when its sender is detected as turned on and no data is available, when its sender is detected as turned on and data is available, and when its sender is detected as turned off, respectively. The transport entity initially behaves as Off, in which case it accepts a receive request, from the receiver participant, or a turning on event, from the sender participant. If a receive request is performed, it performs a timeout event, since the source participant is off and will not provide any data. When it accepts a turning on event, it behaves as the process On. In this state, the transport entity is on and empty: it can receive data from the sender participant or detect a switching off event from it. In the latter case, the entity behaves as Off again. However, if it receives data, the transport entity acknowledges and stores this data, and starts behaving as OnF. In the OnF state, the transport entity can receive new data from its sender participant, in which case the new data is also acknowledged and overwrites the data previously stored. Additionally, it can also transmit the data stored to its receiver participant. It offers a receive request to, then, relay data to the receiver participant throught the receive data channel. After this the transport entity behaves as On again. Finally, it can also detect whether its sender participant has turned off, in which case it behaves as the process Off. This mechanism of data transmission with acknowledgements is required to
make this process buffer tolerant.

The participants of the network contain the business logic of the system. They have a dynamic behavioural feature that allows them to turn on and off, and a functional behaviour that involves data exchange and any business related function. The behavioural specification of a conforming participant is given in the next definition.

**Definition 4.18 (Participant specification)**  Let $C$ be a participant. The participant CSP specification is:

$\text{ParticipantSpec}(C) =$

```
let
  $s = \text{sequence}(C)$
within
  $\text{OnDetect}(C,s);$  
  $\text{SendReceive}(C,s);$  
  $\text{OffDetect}(C,s);$  
  $\text{ParticipantSpec}(C)$
```

where

$\text{SendReceive}(C,s) = \text{Send}(C,s)$

$\text{OffDetect}(C,s) =$

```
if $s \neq <>$ then
  $\text{offEvent}(C,\text{transport}(C,\text{head}(s))) \rightarrow$
    $\text{OffDetect}(C,\text{tail}(s))$
else
  SKIP
```

$\text{OnDetect}(C,s) =$

```
if $s \neq <>$ then
  $\text{onEvent}(C,\text{transport}(C,\text{head}(s))) \rightarrow$
    $\text{OnDetect}(C,\text{tail}(s))$
else
  SKIP
```

$\text{Send}(C,s) =$

```
if $s \neq <>$ then
  $(\text{sendCh}(C,\text{transport}(C,\text{head}(s)))?d \rightarrow$
    $\text{sendAck}(C,\text{transport}(C,\text{head}(s))) \rightarrow$
      $\text{Send}(C,\text{tail}(s)))$
```
D2.4.4 - Comp. Anal. and Des. of CML Models (Public)

|~| SKIP
else
   Receive(Ctr,sequence(Ctr))

Receive(Ctr, s) =
   if s != <> then
      (receiveReq(Ctr, transport(head(s), Ctr)) ->
         ((receiveData(Ctr, transport(head(s), Ctr))?d ->
           Receive(Ctr, tail(s)))
         [] timeoutEvent(Ctr, transport(head(s), Ctr)) ->
           Receive(Ctr, tail(s))))
   |~|
      (receiverOff(Ctr, transport(head(s), Ctr)) -> SKIP)
else
   Send(Ctr, sequence(Ctr))

A participant first behaves as the process \texttt{OnDetect}, which sends a signal to inform that it is turned on to each transport entity to which it acts as a sender. This mechanism abstracts the ability of the transport layer to detect participant status. The \texttt{s} parameter, which is a sequence, gives the order in which the participant interacts with its transport entities. After turning on, it acts recursively, first behaving as a sender (\texttt{Send}) and then as a receiver (\texttt{Receive}). When behaving as a sender, it sends messages to all transport entities that have this participant as sender, following the order recorded in \texttt{s}. The sending process consists of sending data and then receiving an acknowledgement. When acting as a receiver, in the same way, it interacts with the transport entities that have it as a receiver, also following the order stated in \texttt{s}: it first requests data to a transport entity, to then receive incoming data or a timeout signal, which indicates that the sender associated with the transport entity is turned off. Similarly to the transport entity specification, this request-response style of transmitting data between transport entity and participant is needed to make this process buffer tolerant. Note that, when behaving as the send or the receive process, it can internally decide to behave as \texttt{SKIP}, which represents the ability of a participant to fail. After choosing \texttt{SKIP}, the participant behaves as the \texttt{OffDetect} process, which describes the behaviour of a failing participant.

In order to check whether a transport entity or a participant conforms to the corresponding specified behaviour, we use the refinement relation in the stable failures model of CSP. The behavioural restriction imposed by this pattern is given as follows.
Definition 4.19 (Dynamic async behavioural restriction) Let Ctrs be the set of constituent contracts initially available for composition, and the sets TransportEntities and Participants be transport entity and participant contracts of the system, respectively. Then we define:

\[
\text{BehaviourAD}(\text{TransportEntities, Participants}) \equiv \\
\text{Behaviour}(\text{TransportEntities, TransportSpec, } [F=]) \land \\
\text{Behaviour}(\text{Participants, ParticipantSpec, } [F=])
\]

Before introducing the structural restriction of this pattern, we introduce the notions of a sender channel and a receiver channel, which helps in the definitions and understanding of the interactions between participants and transport entities. A sender channel consists of the set of events used in the interaction between a transport entity and its sender participant. This channel is given by function \(\text{SenderChannel}(Ctr_1, Ctr_2)\). Again, note that this interaction requires two channels to be made, one for the transport entity and another for the sender participant, and the first argument tells to which contract the channel belongs. The sender channel is composed by the send channel, by the turning on and off events, and by the send acknowledgement event, as defined next.

Definition 4.20 (\(\text{SenderChannel}(Ctr_1, Ctr_2)\)) Let \(Ctr_1\) and \(Ctr_2\) be two constituent contracts.

\[
\text{SenderChannel}(Ctr_1, Ctr_2) \equiv \\
\text{sendCh}(Ctr_1, Ctr_2) \\
\cup \{\text{onEvent}(Ctr_1, Ctr_2), \text{offEvent}(Ctr_1, Ctr_2), \text{sendAck}(Ctr_1, Ctr_2)\}
\]

A receiver channel consists of the events used for the interaction between a transport entity and its receiver participant. This channel is given by the function \(\text{ReceiverChannel}(Ctr_1, Ctr_2)\) and is composed by the event used for requesting data, by the data reception channel and by the timeout event.

Definition 4.21 (\(\text{ReceiverChannel}(Ctr_1, Ctr_2)\)) Let \(Ctr_1\) and \(Ctr_2\) be two constituent contracts.

\[
\text{ReceiverChannel}(Ctr_1, Ctr_2) \equiv \\
\text{receiveData}(Ctr_1, Ctr_2) \\
\cup \{\text{receiveReq}(Ctr_1, Ctr_2), \text{timeoutEvent}(Ctr_1, Ctr_2)\}
\]
After introducing these channels, we present the structural restriction of the async dynamic pattern. Similarly to the other patterns, the structural restriction is a conjunction of smaller conditions. The first restriction imposes that transport entities and participants must be disjoint and total partitions of the system. This means that each of the constituents of the system must either be a participant or a transport entity.

\[
\text{partition}(\text{Ctrls}, \text{Participants}, \text{TransportEntities}) \equiv \\
\text{Participants} \cap \text{TransportEntities} = \emptyset \land \\
\text{Participants} \cup \text{TransportEntities} = \text{Ctrls}
\]

The \textit{consistent} predicate ensures that the function defining the communication channels, i.e. the \textit{SenderChannel} and \textit{ReceiverChannel}, are defined only for the pairs of contracts that are designed to be connected. This means that the pairs of contracts belonging to the domains of these functions are exactly the pairs of components that are allowed to connect. A participant is allowed to be connected to the transport entities that links it with the participants in the sequence given by the function \textit{sequence}. For each participant in \textit{sequence}, there must be a sender connection, made by channel \textit{SenderChannel}, and a receiver connection, made via \textit{ReceiverChannel}.

\[
\text{consistent}(\text{Participants}, \text{TransportEntities}) \equiv \\
\text{participantsConsistent}(\text{Participants}) \land \\
\text{transportEntitiesConsistent}(\text{TransportEntities})
\]

where

\[
\text{participantsConsistent}(\text{Participants}) \equiv \\
\forall Ctr : \text{Participants} \bullet \\
\text{ran sequence}(Ctr) \subseteq \text{Participants} \land \\
\text{ran sequence}(Ctr) = \{ Ctr' \mid (Ctr, Ctr') \in \text{dom transport} \} \land \\
\text{ran sequence}(Ctr) = \\
\{ Ctr' \mid (Ctr, \text{transport}(Ctr, Ctr')) \in \text{SenderChannel} \} \land \\
\text{ran sequence}(Ctr) = \\
\{ Ctr' \mid (Ctr, \text{transport}(Ctr', Ctr)) \in \text{ReceiverChannel} \}
\]

\[
\text{transportEntitiesConsistent}(\text{TransportEntities}) \equiv \\
\forall Ctr : \text{TransportEntities} \bullet \\
\text{source}(Ctr) \in \text{Participants} \land \text{target}(Ctr) \in \text{Participants} \land \\
\{ \text{source}(Ctr) \} = \{ Ctr' \mid (Ctr, Ctr') \in \text{SenderChannel} \} \land \\
\{ \text{target}(Ctr) \} = \{ Ctr' \mid (Ctr, Ctr') \in \text{ReceiverChannel} \}
\]
The \textit{disjointChannels} condition imposes that the communication channels used for sending and the ones used for receiving must be disjoint. This ensures that a sender channel cannot be used as a receiver one, and vice-versa.

\[
\text{disjointChannels}(Ctrs) \triangleq \\
\forall Ctr : Ctrs \bullet \\
\{ \text{SenderChannel}(Ctr, Ctr') \} \\
\cup \\
\{ \text{ReceiverChannel}(Ctr, Ctr') \} = \emptyset
\]

The \textit{controlledChannels} predicate is satisfied if the channel set $C_{Ctr}$, i.e. the set of channels available for composition, is composed by only sender and receiver channels. As we restrict the behaviour of the these channels, (and these are the channels used for interaction) it ensures that the behaviour related to the interaction between constituents of the network is restricted.

\[
\text{controlledChannels}(Ctrs) \triangleq \\
\forall Ctr : Ctrs \bullet \\
C_{Ctr} = \{ \text{SenderChannel}(Ctr, Ctr') \} \\
\cup \\
\{ \text{ReceiverChannel}(Ctr, Ctr') \}
\]

As mentioned a participant has a cyclic behaviour interacting with its peers through transport entities. These interactions per cycle are guided by the sequence of contracts given by the function \textit{sequence}. The \textit{validOrders} conditions ensures that in a given cycle a participant must interact at most once with another participant.

\[
\text{validOrders}(Participants) \triangleq \\
\forall Ctr : Participants \bullet \text{injective}(\text{sequence}(Ctr))
\]

The \textit{paired} restriction guarantees that if a participant, say $Ctr$, wants to communicate with another one, say $Ctr'$, which is given by the fact that $Ctr'$
belongs to \(\text{sequence}(\text{Ctr})\), then \(\text{Ctr}'\) must also be willing to communicate with \(\text{Ctr}\). This ensures that the participants interact in a two-way fashion, where interacting participants send and receive data from one another.

\[
\text{paired}(\text{Participants}) \triangleq \\
\forall \text{Ctr}_1, \text{Ctr}_2 : \text{Participants} \bullet (\text{Ctr}_1 \in \text{ran sequence}(\text{Ctr}_2) \iff \text{Ctr}_2 \in \text{ran sequence}(\text{Ctr}_1))
\]

The \text{connected} restriction is similar to the one present in the two previous pattern. It ensures that the connections made throughout compositions are exactly the ones intended by the architect of the initial contracts. As \text{BRICK} does not keep track of the connections made, we again make an appeal to the \text{connections} variable, introduced for this purpose. Since we restrict the communication channels (\text{SenderChannel} and \text{ReceiverChannel}) to be the channels to be connected and since we employ the conventions that a channel \text{Channel}(\text{Ctr}, \text{Ctr}') must be connected to \text{Channel}(\text{Ctr}', \text{Ctr})}, the allowed connections are the set of pairs satisfying these conditions. Note that the subset-equal relation guarantees that even incomplete designs, in the sense that it has not been fully connected yet, satisfies this condition.

\[
\text{connected}(\text{connections}) \triangleq \\
\text{connection} \subseteq \\
\{(\text{SenderChannel}(\text{Ctr}, \text{Ctr}'), \text{SenderChannel}(\text{Ctr}', \text{Ctr})) \\
\mid (\text{Ctr}, \text{Ctr}') \in \text{dom SenderChannel}\} \cup \\
\{(\text{ReceiverChannel}(\text{Ctr}, \text{Ctr}'), \text{ReceiverChannel}(\text{Ctr}', \text{Ctr})) \\
\mid (\text{Ctr}, \text{Ctr}') \in \text{dom ReceiverChannel}\}
\]

Finally, the structural restriction of this pattern is given by the conjunction of these clauses, as presented next.

**Definition 4.22 (Async Dynamic network)** Let \(\text{Ctrs}\) be the set of initially available constituent contracts, \(\text{Participants}\) be the set of participants, \(\text{TransportEntities}\) be a set of transport entities, and \(\text{connections}\) the variable that keeps track of the connections made so far.

\[
\text{StructuralAD}(\text{Ctrs}, \text{Participants}, \text{TransportEntities}) \triangleq \\
\text{partition}(\text{Ctrs}, \text{Participants}, \text{TransportEntities}) \land \\
\text{disjointChannels}(\text{Ctrs}) \land \text{controlledChannels}(\text{Ctrs}) \land \\
\text{validOrders}(\text{Participants}) \land \text{paired}(\text{Participants}) \land \\
\text{consistent}(\text{Ctrs}) \land \text{connected}(\text{connections})
\]
The conformance of a network to this pattern is given by a conjunction of the restrictions presented.

Note that, given the restrictions imposed, we should consider the cases when the design is completed and when it is not, as discussed for the other patterns. First, considering a complete design, the only possible way of a path of ungranted requests arising is by participants waiting for receiving data from transport entities that are in turn waiting for data from their sender participants. In such a case, there is a strict order on the participants in this path: a participant must have started broadcasting more recently than the following participant in this path. Note that this is a strict order, since it is transitive and irreflexive. In the case of an incomplete design, similarly to the previous patterns, either all the constituents in a path of ungranted requests have been connected, in which case the participants in this path also obey to the start broadcasting strict order, or a constituent is not connected, which makes an ungrated request impossible, and, as a consequence, a cycle. The reason for this impossibility is that, trivially, an ungrated request cannot arise between unconnected components, since they do not communicate.

After presenting the pattern and formally claiming why it ensures deadlock freedom, we proceed with the presentation of the initial validation required. First of all, the behavioural and structural restrictions are imposed on initial contracts, hence these verification must be performed in the initial step. Considering the structural restriction, the connected clause deserves particular attention, as it must be ensured throughout compositions, since it constrains the connections to be made. Hence, we add a clause for ensuring that the Con element of the initial contracts have the appropriate information about the connections which the components shall make. This is achieved by the connectionObligations clause. As later discussed, this condition together with the ones imposed by the rules guarantees conformance to the pattern. This initial step is given by the following predicate.

**Definition 4.23 (INIT\textsubscript{AD})** Let Ctrs be the set of initially available constituent contracts, Participants be the set of participants and TransportEntities
be the set of transport entities.

\[ \text{INIT}_{AD} \triangleq \text{Behaviour}_{AD}(\text{Participants}, \text{TransportEntities}) \land \text{partitions}(\text{Ctrls}, \text{Participants}, \text{TransportEntities}) \land \text{disjointChannels}(\text{Ctrls}) \land \text{controlledChannels}(\text{Ctrls}) \land \text{paired}(\text{Participants}) \land \text{validOrders}(\text{Participants}) \land \text{connectionObligations}(\text{Ctrls}) \]

where

\[ \text{connectionObligations}(\text{Ctrls}) \triangleq \forall \text{Ctr} : \text{Ctrls} \bullet \text{Con}_{\text{Ctr}} = \{ (\text{SenderChannel}(\text{Ctr}, \text{Ctr}'), \text{SenderChannel}(\text{Ctr}', \text{Ctr})) \mid (\text{Ctr}, \text{Ctr}') \in \text{dom SenderChannel} \} \cup \{ (\text{ReceiverChannel}(\text{Ctr}, \text{Ctr}'), \text{ReceiverChannel}(\text{Ctr}', \text{Ctr})) \mid (\text{Ctr}, \text{Ctr}') \in \text{dom ReceiverChannel} \} \]

Our extension of \textit{BRICK} to include the async dynamic pattern prevents deadlocks. The claim is very similar to the one presented for the other patterns. Note that the initial validation predicate guarantees that the behavioural and structural restrictions are met. The initial step also ensures that the \text{Con} of initial contracts has only allowed connections. Considering composition, the only condition that could be violated by the composition of contracts is the \textit{connected} clause, since all the other constraints are imposed solely on the initial contracts. Nevertheless, the side condition of the rules, together with the new metadata calculation, ensures that the connections made are only the intended ones. This means that, after the initial validation, the systems created with the proposed composition rules are all compliant to the async dynamic pattern. Hence, as claimed before, all such systems are deadlock free.

4.5 Experimenting on the case studies

After proposing this extended version of \textit{BRICK}, we demonstrate the application of this strategy to the two case studies introduced in Section 3. We take the asymmetric dining philosophers and the leadership election SoS examples and demonstrate that these case studies are deadlock free using our extended strategy. Furthermore, we conduct two experiments to demonstrate the improvement in efficiency of this version when compared with the \textit{BRICK} original version. An observation deserving mention is that all the experiments have been conducted using FDR 2.94 and \textit{CSP}M, the machine readable version of the \textit{CSP} notation used by FDR. Nevertheless, as already
mentioned, the adaptation of these CSP case studies to CML imposes no theoretical issue, since a connection has been proposed between CSP and CML, which provides a systematised manner of lifting these case studies from CSP to CML.

### 4.5.1 Asymmetric dining philosophers

In order to manipulate the elements of the contract and the elements of the pattern in a more convenient way, we named the contracts using a pair containing a constant stating the type of the contract, either a fork or a philosopher, and a natural number to differentiate philosophers and forks among themselves. Hence, a contract named $FK.0$, is the contract describing a fork with identifier 0 and $PH.0$ is the contract describing the philosopher with identifier 0. This naming strategy is used because it simplifies the definition of the elements by the use of pattern matching.

In order to use the strategy, the first task to be done is the identification of the elements of the pattern. To begin with, we identify the set of users and the set of resources. In our case study, $Users$ is the set of component contracts describing the philosophers and $Resources$ is the set of component contracts describing the forks. In $CSP_M$, functions can be defined using pattern matching, where a pattern can be a combination of a value and a variable, where values must be matched and variables are used for binding.

We define the functions for yielding the acquire, the release, and the acknowledgment events, together with the resources of a user and the users of a resource using pattern matching as follows.

```plaintext
acquire(FK.idf, PH.idp) = fk.idf.idp.picksup
acquire(PH.idp, FK.idf) = pfk.idf.idp.picksup
release(FK.idf, PH.idp) = fk.idf.idp.putsdown
release(PH.idp, FK.idf) = pfk.idf.idp.putsdown
ack(fk.idf.idp.picksup) = fk.idf.idp.putsack
ack(pfk.idf.idp.picksup) = pfk.idf.idp.putsack
ack(fk.idf.idp.putsdown) = fk.idf.idp.putsack
ack(pfk.idf.idp.putsdown) = pfk.idf.idp.putsack

resources(PH.id) = if id == MAX then <FK.0, FK.MAX>
                    else <FK.id, FK.(next(id))>
resources(Other) = <>
```
After defining the elements of the pattern, we are able to start using our \textsc{brick} pattern-based extension. As we plan to build the same dining philosophers built in Section 3.1 we define the initial contracts exactly as presented there. The $\text{Con}$, which is the new piece of information that needs to be defined on top of it, is defined as follows.

$$
\text{Con}(\text{FK.id}) = \{ (\text{RACh}(\text{FK.id}, \text{PH.idp}), \text{RACh}(\text{PH.idp}, \text{FK.id})) \\
| \text{idp} \leftarrow \{ \text{id}, \text{prev(id)} \} \}
$$

$$
\text{Con}(\text{PH.id}) = \{ (\text{RACh}(\text{PH.id}, \text{FK.idf}), \text{RACh}(\text{FK.idf}, \text{PH.id})) \\
| \text{idf} \leftarrow \{ \text{id}, \text{next(id)} \} \}
$$

These are set comprehension expressions that define all the connections expected by the fork and philosopher constituents, respectively.

Based on these definitions, we are able to apply the steps of the strategy. In our first step, we verify that the initial contracts together with the elements of interest of the pattern satisfy the initial step $\text{INIT}_R$. That is the case, since all the clauses are met. In this execution of the experiment we removed the life channel of the set of channels of the philosophers, so the $\text{controlledChannels}$ clause is satisfied. This provokes no changes in our final system, since we do not remove it from the behaviour of the philosophers and these channels are never used for composition.

After checking that the initial contracts are valid and can be used by the pattern-based enriched rules for composition, we are able to build our system using the same strategy adopted in Section 3.1 but using the pattern based version of the rules. We proceed with the exact same process of composition, except that instead of using the reflexive rule in the last composition, we use our feedback rule. The reason why we are now able to use this rule, (actually is a modified version of this rule), is that it does not have the decoupled channel condition as the original version has, hence it can be used at this point, what does not happen in the original version. For conciseness, as the compositions are basically the same as those presented in Section 3.1 we do not present them in details here. Using the pattern-based enriched rules, we were able to create a deadlock free model of the asymmetric dining philosophers, without using global analysis.

In order to validate the efficiency of our pattern based optimisation, we con-
Figure 11: Results of experiments for the dining philosophers

<table>
<thead>
<tr>
<th>N</th>
<th># of Comps</th>
<th>RA</th>
<th>Metadata</th>
<th>Global</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>10</td>
<td>0.085</td>
<td>0.081</td>
<td>0.008</td>
</tr>
<tr>
<td>4</td>
<td>14</td>
<td>1.009</td>
<td>1.066</td>
<td>0.031</td>
</tr>
<tr>
<td>5</td>
<td>18</td>
<td>1.04</td>
<td>15.007</td>
<td>4.004</td>
</tr>
<tr>
<td>6</td>
<td>22</td>
<td>1.062</td>
<td>297.074</td>
<td>118.037</td>
</tr>
<tr>
<td>7</td>
<td>26</td>
<td>1.088</td>
<td>4521</td>
<td>2029.077</td>
</tr>
<tr>
<td>10</td>
<td>38</td>
<td>2.066</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>100</td>
<td>398</td>
<td>29.008</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>1000</td>
<td>3998</td>
<td>298.07</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>10000</td>
<td>39998</td>
<td>3952.063</td>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>

ducted a practical experiment using the asymmetric dining philosopher case study. For this experiment, we created several instances, varying the N (number of philosophers or forks) of the asymmetric dining philosophers. Nevertheless, we employed the same strategy presented in Section 3.1 using the pattern-based rules. First, we interleave the philosophers, then the forks. After we compose these two resulting components using the communication rule. Next, we compose the rest of the communication channels using the feedback rule. Note that, differently from what happens to the original compositions, we are able to connect the last channels using also the feedback rule.

The verification of side conditions involving set operations were not taken into account because they can be verified by SAT solvers at a rather insignificant cost if compared to behavioural analysis. For this reason, we measured only the time taken for behavioural verification. The * means that FDR failed to provide an answer since it consumed the entire memory of the dedicated server. The results of this practical experimentation are depicted in Figure 11.

These results demonstrate that our optimisation using local analysis considerably improves the efficiency of our strategy. Using our optimisation, BRICK, which needed exponential time for the verification of side conditions, because of one single condition (buffering self-injection compatibility) of the reflexive composition, comes to require only a linear time, with the integration of a behavioural pattern into the original enriched strategy. This allows the strategy to be used in the development of a larger set of systems in a practical manner. With this optimisation, we were able to analyse a system with 10,000 philosophers and 10,000 forks in a little more than one hour. Previously, within this time frame the strategy was only able to analyse an instance with up to 7 philosophers and 7 forks. As a matter of
fact, using our optimisation of \textit{BRICK}, we were able to analyse a network
with 20,000 processes. Hence, we were able to go further than one of the
main strategies used for deadlock verification in the CSP world, the Dead-
lock Checker \cite{Mar96}, a tool designed for the specific purpose of deadlock
verification, which is bounded to verify systems with up to 10,000 processes.
Observe that these results can be transposed to the CML world. The goal
of this strategy is to promote local analysis in the spite of global analysis,
regardless of the technology employed. Hence, this should make deadlock
analysis possible to a set of systems which were, practically, impossible to
check, independently of the language or tool support used.

4.5.2 Leadership election algorithm

For the leadership election algorithm, we use the async dynamic pattern to
build the core of the system, i.e. the sub-system composed of nodes and bus
cells. After this core being designed using the pattern-based strategy, the
memory could be introduced using the communication rule of the original
strategy in an efficient way, since this rule is compositional. Hence, as the
main goal of this section is to introduce the application of this pattern and
evaluate its efficiency, for this case study, we are only concerned with the
construction of this core, leaving the memory out of the scope of this case
study.

As for the dining philosophers, we used a particular nomenclature to help
identifying the elements of our system. The nodes are named \textit{NODE}.\textit{i}, where
\textit{i} is a natural number, and the bus cells are identified by \textit{BUS}.\textit{s}.\textit{t}, where \textit{s}
and \textit{t} are two natural numbers, identifying the source and the target nodes
of this bus cell, respectively. Moreover, note that some of the definitions
given next take into account a global constant \textit{N}, which is a natural number
representing the number of nodes minus one. The reason why we need this
is to make our definitions parametric, so they remain appropriate regardless
of the instance of our model, i.e. of the natural \textit{N}. First, in order to use
the strategy, one has to identify the elements of the pattern. We begin by
identifying the participants and transport entities of our system. In this
case, the participants are the nodes, and the transport entities are the bus
cells.

\textbf{Participant} = \{\textit{NODE}.\textit{n} | \textit{n} <- \{0..\textit{N}\}\}

\textbf{TransportEntities} = \{\textit{BUS}.\textit{s}.\textit{t} | \textit{s} <- \{0..\textit{N}\}, \textit{t} <- \{0..\textit{N}\}, \textit{s} != \textit{t}\}
The sequence in which a participant interacts with its peers, given by the sequence function, is the sequence composed by all other nodes of the network from NODE.0 to NODE.N.

\[ \text{sequence}(\text{NODE.id}) = \langle \text{NODE.n} \mid n \leftarrow \langle 0..N \rangle, n \neq \text{id} \rangle \]

Moving on, we present the functions giving the receive request event, the channel for receiving data, and the event for signalling a timeout. Note that we use pattern matching to describe these functions. In CSP\(_M\), the pattern NODE._ matches all nodes, and BUS.idS.idT, matches all bus cells and makes the appropriate binding of the naturals to idS and idT.

\[
\begin{align*}
\text{receiveReq}(\text{NODE._}, \text{BUS.idS.idT}) &= \text{receiver.idS.idT.out.req} \\
\text{receiveReq}(\text{BUS.idS.idT}, \text{NODE._}) &= \text{receiver_bus.idS.idT.in.req} \\
\text{receiverOff}(\text{NODE._}, \text{BUS.idS.idT}) &= \text{receiver.idS.idT.out.isOff} \\
\text{receiverOff}(\text{BUS.idS.idT}, \text{NODE._}) &= \text{receiver_bus.idS.idT.in.isOff} \\
\text{receiveData}(\text{NODE._}, \text{BUS.idS.idT}) &= \{\text{receiver.idS.idT.in.x} \mid x \leftarrow \text{TPACKS} \} \\
\text{receiveData}(\text{BUS.idS.idT}, \text{NODE._}) &= \{|\text{receiver_bus.idS.idT.out}| \} \\
\text{timeoutEvent}(\text{NODE._}, \text{BUS.idS.idT}) &= \text{receiver.idS.idT.in.timeout} \\
\text{timeoutEvent}(\text{BUS.idS.idT}, \text{NODE._}) &= \text{receiver_bus.idS.idT.out.timeout} \\
\end{align*}
\]

Moving further along, we come to the description of the functions giving: the channel used for sending data, and the event used for acknowledgment of the data being sent.

\[
\begin{align*}
\text{sendCh}(\text{NODE._}, \text{BUS.idS.idT}) &= \{|\text{sender.idS.idT.out.pack}| \} \\
\text{sendCh}(\text{BUS.idS.idT}, \text{NODE._}) &= \{|\text{sender_bus.idS.idT.in.pack}| \} \\
\text{sendAck}(\text{NODE._}, \text{BUS.idS.idT}) &= \text{sender.idS.idT.in.ack} \\
\text{sendAck}(\text{BUS.idS.idT}, \text{NODE._}) &= \text{sender_bus.idS.idT.out.ack} \\
\end{align*}
\]

The functions giving the events used for signalling that a participant has turned on, and that a participant has turned off are given next.

\[
\begin{align*}
\text{onEvent}(\text{NODE._}, \text{BUS.idS.idT}) &= \text{sender.idS.idT.out.isOn} \\
\text{onEvent}(\text{BUS.idS.idT}, \text{NODE._}) &= \text{sender_bus.idS.idT.in.isOn} \\
\text{offEvent}(\text{NODE._}, \text{BUS.idS.idT}) &= \text{sender.idS.idT.out.isOff} \\
\text{offEvent}(\text{BUS.idS.idT}, \text{NODE._}) &= \text{sender_bus.idS.idT.in.isOff} \\
\end{align*}
\]
Finally, we introduce the functions \textit{transport}, \textit{source} and \textit{target}.

\begin{align*}
\text{transport}(\text{NODE.idS}, \text{NODE.idT}) &= \text{BUS.idS.idT} \\
\text{source}(\text{BUS.idS.idR}) &= \text{NODE.idS} \\
\text{target}(\text{BUS.idS.idR}) &= \text{NODE.idR}
\end{align*}

After introducing the elements of the pattern, we introduce the \textit{Con} structure for nodes and bus cells. This structure is used in the initial validation of the system.

\begin{align*}
\text{Con}(\text{NODE.i}) &= \\
&= \{(\text{SenderChannel}(\text{NODE.i,transport}(\text{NODE.i,Ctr})), \\
&\quad \text{SenderChannel}(\text{transport}(\text{NODE.i,Ctr}), \text{NODE.i})), \\
&\quad (\text{ReceiverChannel}(\text{NODE.i,transport}(\text{Ctr,NODE.i})), \\
&\quad \text{ReceiverChannel}(\text{transport}(\text{Ctr,NODE.i}), \text{NODE.i})), \\
&\quad | \text{Ctr} <- \text{sequence}(\text{NODE.i})\}\}
\end{align*}

\begin{align*}
\text{Con}(\text{BUS.s.t}) &= \\
&= \{(\text{SenderChannel}(\text{BUS.s.t,NODE.s})), \\
&\quad \text{SenderChannel}(\text{NODE.s,BUS.s.t})), \\
&\quad (\text{ReceiverChannel}(\text{BUS.s.t,NODE.t})), \\
&\quad \text{ReceiverChannel}(\text{NODE.t,BUS.s.t})))\}
\end{align*}

After introducing this, we present the strategy used for the composition of the nodes and bus cells. In our first step, we verify that the initial contracts together with the elements of interest of the pattern satisfy the initial step \textit{INIT} \textit{AD}. After checking that the initial contracts are valid and can be used by the pattern-based enriched rules for composition, we are able to build our system using the same strategy adopted in Section 3.2, but using the pattern based version of the rules. We proceed with the exact same process of composition, except that in the steps where we used the original reflexive rule, we now use our pattern-based feedback rule, with its relaxed condition that does not require channels to be decoupled. Again, for conciseness, as the compositions are basically the same as those presented in Section 3.2, we do not present them in details here. Using the pattern-based enriched rules, we were able to create a deadlock free model of the core of the leadership election SoS, without using global analysis.

For the experiments, we considered our pattern-based version of \textit{BRICK}, the original \textit{BRICK} strategy, and a global analysis strategy, which consists of verifying whether the final system is deadlock free. For this final case, we used the deadlock freedom assertion of FDR. Once again, the verification of
side conditions involving set operations were not taken into account because they can be verified by SAT solvers at a rather insignificant cost if compared to behavioural analysis. Hence, the time presented concerns only behavioural checking. As we have a parametrised model, we are able to easily generate a set of instances, varying the parameter $N$. We varied $N$ up to 32, since this represents the maximal number of nodes that this $B&O$ system might have. The * means that FDR failed to provide an answer since it consumed the entire memory of the dedicated server. The results of this practical experimentation are depicted in Figure 12.

The original $BRICK$ strategy is heavily affected by the fact that the design of this core requires several reflexive compositions. Note the rapid growth in the time taken to verify the system; for this case, this strategy goes from a total time of about 3 seconds to verify a 2-nodes configuration to more than 6 hours to verify a 3-nodes configuration. For the global case, note again how it is affected by the explosion in the state space; this strategy is able to verify efficiently a 2 and 3-nodes configuration, but fails to verify whether a 5-nodes configuration is deadlock free. It is worth mentioning that FDR takes more than 2 hours to fail for this configuration, which means that even if more memory were available, the amount of time taken would represent an exponential growth in the time for verification. Considering these two strategies, even if we analyse the growth in time taken for verification with the number of compositions per configuration, which growths quadratically, this increase in time is substantially faster, indicating the inability of these strategies to handle this example. Finally, the time for the verification of the system using our pattern-based version grows steadily with the growth in the number of compositions. It is true that the rise in the verification time is faster than the growth in the number of compositions (see chart in Figure 12), which might represent an inability of our strategy to handle larger cases, and a different scenario when compared with the philosophers
case study, where our strategy behaved linearly. This comes from the fact that as we increase the $N$, we also make the individual nodes more complex. With the increase of $N$, individual nodes have to augment their behaviour to communicate with more nodes, as each node in the system must communicate with the others. Hence, instead of only increasing the number of processes, this growth also implies the increase in the complexity and, consequently, the time for the individual behavioural verification of the nodes. Nevertheless, our strategy is able to verify a 32-nodes configuration in about 9 hours and 15 minutes, whereas the other strategies were not able to cope with more than 3-nodes.

4.6 Automating the extended strategy

After introducing the application of this strategy to these case studies, we discuss what could be automated in this extended strategy, so that a tool could be aimed to aid the end-user in the application of BRICK. Firstly, in this pattern-based strategy, we do not intend to guess the pattern that could be applied to a given system. Hence, the choice of the pattern is not meant to be automated, this being the responsibility of the specifier of the system. Second, the elements of the pattern are to be identified by the end user as well. In this point some automation could be provided, though; this automation could incur in a highly inefficient step due to the numerous possibilities of entities of the system that could characterise valid elements of a pattern. Also, the design of a system based on a pattern should be guided exactly by these elements, which is an additional evidence that this task is the responsibility of the specifier. After this, the initial step could be completely automated. As it verifies conditions of decidable theories, and these conditions can be either locally analysed or statically checked, this entire step should not be very costly. Finally, the choice of application of the rules and the components to be composed are the responsibility of the end-user. Note that we do not have any means of knowing if the intended system has been accomplished so as to automate this part. Nevertheless, once a rule and the operand contracts are chosen, the side conditions and the calculation of the resulting component, could be fully automated. Again, this automation should not be a problem from an efficiency point of view, as the side conditions of these rules are local and the calculation of the resulting component is rather simple.
5 Preserving Livelock

Our previous deliverable [OSA13] proposed a compositional analysis technique that considered only deadlock freedom. In this section, we present a strategy for local livelock analysis, based on constructive rules similar to those that ensure deadlock freedom, but with additional side conditions.

First, in Section 5.1 we motivate our efforts for proposing such strategy. Next, in Section 5.2 we present a general discussion on livelock. We provide an exercise that takes into consideration the possible causes of livelock within a specification. In Section 5.3 we formalise the notion of component contract livelock for BRIC. This new notion is needed because BRIC compositions do not hide the composed channels from the behaviour; they are just removed from the interface avoiding further compositions on them. This gives us a grey-box style abstraction. Hence, in Section 5.3 we consider a black box composition style in which we formalise the notion of component contract livelock for BRIC. Following this formalisation, due to hiding, compositions may lead to contract divergences, even considering that the constituents to be composed are livelock free.

The new definition of component contract livelock fosters a further discussion on the possible causes of such a livelock in BRICK, which is presented in Section 5.4. The ideas presented in this section are the basis of the approach for local livelock analysis proposed and formalised in Section 5.5, where we present the conditions in which the BRICK compositions also guarantee livelock freedom.

The benefits of using our approach are demonstrated with the application of our strategy to the dining philosophers case study discussed in the Section 5.6.

5.1 Compositional Livelock Analysis

In the Deliverable D24.1 [OSA13], we proposed a set of composition rules in order to develop deadlock-free systems in BRIC [Ram11]. However, only this verification is not enough to avoid all problems when constituent systems are being integrated, especially in concurrent systems. In some cases, a system may be free of deadlock and yet it makes no progress in executing its tasks. This situation is referred to as livelock, which is regarded as even worse than deadlock.
Besides the deadlock analysis, we provided some conditions to also ensure livelock-free compositions in \textit{BRIC}. In this approach, the only way to preserve livelock is to force wrappings to perform safe hidings. However, this makes the verification very restricted, since the approach only considers divergent free processes.

Ensuring livelock-free systems is a challenging task and it is highly desirable to guarantee that systems are reliable. In fact, there are few approaches which deal with this problem. Furthermore, as far as we are aware, they have not provided a trustworthy composition strategy to ensure that a system is livelock-free.

The traditional approach to prove that a concurrent system is livelock-free performs a global analysis in order to verify that a livelock state cannot be reached \cite{Ros10}. However, this verification is costly and can become unfeasible, considering the number of components of a large system. This strategy is fully automatised by FDR \cite{For05}.

One alternative to the traditional approach is to perform a local livelock analysis, which allows us to check only some parts of a system. Moreover, this strategy requires less calculation, and, consequently, it is possible to reduce time and effort in this verification.

In order to achieve this goal, we investigated additional conditions that guarantees that the original composition rules ensure absence of livelock. Furthermore, we provide a definition of livelock at the level of \textit{BRIC} component contracts, in which we analyse the behaviour $B$ of the component, but we hide the communication channels that are not in the component’s set of visible channels $C$. The reason for this is because in \textit{BRIC} the channels used in the compositions are removed from the set of visible channels $C$, and, consequently, they cannot be used in future compositions; however, they are not hidden in the component’s behaviour.

As explained in the introduction, the results presented here were developed based on the CSP process algebra \cite{Ros98} due to the fact that the CML model-checker was still under development. This was intentionally done so as we can simply lift our results to CML using the theoretical link from CSP to CML for processes and refinement presented in the previous deliverable \cite{OSA13}.

The remainder of this section is structured as follows. The next section gives more details about how livelock can be reached in concurrent systems. Afterwards, we present in details our approach to build livelock-free systems in \textit{BRIC}. Finally, Section \ref{case_study} presents the case study that was implemented.
to evaluate our approach.

5.2 A General Investigation on Livelock

In order to ensure livelock-free systems, it is essential to investigate how this problem can be detected. As the dynamic behaviour in our approach is represented as a CSP process, we must be aware as livelock may arise in the context of CSP.

According to Roscoe [Ros10], the possibility of writing down a divergent CSP process may arise by using hiding, which converts visible actions into internal ones, since it may create an infinite loop of internal events. As an example, let us consider the process $P = a \rightarrow P \{a\}$, in which $P$ indefinitely performs the internal event $a$, configuring a divergence.

Besides this simple livelock process, there exists more complex cases in which we compose processes using some of the CSP operators. Here, we focus on two operators: interleaving and linked parallel, since they link exactly with the concepts of our composition rules.

The interleaving operator allows processes to run independently. On the other hand, the linked parallel is a combination of parallel, hiding and renaming. More details about these operators are described in the sequel.

In order to illustrate how livelock may be introduced by using these operators, let us consider two simple livelock-free CSP processes:

$$P = a \rightarrow P$$
$$Q = b \rightarrow Q$$

If we combine these processes in interleaving, $P || Q$, the resulting process is also livelock-free. This is due the fact that, as they do not share any event, their communication with the environment is not changed. However, we may change the process $P$ to perform only internal actions:

$$P' = (a \rightarrow P') \{a\}$$

In this case, we get a divergent behaviour. The composition $P' || Q$ has a livelock, since the original process $P'$ already has a livelock. Therefore, the resulting process of an interleaving composition directly depends on the original processes. In other words, livelock will only be introduced if one of them also introduces livelock before the composition.
Another way of combination that we are concerned is by using the linked parallel. In this case, \( P[a \leftrightarrow b]Q \), represents a process in which runs \( P \) and \( Q \) in parallel, synchronises them on \( a \) and \( b \), and, finally, hides the synchronised events. In our example, the result of the linked parallel composition, \( P[a \leftrightarrow b]Q \), is not livelock-free. This is due the fact that, after hiding the actions that are synchronised, \( a \) and \( b \), the resulting process performs recursive internal actions, causing divergence.

Obviously, there are cases in which the linked parallel composition does not introduce livelock. For instance, let us consider the following livelock-free processes.

\[
P'' = a \rightarrow c \rightarrow P'' \\
Q'' = b \rightarrow Q''
\]

The composition \( P''[a \leftrightarrow b]Q'' \) is livelock-free because after hiding the synchronised actions, the resulting process still communicates with the environment on channel \( c \) at every cycle; hence, it does not have a recursive internal behaviour.

Based on these cases, roughly speaking, we can conclude that one way to perform livelock-free linked parallel compositions is to ensure that at least one original channel continues to be offered to the environment after hiding the synchronised events.

Besides these two examples, there are other particular cases that must be carefully analysed. These cases are related to the processes that are composed by channels that communicate values. By way of illustration, let us consider the following processes:

\[
P''' = a.2 \rightarrow a.3 \rightarrow a.1 \rightarrow P''' \\
Q''' = b.1 \rightarrow Q''' \\
\square b.2 \rightarrow Q''' \\
\square b.3 \rightarrow Q'''
\]

The parallel composition \( P'''[a \leftrightarrow b]Q''' \) is not livelock-free. In summary, this is because \( P''' \) and \( Q''' \) are able to internally communicate on \( a \) and \( b \) respectively in an infinite loop: every cycle on \( P''' \) corresponds to three cycles on \( Q''' \) (one for each branch).

On the other hand, there are cases when the synchronisation is not feasible, and, consequently, no infinite loop takes place. In this context, livelock is not introduced. For instance, let us consider the following process:
\[ P''' = a.2 \rightarrow a.1 \rightarrow a.5 \rightarrow a.3 \rightarrow P''\]

The composition \( P''''[a \leftrightarrow b]Q''\) is livelock-free. This situation is caused because the processes must synchronise on channels \( a \) and \( b \), but at some point of their communication history they are not able to synchronise. More specifically, there is no event in \( Q''\) that communicates the value 5.

These examples presented here do not cover all CSP processes, but they give us insights on how livelock can be introduced in different situations, which are relevant for our livelock analysis in \textit{BRIC}.

### 5.3 Livelock in \textit{BRIC}

In the Deliverable D24.1 [OSA13], we presented a strategy to preserve deadlock freedom in \textit{BRIC}. The composition rules presented do not introduce divergence in the resulting process behaviour \( B \) because they do not hide the composed channels, but simply remove them from the communication channel set \( C \), preventing further compositions on these channels.

However, a notion of livelock at the \textit{BRIC} level was not provided. We propose a definition of Livelock-free Component Contract that considers \textit{BRIC} components as black-boxes. The main difference between our definition and that of simply looking at the component’s behaviour \( B \) is that we consider only the visibility of the channels that are in the component’s set of channels \( C \), which are eligible for future compositions. As a result, the \textit{BRIC} may introduce livelock at the component contract level. For us, a component contract \( Ctr \) is livelock-free if after hiding all channels that are not in the set of communication channels \( C_{Ctr} \) in the behaviour \( B_{Ctr} \), no divergence is introduced. Otherwise, a component is not livelock-free.

**Definition 5.1 (Livelock-free Component Contract)** Let \( Ctr \) be a component contract defined as \( \langle B, R, I, C \rangle \). Then \( Ctr \) is a livelock-free component contract if, and only if, divergences\( (B_{Ctr} \mid C_{Ctr}) = \emptyset \).

In our definition of livelock-free component contract, we hide all channels that cannot be used for composition because they will never interact with another component, and, consequently, will only provide services for that specific component. In this way, the other components do not need to know about the internal services provided by them, increasing the abstraction level of the system.
5.4 Livelock Analysis

Before formalising the local livelock analysis in BRIC, we investigate the circumstances in which livelock can be introduced in the original approach based on the discussions presented in Section 5.2. As we discussed before, Ramos [Ram11] proposed four compositions rules to build trustworthy component systems: interleave, communication, feedback and reflexive compositions. In what follows, we briefly discuss the conditions in which they may introduce a contract livelock.

The interleave composition $P[|||]Q$ is the most basic form of composition: two components are composed with no communication between them. They directly communicate with the environment as before, with no interference from each other. Since the behaviour of the BRIC components is divergence-free processes (Definition 2.6), we do not need to perform any additional verification to ensure contract livelock freedom. This is because, as we discussed earlier, the CSP interleaving composition only introduces livelock if initially one of the processes diverge. The remaining rules, however, require a more careful investigation.

The communication composition $P[ic \leftrightarrow oc]Q$ attaches two components connecting two channels, one from each component. The resulting behaviour is defined as the synchronisation of the components (P and Q) on the communication channels. The remaining events occur independently. This behaviour is similar to CSP’s linked parallel composition, which was discussed in Section 5.2. As a result, a contract livelock may be introduced because we also hide the synchronised events since they are removed from the channel set of the resulting component.

In order to illustrate how a contract livelock can be introduced in such a BRIC composition, let us consider two component contracts $Ctr_1$ and $Ctr_2$, with the following I/O processes as their behaviours.

\[B_1 = ic \rightarrow B_1\]
\[B_2 = oc \rightarrow B_2\]

Let us also consider that both contracts satisfy the conditions needed for a communication composition on $ic$ and $oc$, respectively. The communication composition $Ctr_1[ic \leftrightarrow oc]Ctr_2$ removes $ic$ and $oc$ from set of channels $C$ of the resulting contract. Hence, the resulting contract indefinitely communicates on internal channels, causing a contract livelock.

Some others examples in which linked parallel compositions may cause livelock were presented in Section 5.2. All of them can be considered as possible
contract behaviours and, for the same reasons discussed, they can also introduce livelock.

Besides this binary composition, livelock can be reached by the two remaining rules, feedback and reflexive. These rules represent unary compositions, which assemble two channels of the same component. Although these rules have distinct conditions to be performed, livelock is reached in both compositions for the same reasons as those of a communication composition, since feedback and reflexive compositions synchronise the channels used in the composition.

Therefore, in order to ensure contract livelock-free compositions in BRIC, we have to extend the set of side conditions of the composition rules, adding new conditions in which we can also ensure livelock freedom when components are being composed. In this context, a distinguished feature of our strategy is that we perform a local livelock analysis, rather than the traditionally used global analysis. Furthermore, we can provide strategies to predict this problem before the composition is performed. In what follows, we describe our approach to build livelock-free systems in BRIC.

5.5 Local Livelock Analysis

A livelock-free contract never performs an infinite sequence of internal events without communicating with its environment. Hence, reasoning about divergences requires reasoning about infinite behaviours. Therefore, in our approach, the first step consists of identifying what are the infinite behaviours of a given component contract. We define the function $IP(B_{Ctr})$ that returns the traces that leads the process to a recursion. Notice that, in our case, as the behaviours are I/O processes, they only have infinite traces. So, this function will return all the possible paths the process may perform before it recurs.

**Definition 5.2 (Set of Interaction Patterns)** Let $P$ be a CSP process. The maximal (potentially infinite) set of interaction patterns is defined as:

$$IP(P) = \{ s : traces(P) \mid P \equiv_{FD} (P/s) \}$$

where $P/s$ (pronounced $P$ after $s$) represents the behaviour of $P$ after the trace $s$ is performed, for any $s \in traces(P)$.

The definition above gives a possibly infinite set of traces that leads the process to its initial state.
By way of illustration, let consider the following CSP processes:

\[ \begin{align*}
P &= a \rightarrow b \rightarrow P \\
    &\quad \Box n.1 \rightarrow n.2 \rightarrow P \\
Q &= c \rightarrow d \rightarrow Q \\
    &\quad \Box s.1 \rightarrow Q
\end{align*} \]

In this case, their interaction patterns would be defined as follows.

\[ \begin{align*}
IP(P) &= \{ (a, b), (n.1, n.2), (a, b, a, b), \ldots \} \\
IP(Q) &= \{ (c, d), (s.1), (c, d, c, d), \ldots \}.
\end{align*} \]

Unfortunately, the set of interaction patterns is potentially infinite as we can see in the example above. Nevertheless, our strategy needs the set of minimal interaction patterns that considers only the paths that lead to the first recursion. The definition of this sets uses the auxiliary function \( S^\circ \), which, given a set of traces \( S \) (in our case, interaction patterns), returns the concatenation closure on \( S \), i.e. the set of all sequences you can make by taking any subset of traces from the original \( S \) and concatenating them together (possibly with repetitions).

\[ S^\circ = \{ t : \Sigma^* \mid (\exists ss : \text{seq}(\Sigma^*) \mid \text{ran}(ss) \subseteq S \land t = \sqcup ss \} \]

Using this function, we define the set of minimal interaction patterns of a process \( P \), \( \text{MIN}_IP(P) \), as a set such that \( \text{MIN}_IP(P)^\circ = IP(P)^\circ \), i.e., it is able to generate the same traces that can be generated by its set of interaction patterns \( IP(P) \). However, it contains only the minimal interaction patterns. As a consequence, it is a subset of any other subset of interaction patterns \( S \) of \( IP(P) \), such that \( S^\circ = IP(P)^\circ \).

**Definition 5.3 (Set of Minimal Interaction Patterns)** Let \( P \) be a CSP process. The set of minimal interaction patterns of \( P \), \( \text{MIN}_IP(P) \), is such that:

\[ \text{MIN}_IP(P)^\circ = IP(P)^\circ \land \forall S : \mathbb{P}(\Sigma^*) \mid S \subseteq IP(P) \land S^\circ = IP(P)^\circ \cdot \text{MIN}_IP(P) \subseteq S \]

In our example, we obtain the following minimal interaction patterns (henceforth called interaction pattern).

\[ \begin{align*}
\text{MIN}_IP(P) &= \{ (a, b), (n.1, n.2) \} \\
\text{MIN}_IP(Q) &= \{ (c, d), (s.1) \}.
\end{align*} \]
Our implementation to calculate the interaction patterns refines this denotational definition. It, however, restricts the classes of processes that are accepted in our strategy. Namely, we consider only processes that recur to themselves and can be normalised to a single process. For instance, a process $P$ defined as

\[
P = a \rightarrow Q \\
Q = b \rightarrow P
\]

can be normalised to a single process

\[
P' = a \rightarrow b \rightarrow P'
\]

whilst a process $Q$ defined as

\[
Q = a \rightarrow R \\
R = b \rightarrow R
\]

cannot be normalised to a single process and is not accepted by our strategy.

After normalisation, we use the following syntax based implementation to calculate the interaction patterns of a process. In what follows, we use $P$ to denote different processes names and $\overline{P}$ to denote its behaviour. Furthermore, we use $W_1$ and $W_2$ to denote any CSP behaviour.

**Definition 5.4 (Interaction Patterns Calculation)** Let $P$ be a CSP process defined as:

\[
P = \overline{P}
\]
The interaction patterns of $P$ is given by $IP_P(P)$, where:

- $IP_P(P) = \{()\}$
- $IP_P(SKIP) = IP_P(STOP) = \{\}$
- $IP_P(c \rightarrow W_1) = \{(c) \cap x \mid x \in IP_P(W_1)\}$
- $IP_P(c!e \rightarrow W_1) = IP_P(c.e \rightarrow W_1) = \{(c.e) \cap x \mid x \in IP_P(W_1)\}$
- $IP_P(c?x \rightarrow W_1) = \{(c.e) \cap x \mid e \in T(x) \land x \in IP_P(W_1)\}$
- $IP_P(W_1 \oplus W_2) = IP_P(W_1) \cup IP_P(W_2)$
- $IP_P(g & W_1) = IP_P(W_1)$
- $IP_P(\text{if } g \text{ then } W_1 \text{ else } W_2) = IP_P(W_1) \cup IP_P(W_2)$
- $IP_P(W_1 \triangleright W_2) = \left\{ \left. \text{front}(t_1) \cap t_2 \mid t_1 \in \text{traces}(W_1) \land \text{last}(t_1) = \checkmark \right\} \land t_2 \in IP_P(W_2)\right\}$
- $IP_P(P \setminus a) = \{x - \{a\} \mid x \in IP_P(P)\}$

where $s - A$ removes all events in $A$ from the sequence $s$.

Operationally, the calculation of these interaction patterns is based on the calculation of traces proposed by Roscoe [Ros98].

Our strategy does not allow the basic components to have themselves parallel behaviour, i.e., the CSP processes that can be used to describe a contract’s behaviour, $B$, are strictly sequential (possibly with choices). Parallelism may be achieved by composing contracts using the composition rules. Although not necessary from the theoretical point of view, this restriction allows us to provide an optimisation in the tool that implements the strategy described later in this section.

Furthermore, for some operators, we have a specific approach to capture the interaction patterns. One example is the calculation of the interaction patterns of a guarded processes $IP_P(g \& W_1)$, in which we simply ignore the guard $g$ and take the interaction patterns $IP_P(W_1)$ of $W_1$ as the result. As a consequence, our approach may find false negatives because we take interaction patterns which could not be feasible depending on the evaluation of $g$. The same happens with the restricted input $c?x : S \rightarrow W_1$ and alternation if $g$ then $W_1$ else $W_2$.

Based on the interaction patterns, we may infer which channels can be used to perform a contract livelock-free BRIC composition. For that, we define a function $Allowed(Ctr)$, which returns all communication channels that can be used for compositions without generating a divergent contract.

124
Definition 5.5 (Allowed) Let $\text{Ctr}$ be a component contract. Then, the set of allowed channels of $\text{Ctr}$ is given by the function $\text{Allowed} (\text{Ctr})$, which is defined as follows:

$$\text{Allowed} (\text{Ctr}) = \mathcal{C}_{\text{Ctr}} \setminus \{ c : \mathcal{C}_{\text{Ctr}} \mid \exists s : \text{MIN}_\text{IP} (\mathcal{B}_{\text{Ctr}}) \bullet \text{ran}(s) \subseteq \{ | \ c \ | \} \}$$

Informally, a communication channel $c$ from $\mathcal{C}_{\text{Ctr}}$ belongs to $\text{Allowed}$ only if there is no interaction pattern $\text{MIN}_\text{IP} (\mathcal{B}_{\text{Ctr}})$ composed only by events on $c$. Hence, using these channels on binary compositions do not introduce contract livelock because even after hiding these events, all interaction patterns still have at least one further communication on a different channel. That means that the process is still able to communicate with its environment.

By way of illustration, let us consider these two component contracts, $\text{Ctr}_1$ and $\text{Ctr}_2$, with the following I/O processes as their behaviours.

$$\mathcal{B}_{\text{Ctr}_1} = a \rightarrow b \rightarrow \mathcal{B}_{\text{Ctr}_1}$$
$$\square \ n.1 \rightarrow n.2 \rightarrow \mathcal{B}_{\text{Ctr}_1}$$
$$\mathcal{B}_{\text{Ctr}_2} = c \rightarrow d \rightarrow \mathcal{B}_{\text{Ctr}_2}$$
$$\square \ s.1 \rightarrow \mathcal{B}_{\text{Ctr}_2}$$

In a very similar way, we define a function $\text{AllowedBin}(\text{Ctr})$, which returns all pairs of communication channels that can be used for compositions without generating a divergent contract.

Definition 5.6 (AllowedBin) Let $\text{Ctr}$ be a component contract. Then, the set of allowed pairs of channels of $\text{Ctr}$ is given by the function $\text{AllowedBin}(\text{Ctr})$, which is defined as follows:

$$\text{AllowedBin}(\text{Ctr}) = \{ (c_1, c_2) : \mathcal{C}_{\text{Ctr}} \mid (\neg \exists s : \text{MIN}_\text{IP} (\mathcal{B}_{\text{Ctr}}) \bullet \text{ran}(s) \subseteq \{ | \ c_1 \ | \} \} \}$$

Informally, a pair communication channels $(c_1, c_2)$ from $\mathcal{C}_{\text{Ctr}}$ belongs to $\text{AllowedBin}$ only if there is no interaction pattern $\text{MIN}_\text{IP} (\mathcal{B}_{\text{Ctr}})$ composed only by events on $c_1$ and $c_2$. Hence, using these channels on unary compositions do not introduce contract livelock because even after hiding these events, all interaction patterns still have at least one further communication on a different channel. That means that the process is still able to communicate with its environment.
In order to find out the channels that can be used in a binary composition of \textit{Ctr}_1 and \textit{Ctr}_2, we need to identify the interaction patterns of their behaviours. In this example, they are: \textit{MIN}\_\textit{IP}(\mathcal{B}_{\textit{Ctr}_1}) = \{\langle a, b \rangle, \langle n.1, n.2 \rangle\} and \textit{MIN}\_\textit{IP}(\mathcal{B}_{\textit{Ctr}_2}) = \{\langle c, d \rangle, \langle s.1 \rangle\}. Now, we are able to identify the interaction patterns that do not contain communications on a single channel. In our example, the interaction patterns \langle a, b \rangle for \mathcal{B}_{\textit{Ctr}_1} and \langle c, d \rangle for \mathcal{B}_{\textit{Ctr}_2} are the only ones that satisfy this constraint. For this reason, in our example, since the set of allowed channels is composed by the channels that are used in these interaction patterns, we get \text{Allowed}(\textit{Ctr}_1) = \{a, b\} and \text{Allowed}(\textit{Ctr}_2) = \{c, d\}. This means that these channels may be used in compositions.

Intuitively, the channels in the set of allowed channels may not introduce any livelock if they are used in a binary composition, because even if we hide them, no infinite path of invisible events is introduced. For example, \( P = (a \rightarrow b \rightarrow P) \setminus \{a\} \), behaves like \( P = (b \rightarrow P) \): it does not indefinitely performs internal actions, but rather it continues communicating with its environment through channel \( b \). On the other hand, channel \( n \) does not belong to \text{Allowed}(\textit{Ctr}) = P, due the fact that there is an interaction pattern, \langle n.1, n.2 \rangle, that only contains interactions on \( n \).

It is important to notice that the fact that a channel does not belong to the set of allowed channels, does not directly imply that it may cause divergence if it is used in a composition. The details of this particular case will be discussed later in this section.

We now turn to presenting the extra conditions in which we guarantee that the composition of components using \textit{BRIC} composition rules does not introduce divergence.

### 5.5.1 Conditions for Livelock Freedom

In order to guarantee livelock freedom in \textit{BRIC} components, we propose some additional conditions that must be incorporated in the set of side conditions of the composition rules. These new conditions in \textit{BRIC} are classified in two groups: \textit{Safe Simple Composition} and \textit{Safe Complex Composition}.

The first group, \textit{Safe Simple Composition}, represents the simplest conditions and requires less complex verifications. On the other hand, the second group, \textit{Safe Complex Composition}, deals with more complex cases that require a more careful analysis of constituent behaviours.

As explained before, the interleave composition rule does not introduce a con-
tract livelock because the behaviour of the original components are divergent-free by Definition 2.6. For this reason, throughout this section, binary composition refers to communication composition rule.

5.5.2 Safe Simple Composition

The Safe Simple Composition is divided into two conditions: Safe Simple Binary Composition and Safe Simple Unary Composition. The first condition is related to binary compositions and is defined as follows.

**Definition 5.7 (Safe Simple Binary Composition)** Let $\text{Ctr}_1$ and $\text{Ctr}_2$ be two component contracts, and $\text{ic}$ and $\text{oc}$ two communication channels in $\mathcal{C}_{\text{Ctr}_1}$ and $\mathcal{C}_{\text{Ctr}_2}$, respectively. The tuple $(\text{Ctr}_1, \text{Ctr}_2, \text{ic}, \text{oc})$ satisfies Safe Simple Binary Composition if, and only if:

- $\text{ic} \in \text{Allowed}(\text{Ctr}_1) \lor \text{oc} \in \text{Allowed}(\text{Ctr}_2)$

According to the above definition, to check for Safe Simple Binary Compositions, we only need to check if the communication channels used by the composition belong to the set of allowed channels of the corresponding contract.

The second condition is related to unary compositions, which assemble two channels of the same component. This condition is defined below.

**Definition 5.8 (Safe Simple Unary Composition)** Let $\text{Ctr}$ be a component contract, and $\text{ic}$ and $\text{oc}$ two communication channels in $\mathcal{C}_{\text{Ctr}}$. The triple $(\text{Ctr}, \text{ic}, \text{oc})$ satisfies Safe Simple Unary Composition if, and only if:

- $(\text{ic}, \text{oc}) \in \text{AllowedBin}(\text{Ctr})$

In order to check for safe simple unary compositions, we need to check if the channels involved in the composition cannot compose an interaction pattern that only communicates on at most both channels. This restriction is due the fact that by hiding both channels of an interaction pattern that contains only $\text{ic}$ or $\text{oc}$ a contract livelock is introduced.

The verification of both conditions, Safe Simple Unary Composition and Safe Simple Binary Composition, are extremely simple and their implementations are computationally rather cheap. Unfortunately, they are too restrictive since they forbid compositions that might not introduce livelock using channels that not belong to the set of allowed channels. These compositions require a more careful verification as we explain in the sequel.
5.5.3 Safe Complex Composition

A communication channel is not in *Allowed* when there is an interaction pattern that contains only that channel. We classify such a channel as a *complex channel*. By way of illustration, let us consider these two component contracts, $Ctr_1$ and $Ctr_2$, with the following I/O processes as their behaviours.

$$
\begin{align*}
B_{Ctr_1} &= a.1 \rightarrow a.3 \rightarrow B_{Ctr_1} \\
B_{Ctr_2} &= b.2 \rightarrow b.1 \rightarrow B_{Ctr_2}
\end{align*}
$$

In this example, we observe that $Ctr_1$ and $Ctr_2$ have no channels in their set of *Allowed* channels because their interaction patterns are composed only by events on $a$ and $b$, respectively. Therefore, if we hide the channel $a$, the process $B_{Ctr_1}$ will perform an infinite sequence of internal events without communicating with the environment. The same behavior takes place when we hide the channel $b$ in $B_{Ctr_2}$. However, if we perform a communication composition of $Ctr_1$ and $Ctr_2$ via the complex channels $a$ and $b$, a contract livelock is not introduced.

Intuitively, one way to ensure a safe composition using two complex channels is to check if the events of these channels are able to synchronise, generating an infinite loop and, consequently, introducing a contract livelock. In what follows, we describe in details our local strategy to ensure a safe complex composition, checking if an infinite internal behaviour may be reached due the synchronisation on those events.

Our first step in this verification captures the interaction patterns of the channels that are used in a complex composition. To achieve this goal, we define the function $\text{ComplexIP}(Ctr, c)$ that takes a contract $Ctr$ and a complex communication channel $c$, and returns all interaction patterns that only interact on $c$.

**Definition 5.9 (ComplexIP.)** Let $Ctr$ be a component contract and $c$ a communication channel. The set of complex interaction patterns is defined as:

$$
\text{ComplexIP}(Ctr, c) = \{ s : \text{MIN}_\text{IP}(B_{Ctr}) \cdot \text{ran}(s) \subseteq \{|c|\} \} $

Once we have the sets of complex interaction patterns, we have to investigate if the synchronisation is feasible. To achieve this goal, we use an auxiliary function $\text{RenameCIP}(P, c)$, which renames the channels of the set of complex interaction patterns on $c$ to a fresh name.
Definition 5.10 (RenameCIP.) Let Ctr be a component contract, c a communication channel, and x a fresh name. The set of renamed complex interaction patterns on c is defined as:

\[ \text{RenameCIP}(Ctr, c) = \{ s : \text{ComplexIP}(Ctr, c) \cdot \text{rename}(s, c, x) \} \]

where, \( \text{rename}(s, c, x) \) replaces all references to c on s by x.

This function renames all complex interaction patterns, which only communicate on c. The reason for this renaming is that we have to check if these complex channels used in the composition are able to communicate on the same events, generating an infinite loop.

After renaming all complex interaction patterns of both channels used in the composition to the same fresh name, we are able to verify if there exists a sequence that can be reached by the concatenation of the complex interaction patterns in each of the sets of renamed complex interaction patterns. This sequence represents a possible behavior of the composition leading to an infinite path without communicating with the environment. If such a sequence exists, the synchronisation on those events is feasible and, consequently, a contract livelock may be introduced.

In our analysis, we actually look for the Minimum Common Interaction Pattern (MCIP) because we only need to perform this verification until the first minimum sequence is found.

In order to perform this verification, we define a function \( S^n \) that returns the set of all possible sequences sizing up to n that result from any arbitrary concatenation of elements of the set S of interaction patterns (possibly with elements repetition).

\[ S^n = \{ t : \Sigma^* | (\exists ss : \text{seq}(\Sigma^*) \mid \text{ran}(ss) \subseteq S \wedge t = \sim ss \wedge \#t \leq n) \} \]

Finally, we formally define the function MCIP that receives two set of interaction patterns, \( S_1 \) and \( S_2 \), and returns the set of Minimum Common Interaction Pattern, i.e. that sequences of events resulting from the concatenation of the elements of each interaction pattern individually, on which both sets agree:

\[ \text{MCIP}(S_1, S_2) = S_1^n \cap S_2^n, \quad \text{where } n = \max(\sum_{e:S_1} \#e, \sum_{e:S_2} \#e) \]

We are now able to define our second group of compositions, Safe Complex Composition, which is also divided into two groups: Safe Complex Unary Composition and Safe Complex Binary Composition. The first group deals with unary compositions.
Definition 5.11 (Safe Complex Unary Composition) Let $C_{tr}$ be a component contract, and $ic$ and $oc$ two communication channels in $C_{C_{tr}}$. The triple $(C_{tr}, ic, oc)$ satisfies the Safe Complex Unary Composition if, and only if:

- $MCIP(RenameCIP(C_{tr}, ic), RenameCIP(C_{tr}, oc)) = \emptyset$

The second group is related to binary compositions.

Definition 5.12 (Safe Complex Binary Composition) Let $C_{tr_1}$ and $C_{tr_2}$ be two component contracts, and $ic$ and $oc$ two communication channels in $C_{C_{tr_1}}$ and $C_{C_{tr_2}}$, respectively. The tuple $(C_{tr_1}, C_{tr_2}, ic, oc)$ is a Safe Complex Binary Composition if, and only if:

- $MCIP(RenameCIP(C_{tr_1}, ic), RenameCIP(C_{tr_2}, oc)) = \emptyset$

The composition of two complex channels is problematic if it is feasible for them to interact. More specifically, this interaction represents an infinite sequence of internal events that will perform without communicating with the environment if these complex channels are used in the composition, resulting in a contract livelock. Hence, if no such combination can be reached the composition is safe.

Finally, we establish the interesting results that represent the general conditions in which the compositions can be performed in order to build livelock-free component contracts.

Theorem 5.1 (Safe Unary Composition) Let $C_{tr}$ be a component contract, and $ic$ and $oc$ two communication channels in $C_{C_{tr}}$. The compositions $C_{tr}[ic \leftrightarrow ic]$ and $C_{tr}[ic \leftrightarrow oc]$ are livelock-free if, and only if, the triple $(C_{tr}, ic, oc)$ is either a Safe Simple Unary Composition or a Safe Complex Unary Composition.

Theorem 5.2 (Safe Binary Composition) Let $C_{tr_1}$ and $C_{tr_2}$ be two component contracts, and $ic$ and $oc$ two communication channels in $C_{C_{tr_1}}$ and $C_{C_{tr_2}}$, respectively. The composition $P[ic \leftrightarrow oc]Q$ is a livelock-free if, and only if, the tuple $(C_{tr_1}, C_{tr_2}, ic, oc)$ is a Safe Simple Binary Composition or a Safe Complex Binary Composition.

Theorems 5.1 and 5.2 ensure that the contract resulting from a composition using the BRICK composition rules are livelock-free. Nevertheless, in order to be able to perform further compositions using the resulting contracts in an efficient manner, we still have to calculate the new interaction patterns of the resulting contract. This resulting interaction pattern is part of the resulting contract’s metadata and can be used in the verification of future
compositions.

5.5.4 Dealing with Metadata

The first step after a successful composition consists of calculating the new interaction patterns. As mentioned before, a composition results in a new component contract, which encompasses the original ones. Thus, in order to enable the reuse of the resulting livelock-free component contract, we need to calculate its new interaction patterns.

A strategy to perform this calculation of the new interaction patterns can be based on the trace merge function proposed by Roscoe [Ros10] to calculate the trace semantics of the parallel composition. In this case, the interaction patterns resulting from an interleaving composition is simply the interleaving of the interaction patterns of both contracts merged in interleaving. Furthermore, the resulting interaction patterns of the communication composition is the combination of the interaction patterns of each contract on which the synchronised events are shared and all others events are interleaved. However, in our strategy, the synchronised events must be removed from the resulting interaction patterns since they are removed from the resulting contracts interface. Finally, the calculation of the resulting interaction patterns for the feedback and reflexive compositions is relatively simple: we only need to remove both communication channels from the original interaction patterns.

As we discuss in details in Section 5.6, we applied our strategy to a case study in order to compare the performance between our local livelock analysis and the global analysis. The experiments using the strategy based on Roscoe’s merge [Ros10] to calculate the new interaction patterns were extremely inefficient causing exceeding the memory capacity of our experiment environment due to the large amount of interaction patterns calculated after several interleave compositions.

During further analysis, we detected that our strategy actually does not need to calculate the interleaving of the interaction patterns after binary compositions as the trace semantics does. By way of illustration, let us consider two component contracts $Ctr_1$ and $Ctr_2$, such that:

\[
\begin{align*}
B_{Ctr_1} &= a \rightarrow b \rightarrow B_{Ctr_1} \\
B_{Ctr_2} &= c \rightarrow d \rightarrow B_{Ctr_2}
\end{align*}
\]

The interaction patterns of these contracts are: $MIN_IP(B_{Ctr_1}) = \{(a, b)\}$
and $MIN\_IP(B_{Ctr_2}) = \{\langle c, d \rangle\}$. If we perform an interleaving composition of these contracts using the strategy based on Roscoe’s merge to calculate the new interaction patterns, we get all possible sequences resulting from merging these two interaction patterns $\{\langle a, b, c, d \rangle, \langle a, c, d, b \rangle, \ldots \}$. This calculation generates a large quantity of interaction patterns that always communicate on the same channels, in this case, $a, b, c, d$. The only difference between these sequences is the order that they happen. This order, however, is not relevant for our strategy. This is because, any BRIC binary composition requires the alphabets of the contracts to be disjoint; hence, the resulting interaction patterns will not interact on a single channel. Moreover, following BRIC, further compositions, say with $Ctr_3$, will be made on a one-to-one basis (i.e. one channel to one channel). As a consequence, the composition of the interleaved component with $Ctr_3$ might be problematic only because of the interaction of $Ctr_3$ with either $Ctr_1$ or $Ctr_2$ (although the composition is made between $Ctr_3$ and $Ctr_1$ $[[[]]]$ $Ctr_2$, interactions will be a result of communication between $Ctr_3$ and either $Ctr_1$ or $Ctr_2$).

In this context, our strategy does not need to be concerned about the interaction patterns generated by the interleaving of the original interaction patterns. The reason for this is that a livelock is introduced because the original interaction patterns were transformed into internal events, since the interaction patterns generated by the interleaving interact with all channels of both processes in order to lead to a recursion. As a result, they will never introduce an infinitive path before it happens in the original ones. In summary, we are only interested in the minimum interaction patterns that can introduce livelock, rather than in the sequences that are formed by the permutation of the minimum interaction patterns after a composition.

Based on these analysis, we define a variation of the merge function originally presented by Roscoe. Intuitively, the resulting interaction patterns are those in which either of the contracts may execute independently and those resulting from the interaction of both contracts.

**Definition 5.13 (Interaction Patterns Merge)** Let $Ctr_1$ and $Ctr_2$ be two component contracts and $ic$ and $oc$ two communication channels in $C_{Ctr_1}$ and $C_{Ctr_2}$, respectively. The interaction patterns merge function is defined as follows.

Let $IP_1$ and $IP_2$ be the sets of minimum interaction patterns $MIN\_IP(B_{Ctr_1})$ and $MIN\_IP(B_{Ctr_2})$, which are part of the metadata $K_{Ctr_1}$ and $K_{Ctr_2}$, respectively.

132
MergeIP(Ctr1, Ctr2, ic, oc) =
\{ s \mid s \in IP_1 \land \{ ic \} \cap \text{ran}(s) = \emptyset \}
\cup
\{ t \mid t \in IP_2 \land \{ oc \} \cap \text{ran}(t) = \emptyset \}
\cup
\left\{ s, t, sx, tx \mid \begin{array}{l}
s \in IP_1 \land \{ ic \} \cap \text{ran}(s) \neq \emptyset \\
\land t \in IP_2 \land \{ oc \} \cap \text{ran}(t) \neq \emptyset \\
\land sx = \text{rename}(s, ic, x) \land tx = \text{rename}(t, oc, x) \\
\bullet sx \parallel \langle sx, tx \rangle \{ x \} tx \end{array} \right\}

where, x is a fresh name, and the merge function is defined as follows.

1. Our merge function has the initial traces sx_0 and tx_0 as arguments. This allows us to merge a n concatenations of a trace sx with m concatenations of a trace tx.

2. In the cases in which one side is willing to perform an independent event (lines 4 and 5), we do not take all possible combinations of permuting the independent events for the reasons previously explained;

3. In the cases in which one side is willing to perform a synchronisation event, say c, and the other side has finished (lines 2 and 3), we “reset” the side that has finished. As a consequence, our merge includes interaction patterns that are the result of merging n concatenations of the left trace with m concatenations of the right trace. It is important to notice that this will not lead to an infinite calculation because this function is defined only for traces that interact at least once. Hence, by “resetting” one of the sequences, we are enforcing at least one synchronisation, which will consume c, hence, decreasing the size of one of the sequences by at least one.
Based on these functions, the interaction patterns of the compositions are defined as follows.

**Definition 5.14 (Binary Composition Interaction Patterns)** Let $\text{Ctr}_1$ and $\text{Ctr}_2$ be two component contracts and $\text{ic}$ and $\text{oc}$ two communication channels in $\text{C}_{\text{Ctr}_1}$ and $\text{C}_{\text{Ctr}_2}$, respectively. The interaction patterns of the binary compositions are defined as follows.

\[
\begin{align*}
\text{MIN}_{\text{IP}}(\text{Ctr}_1 \parallel \text{Ctr}_2) &= \text{MIN}_{\text{IP}}(\mathcal{B}_{\text{Ctr}_1}) \cup \text{MIN}_{\text{IP}}(\mathcal{B}_{\text{Ctr}_2}) \\
\text{MIN}_{\text{IP}}(\text{Ctr}_1[\text{ic} \leftrightarrow \text{oc}]\text{Ctr}_2) &= \text{MergeIP}(\text{Ctr}_1, \text{Ctr}_2, \text{ic}, \text{oc})
\end{align*}
\]

In summary, the resulting set of interaction patterns of any interleaving composition is the union of each individual set of interaction patterns of the original component contracts, since these are the minimum interaction patterns that first introduce livelock in the new contract. The resulting interaction patterns of a communication composition uses the auxiliary function $\text{MergeIP}$ that calculates the unique minimum sequences that leads to a recursion after a communication composition.

Finally, we formalise the metadata calculation for the unary compositions, which simply remove the composed channels from the original interaction patterns.

**Definition 5.15 (Unary Composition Interaction Patterns)** Let $\text{Ctr}$ be a component contract and $\text{ic}$ and $\text{oc}$ two communication channels in $\text{C}_{\text{Ctr}}$. The interaction patterns of the unary compositions are defined as follows.

\[
\begin{align*}
\text{MIN}_{\text{IP}}(\text{Ctr}[\text{ic} \leftrightarrow \text{oc}]) &= \{ \text{ip} : \text{MIN}_{\text{IP}}(\mathcal{B}_{\text{Ctr}}) \cdot \text{ip} - \{ \text{ic}, \text{oc} \} \} \\
\text{MIN}_{\text{IP}}(\text{Ctr}[\text{ic} \rightarrow \text{oc}]) &= \{ \text{ip} : \text{MIN}_{\text{IP}}(\mathcal{B}_{\text{Ctr}}) \cdot \text{ip} - \{ \text{ic}, \text{oc} \} \}
\end{align*}
\]

In order to demonstrate the usefulness and the efficiency of our strategy, we have applied it to a real case study to verify livelock freedom in several compositions. The case study used is the dining philosophers, which was defined in Section 4.5. The results are presented in the next section.

### 5.6 Experimental analysis

Our experiment consisted in a comparative analysis of two different approaches: (i) local analysis, which we presented in the previous Section 5.5.1 and (ii) a traditional global analysis, which was proposed by Roscoe, and can be directly checked by FDR. In this case study, we start with 3 philosophers
Figure 13: Example of 3 philosophers

and 3 forks (Figure 13), and we increase this number up to 10,000 philosophers and 10,000 forks.

We have conducted the analysis of these cases in two different contexts to calculate the resulting interaction patterns: (i) using the original merge function proposed by Roscoe [Ros10] and used in the definition of the trace semantics, and (ii) using our new approach to calculate the resulting interaction patterns, which partially uses the merge function. This second context was applied in two approaches: for a livelock-free system and for a system with livelock.

In our experiment, we conducted the verification for an increasing number of philosophers (and forks) (3 - 10, 15, 20, 100, 1,000, 10,000). For these verifications, we used a dedicated server with an 8 core Intel(R) Core(TM) i7-2600K, 16 GB of RAM and 160GB of SSD in an Ubuntu system.

The specification of this case study follows the same model that was presented in Section 4.5. Nevertheless, since our approach does not accept parameterised processes, we instantiated every process invocation and, using FDR, verified that we have indeed refined the original processes. An example, we present below the instantiation of Phil(0).

\[
PPH0 = life.0.thinks \rightarrow life.0.sits \rightarrow pfk.0.0.picksup \rightarrow pfk.0.0.picksack \rightarrow pfk.1.0.picksack \rightarrow pfk.0.0.putsdown \rightarrow pfk.0.0.putsack \rightarrow pfk.1.0.putsack \rightarrow pfk.1.0.putsdown \rightarrow pfk.1.0.putsack \rightarrow life.0.getsup \rightarrow PPH0
\]

Our first experiment was focused on our initial strategy to calculate the resulting interaction patterns after a successful composition. As we discussed before, after a binary composition the calculation of the interaction patterns was based on Roscoe’s merge function [Ros10]. During the experiments, we were not able to complete the verification for networks larger than 8.
Table 1: Summary of the Experiment’s Results (livelock-free version)

<table>
<thead>
<tr>
<th>N</th>
<th># Comp</th>
<th>FDR</th>
<th>BRIC(^1)</th>
<th>BRIC(^2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>10</td>
<td>0.942s</td>
<td>0.878s</td>
<td>0.223s</td>
</tr>
<tr>
<td>4</td>
<td>14</td>
<td>1m10.823s</td>
<td>1.231s</td>
<td>0.243s</td>
</tr>
<tr>
<td>5</td>
<td>18</td>
<td>*</td>
<td>3.757s</td>
<td>0.268s</td>
</tr>
<tr>
<td>6</td>
<td>22</td>
<td>*</td>
<td>5.266s</td>
<td>0.272s</td>
</tr>
<tr>
<td>7</td>
<td>26</td>
<td>*</td>
<td>7.479s</td>
<td>0.278s</td>
</tr>
<tr>
<td>8</td>
<td>30</td>
<td>*</td>
<td>*</td>
<td>0.289s</td>
</tr>
<tr>
<td>9</td>
<td>34</td>
<td>*</td>
<td>*</td>
<td>0.298s</td>
</tr>
<tr>
<td>10</td>
<td>38</td>
<td>*</td>
<td>*</td>
<td>0.307s</td>
</tr>
<tr>
<td>15</td>
<td>54</td>
<td>*</td>
<td>*</td>
<td>0.362s</td>
</tr>
<tr>
<td>20</td>
<td>68</td>
<td>*</td>
<td>*</td>
<td>0.393s</td>
</tr>
<tr>
<td>100</td>
<td>398</td>
<td>*</td>
<td>*</td>
<td>0.809s</td>
</tr>
<tr>
<td>1,000</td>
<td>3,998</td>
<td>*</td>
<td>*</td>
<td>4.870s</td>
</tr>
<tr>
<td>10,000</td>
<td>39,998</td>
<td>*</td>
<td>*</td>
<td>3m47s</td>
</tr>
</tbody>
</table>

philosophers and 8 forks because the execution of our strategy caused the machine to run out of memory. In Table 1, we summarise the experiment’s results (* indicates the error, BRIC\(^1\) indicates the results using Roscoe’s merge, and BRIC\(^2\) indicates the results using our optimised merge). This was due to the large amount of interaction patterns, specially after interleave compositions.

Although FDR was already presenting problems in a network of 5 philosophers, we considered that a deeper analysis was needed. In a deep and careful analysis after these results, we realized that, in fact, the interleaving of the original interaction patterns is not relevant for our strategy. The details behind this verification were discussed in 5.5.1.

We then conducted a second experiment performing the same comparative analysis using our new approach to calculate the resulting interaction patterns, which does not compute the interleaving of both sets of the interaction patterns. The results of this verification are summarised in the Table 1.

With our optimisation, our new strategy was able to calculate the interaction patterns for large systems with several compositions. In our case study, for example, the strategy was able to deal with all instances and all compositions correctly in a considerably efficient manner. For instance, for 10,000 philosophers, it executed in less than 4 minutes, considering that 39,800 compositions were verified. On the other hand, the same verification cannot be performed in a feasible time using a global analysis.
The original case study was a livelock free system. For this reason, our approach was not completely verified by using it. The reason for that is because the original specification of the dining philosophers ensures that each philosopher communicates with its environment through the life channel. Hence, when we perform our local livelock analysis, the verification only checks the first condition, Safe Simple Composition, and concludes that the resulting composition is a livelock-free contract. As a consequence, the expensive bit, MCIP, is never invoked. In order to evaluate the performance when complex compositions are performed, we changed the original specification, removing the communication on life from the original processes to intentionally introduce livelock. Using this new specification, we were able to perform and evaluate complex compositions using our MCIP verification.

A sketch of this new specification without the channel life is presented below.

\[
PPH_0 = pfk.0.0.picksup \rightarrow \\
pfk.0.0.picksack \rightarrow pfk.1.0.picksup \rightarrow pfk.1.0.picksack \rightarrow \\
pfk.0.0.putsdown \rightarrow pfk.0.0.putsack \rightarrow pfk.1.0.putsdown \rightarrow \\
pfk.1.0.putsack \rightarrow PPH_0
\]

The results of the application of our strategy to the changed system with livelock is summarised in Table 2.

As we can observe in Table 2, the non-trivial MCIP verification requires
almost the same time as that presented in Table I which does not perform the MCIP analysis. The experiments demonstrate the efficiency of our local livelock analysis in contrast to the traditional global analysis. Finally, although we only use the same example to validate our approach, we can observe that our strategy to verify complex systems for livelock freedom seems promising, since our tool was able to deal with 20,000 processes and 39,800 compositions in only 3m351s.
6 Service Conformance

In this section, we introduce two notions of conformance, named protocol conformance and behavioural conformance, which guarantees that the protocol behaviours and the overall behaviour, respectively, of a component is maintained on a given set of communication channels \(cs\) after composition. First, in Section 6.1, we present an overview on the informal aspects of the notion of service conformance. The formalisation of these aspects within BRICK is presented in Section 6.2. Finally, we illustrate our definitions by presenting the verification of service conformance in the dining philosophers case study in Section 6.3.

6.1 Overview on Service Conformance

The composition rules presented in the Deliverable D24.1 on Compositional Analysis and Design of CML Models [OSA13] were defined on the top of the notion of service conformance, which can be understood as a design principle to be followed, when permitted: unused services (channels and their corresponding behaviour) of a component should be still available after composition. In [Ram11], the degree of satisfaction of this notion varied from preserving all services (strong conformance) to at least one (weak conformance). The set of composition rules for the BRIC component model satisfy the weak service conformance, and the set of composition rules for the BRICK component model satisfy the strong service conformance.

It is essential to discuss the kind of quality attributes we are concerned with. We are interested in preserving behavioural properties. In particular, we support the argument that the preservation of basic properties helps in the preservation of more complex ones. For instance, let’s have a look on notions of service conformance according to [Ram11].

**Definition 6.1 (Service conformance)** Let \(\{Ctr_1, Ctr_2, \ldots, Ctr_n\}\) be a set of components, and \(CCtr\) be a component contract, such that

\[
CCtr = (Ctr_1 \triangleright s_{11} \triangleright s_{12} Ctr_2) \triangleright s_{21} \triangleright s_{22} \ldots \triangleright s_{m1} \triangleright s_{m2} Ctr_n.
\]

Then, the composition \(CCtr\)

- weakly conforms to the services of the components \(Ctr_1...Ctr_n\) if, and only if:

\[
\exists i, c \mid c \in C_{CCtr} \cap C_{Ctr_i} \bullet (B_{Ctr_i} \uparrow \{ c \}) \sqsubseteq_F (B_{CCtr} \uparrow \{ c \})
\]
• **strongly conforms to the services of the components Ctr₁…Ctrₙ if, and only if:**

\[ \forall i, c \in \mathcal{C}_{CCtr} \cap \mathcal{C}_{Ctr_i} \bullet (\mathcal{B}_{Ctr_i} \upharpoonright \{c\}) \sqsubseteq \mathcal{F} (\mathcal{B}_{CCtr} \upharpoonright \{c\}) \]

Satisfying these properties means that the composition conforms to the services of the original components, or, in other words, that services not directly involved in the composition are preserved. The reason these conformance relations do not consider the services involved in the composition is because they are not available afterwards for new assemblies. With this in mind, they considered that these services assist in the implementation of the other external visible services. They captured this notion with a refinement expression which requires that the observed behaviour of the composed components refines the behaviour of the original components, hiding all channels involved in the composition.

As in any composition the channels involved in the integration are removed from the set \( \mathcal{C}_{CCtr} \). As a result, channels within \( \mathcal{C}_{CCtr} \cap \mathcal{C}_{Ctr_i} \) represent exactly the channels of \( Ctr_i \) that remains externally observable after the composition. The behaviour restricted to these channels constitutes the services we look for. The conformance is then expressed by the refinement of services in the components (the original one and the composition).

Strong conformance may not be always desired. It is easy to think in scenarios that part of the component is put aside in order to other components achieve a global goal as part of a coordination strategy of the components of the system. So, strong conformance depends as much of the careful design of its components as of their correct composition. As its name says, the weak conformance notion is more flexible to such condition.

It is interesting to observe that not all quality properties are independent. In fact, the service conformance notion depends on more basic properties. Analysing the refinement expression, we observe that it is based on the semantics of hiding. Any event outside the alphabet used for comparison is assumed to be an internal event. However, we know that this might not be true after component integration. Problems might arise during integration, and stuck the communication of those events.

According to the CSP semantics, the reasons for a system to get stuck are the two classical quality attributes, commonly cited as emergent properties: deadlock-freedom and livelock-freedom [Lev95]. They are cited as emergent properties because in order to verify them it is necessary to check the interaction of several components in the system. Deadlock arises when components
in a composition might be waiting for each other to proceed, so that no external communication is visible. Livelock arises when components infinitely communicate on internal channels (those not externally observable). Deadlock and livelock problems are introduced by different CSP operators. In this context, deadlocks might arise from the synchronisation between components, while livelocks are possible introduced after applying the hiding operator.

The analyses of those quality attributes give us some insights about how to restrict the scenarios in which undesirable problems do not arise in the component integrations. Based on this analysis we presented the composition rules in [OSA13], mainly focusing on deadlock. In Section 5, we extend this work by also considering livelock freedom. As observed for the service conformance notion, the preservation of these classical properties is important in the analysis of general integration problems.

6.2 BRICK Service Conformance

In our work, we consider the strong conformance notion. This notion of conformance is concerned with the preservation of the behaviour of each individual port-protocol implementation, whose definition is presented in Section 2.3. Other conformance notions might be interesting for more specific coordination purposes.

Our first notion of conformance, protocol conformance, which ensures the preservation of the behaviour of each individual port-protocol implementation:

**Definition 6.2 (Protocol conformance)** Let $C_{tr}$ and $C_{ctr}$ be component contracts, and $cs$ a set of communication channels such that $cs \subseteq C_{C_{tr}} \cap C_{C_{ctr}}$: The contract $C_{ctr}$ is in protocol conformance with the component $C_{tr}$ on the communication channels in $cs$ if, and only if:

$$\forall c : cs \bullet (B_{C_{tr}} \upharpoonright \{c\}) \sqsubseteq \mathcal{F} (B_{C_{ctr}} \upharpoonright \{c\})$$

The notion of protocol conformance is concerned only with the maintenance of the behaviour of individual port-protocols for each of the channels in $cs$. A stronger notion, behavioural conformance, guarantees that the overall behaviour of a component is maintained on a given set of communication channels $cs$: 

141
Definition 6.3 (Behavioural conformance) Let $\text{Ctr}$ and $\text{CCtr}$ be component contracts, and $\text{cs}$ a set of communication channels such that $\text{cs} \subseteq \text{C}_{\text{Ctr}} \cap \text{C}_{\text{CCtr}}$. The contract $\text{CCtr}$ is in behavioural conformance with the component $\text{Ctr}$ on the communication channels in $\text{cs}$ if, and only if:

$$B_{\text{Ctr}} \restriction \text{cs} \preceq F B_{\text{CCtr}} \restriction \text{cs}$$

This is a stronger notion of conformance and, as a matter of fact, implies the weaker notion of protocol conformance.

As expected the compositional rules presented in Section 2.3 preserve both protocol conformance and behavioural conformance on the channels that remain in the contract’s interface $\text{C}$. The theorems that follow state these properties.

The first theorem is related to the interleaving composition.

**Theorem 6.1 (Interleaving and Behavioural Conformance)** Let $P$ and $Q$ be two component contracts that satisfy the requirements for a safe interleaving composition $P[\ ||| \ ] Q$. The resulting contract $P[\ ||| \ ] Q$ is in behavioural conformance with the component $P$ on the communication channels in $\text{cs}$ for any $\text{cs} \subseteq \text{C}_{P}$.

A similar theorem is also available and guarantees that the communication composition also respects behavioural conformance on the remaining channels.

**Theorem 6.2 (Communication and Behavioural Conformance)** Let $P$ and $Q$ be two component contracts and $\text{ic}$ and $\text{oc}$ be two communication channels, such that they satisfy the requirements for a safe communication composition $P[\text{ic} \leftrightarrow \text{oc}] Q$. The resulting contract $P[\text{ic} \leftrightarrow \text{oc}] Q$ is in behavioural conformance with the component $P$ on the communication channels in $\text{cs}$ for any $\text{cs} \subseteq \text{C}_{P} \{ \| \text{ic} \| \}$.

Finally, we also present a theorem that guarantees that both the feedback and the reflexive compositions also respect behavioural conformance on the remaining channels.

**Theorem 6.3 (Feedback/Reflexive and Behavioural Conformance)** Let $P$ and $Q$ be two component contracts and $\text{ic}$ and $\text{oc}$ be two communication channels, such that they satisfy the requirements for safe reflexive and feedback compositions $P[\text{oc} \leftrightarrow \text{ic}]$ and $P[\text{oc} \overset{\leftrightarrow}{\rightarrow} \text{ic}]$. The resulting contracts $P[\text{oc} \leftrightarrow \text{ic}]$ and $P[\text{oc} \overset{\leftrightarrow}{ightarrow} \text{ic}]$ are in behavioural conformance with the component $P$ on the communication channels in $\text{cs}$ for any $\text{cs} \subseteq \text{C}_{P} \{ \| \text{ic, oc} \| \}$. 

142
6.3 Case Study on Service Conformance

We have conducted an experiment using the case study presented in Section 3.1, the dining philosophers specification. In this experiment we have verified, using FDR2, that the result of every composition, including the final system is in both protocol conformance and behavioural conformance with the basic contracts. The results of the experiment are presented in Figure 14.

For example, we verified that the contract $\text{Ctr} \_ \text{Phils} \_ \text{Forks} \_ \text{0} \_ \text{0}$ (COMP.2 in the script presented below), which corresponds to the first communication composition between all the philosophers and all the forks, is in protocol conformance and behavioural conformance on the remaining channels with each philosopher and each fork, individually. The channels $\text{fk.0.0}$ and $\text{pfk.0.0}$ are composed, hence, the verification of these properties does not consider them. As a result, for example, the following assertions checks for protocol conformance and behavioural conformance, between $\text{Ctr} \_ \text{Phils} \_ \text{Forks} \_ \text{0} \_ \text{0}$ and $\text{Ctr} \_ \text{Phils} \_ \text{0}$, on the remaining channels, namely $\text{life.0}$ and $\text{pfk.1.0}$. 

![Figure 14: FDR Results for the dining philosophers](image-url)
-- Protocol Conformance
assert PROJECTION(PH.0, {life.0})
    [F= PROJECTION(COMP.2, {life.0})]
assert PROJECTION(PH.0, {pfk.1.0})
    [F= PROJECTION(COMP.2, {pfk.1.0})]

-- Behaviour Conformance
assert PROJECTION(PH.0, {life.0,pfk.1.0})
    [F= PROJECTION(COMP.2, {life.0,pfk.1.0})]

The verification of protocol conformance and behavioural conformance between \textit{CtrPhils} and the other basic components that do not refer to \textit{fk.0.0} and \textit{pfk.0.0} are slightly different as they consider all channels of each basic contract. For example, the following assertions checks for protocol conformance and behavioural conformance, between \textit{CtrPhils\_Forks\_0} and \textit{CtrPhils\_1}, on all channels of \textit{CtrPhils\_1}.

-- Protocol Conformance
assert PROJECTION(PH.1, {life.1})
    [F= PROJECTION(COMP.2, {life.1})]
assert PROJECTION(PH.1, {pfk.1.1})
    [F= PROJECTION(COMP.2, {pfk.1.1})]
assert PROJECTION(PH.1, {pfk.2.1})
    [F= PROJECTION(COMP.2, {pfk.2.1})]

-- Behaviour Conformance
assert PROJECTION(PH.1, {life.1,pfk.1.1,pfk.2.1})
    [F= PROJECTION(COMP.2, {life.1,pfk.1.1,pfk.2.1})]

As previously said, we checked for protocol conformance and behavioural after each composition. The last composition results in the final \textit{SYSTEM}, which does not contain any fork channel in its interface. The only channels available are the philosophers’ \textit{life} channels. Hence, the final assertions verifies if the \textit{SYSTEM} is in protocol conformance and behavioural conformance on \textit{life} with each philosopher individually.

assert PROJECTION(PH.0, {life.0})
    [F= PROJECTION(SYSTEM, {life.0})]
assert PROJECTION(PH.1, {life.1})
    [F= PROJECTION(SYSTEM, {life.1})]
assert PROJECTION(PH.2, {life.2})
    [F= PROJECTION(SYSTEM, {life.2})]
Since, the set of remaining channels contains only one channel each, in this case, the verification of protocol conformance corresponds to the verification of behavioural conformance. As expected, the final system, since it was built using only the composition rules, is in protocol conformance and behavioural conformances with the basic components contracts.
7 Substitutability

Besides composition, substitution is another important aspect in the development of SoSs. According to the principle of type substitutability \cite{LW94, Weh03}, an instance of the subtype should be usable wherever an instance of the supertype is expected, without a client being able to tell the difference. The strongest relation that fulfils this principle is refinement \cite{Ros98}. In CSP, we have, for each denotational model, a refinement relation that obeys substitutability in different degrees; such a notion is absent in BRIC. In this Section, we define a refinement notion, at the BRICK level, which allows a constituent to be replaced with a refined version, leading to refinement of the overall system.

In Section 7.1, we formalise the denotational semantics for BRIC constituents as a function from BRIC contracts to BRIC constituent. Based on this novel semantics, we are then able to provide, in Section 7.2, the formalisation of a notion that plays a central role in our theory: BRIC refinement. This notion guarantees many properties required in constituent compositions, fulfilling the substitutability principle. As desired, all the composition operators are monotonic with respect to refinement, as formally established in Section 7.3. Finally, in Section 7.4, we give a concrete example of our notions of substitutability, by refining of an abstract specification of the leadership election algorithm to the one presented in Section 3.2.

7.1 BRIC semantics

In order to reason about the meaning of a program we need its formal semantics. This can be given in the denotational, algebraical or operational styles. Whatever we choose (driven by project restrictions or taste) the properties satisfied in one model must hold in the others (congruence of semantics). CSP has three main denotational models: traces (\( T \)), stable failures (\( F \), or just failures) and failures-divergences (\( FD \)). Depending on the properties we are interested, one model is more suitable or efficient than others. For example, if what a process can do (or refuse to do) at a certain point of this execution is relevant, it is preferable the adoption of the failures model \( F \), since it copes with refusals (and consequently with acceptances).

We start by proposing a function \( S[\cdot] \) from a BRIC constituent to an underpinning mathematical model. Consider the constituent \( T : (B, R, I, C) \); the semantics of \( T \) is given by \( S[\cdot] \), such that:
\[
S[\langle \mathcal{T} \rangle] = (\text{failures}(\mathcal{B}), \{(c, \text{failures}(\mathcal{B} \upharpoonright c)) \mid c \in \mathcal{C}\})
\]

Each declaration of channel \(c\) has an interface \(i \in \mathcal{I}\) (a type comprising a set of values and operations), and it implies that \(c \mathcal{R} i\). From this point of view, although \(\mathcal{R}\) and \(\mathcal{I}\) do not appear explicitly in \(S[\langle \mathcal{T} \rangle]\) they are implicitly there through \(c\)'s declaration. This semantics captures the relevant properties of a BRIC constituent: its overall behaviour and those exhibited through its channels, which are crucial in composition rules. Note that \(\mathcal{C}\) stands for a set of names and each pair \((c, \text{failures}(\mathcal{B} \upharpoonright c))\) (for all \(c \in \mathcal{C}\)) is constructed by retrieving \(c\)'s interface from \(\mathcal{R}\).

It is possible to extend the channel's interface of a constituent without changing its semantics. In Lemma 7.1 the interface \(i\) stands for a data type where values are tagged with input and output labels. It states that we can always replace \(i\) by one of its conservative extensions (set containment relation), let's say \(i'\) (or vice-versa).

**Lemma 7.1 (Effect of interface extension in \(S[\langle \cdot \rangle]\))** Consider two BRIC constituents \(\mathcal{T} : (\mathcal{B}, \mathcal{R} \cup \{(ch, i')\}, \mathcal{I} \cup \{i\}, \mathcal{C})\) and \(\mathcal{T}' : (\mathcal{B}, \mathcal{R} \cup \{(ch, i)\}, \mathcal{I} \cup \{i'\}, \mathcal{C})\), such that \(\{i, i'\} \cap \mathcal{I} = \emptyset\). We say that \(\mathcal{T} \equiv_{\text{bric}} \mathcal{T}'\) if and only if \(i \subseteq i'\) or \(i' \subseteq i\).

**Proof**

\[
S[\langle \mathcal{T} \rangle] = (\text{failures}(\mathcal{B}), \{(c, \text{failures}(\mathcal{B} \upharpoonright c)) \mid c \in \mathcal{C}\} \cup \{(ch, \text{failures}(\mathcal{B} \upharpoonright \{ch.v \mid v \in i\}))\}) \land
S[\langle \mathcal{T}' \rangle] = (\text{failures}(\mathcal{B}), \{(c, \text{failures}(\mathcal{B} \upharpoonright c)) \mid c \in \mathcal{C}\} \cup \{(ch, \text{failures}(\mathcal{B} \upharpoonright \{ch.v \mid v \in i'\}))\})
\]

\(\Rightarrow [\mathcal{T}\text{ is a BRIC constituent}]\)

\(\forall (ch, i) \in \mathcal{R} \bullet ch \in \mathcal{C} \land i \in \mathcal{I} \land v \in \bigcup \mathcal{I} \land ch.v \in \alpha B \Rightarrow v \in i\)

\(\Rightarrow [i \subseteq i' \land \text{set theory} \land v \in i']\)

\(\Rightarrow [\alpha P \cap Y = \emptyset \Rightarrow \text{failures}(P \setminus X) = \text{failures}(P \setminus (X \cup Y))]\)

\(\text{failures}(\mathcal{B} \upharpoonright \{ch.v \mid v \in i\}) = \text{failures}(\mathcal{B} \upharpoonright \{ch.v \mid v \in i'\})\)

\(\Rightarrow S[\langle \mathcal{T} \rangle] = S[\langle \mathcal{T}' \rangle]\)

\(\Rightarrow \mathcal{T} \equiv_{\text{bric}} \mathcal{T}'\)
7.2 BRIC refinement

The notion of BRIC refinement plays a central role in our theory. It guarantees many properties required in constituent compositions, fulfilling the substitutability principle. All the composition operators are monotonic with respect to refinement, as formally established in the end of this section.

Definition 7.1 (BRIC refinement based on failures) Consider two constituents $T$ and $T'$. We say that $T'$ refines on failures $T$ $(T \sqsubseteq_{bric}^{bric} T')$, if and only if the following condition holds:

$$B_T \subseteq_F B_{T'} \land C_T = C_{T'} \land \forall c : C_T \bullet R_T(c) \subseteq R_{T'}(c)$$

This definition ensures that $T$ and $T'$ have the same interaction points. Moreover, it guarantees that the constituent behaviour of $T'$ refines that of $T$. In semantics terms, if we consider $S[\cdot] = (f, fp)$ and $S[T'] = (f', fp')$, then it means:

$$f' \subseteq f \land \text{dom } fp = \text{dom } fp' \land \forall c \in \text{dom } fd \bullet fp'(c) \subseteq fp(c)$$

As we said before this relation is monotonic and, therefore, it fulfils the substitutability principle in composition rules. To prove this result, we use some straightforward lemmas about our BRIC refinement relation. The following lemma shows that our refinement relation is transitive.

Lemma 7.2 (BRIC refinement is transitive) Consider three BRIC constituents $T : (B, R, I, C)$, $T' : (B', R', I', C')$ and $T'' : (B'', R'', I'', C'')$. If $T \sqsubseteq_{bric}^{bric} T'$ and $T' \sqsubseteq_{bric}^{bric} T''$ then $T \sqsubseteq_{bric}^{bric} T''$.

Proof

$$[S[\cdot]] = (\text{failures}(B), \{(c, \text{failures}(B \upharpoonright c)) \mid c \in C\}) \land$$

$$S[T] = (\text{failures}(B), \{(c, \text{failures}(B \upharpoonright c)) \mid c \in C'\}) \land$$

$$S[T'] = (\text{failures}(B'), \{(c, \text{failures}(B' \upharpoonright c)) \mid c \in C'\})$$

$$\Rightarrow [T \sqsubseteq_{bric}^{bric} T' \land T' \sqsubseteq_{bric}^{bric} T'']$$
failures($B''$) $\subseteq$ failures($B'$) $\land$ failures($B'$) $\subseteq$ failures($B$)  
$\Rightarrow$ [set theory]  
failures($B''$) $\subseteq$ failures($B$)  
$\Rightarrow$ $[T \sqsubseteq^{bric} T' \land T' \sqsubseteq^{bric} T'']$  
$C = C' = C''$ $\land$ $\forall c \in C$ $\bullet \mathcal{R}(c) \subseteq \mathcal{R}'(c) \land \mathcal{R}'(c) \subseteq \mathcal{R}''(c)$  
$\Rightarrow$ [well-formedness conditions]  
$\forall c : C$ $\bullet$ failures($B'$ $| c$ $\subseteq$ failures($B$ $| c$ $\land$ failures($B''$ $| c$ $\subseteq$ failures($B'$ $| c$)  
$\Rightarrow$ [set theory]  
$\forall c : C$ $\bullet$ failures($B''$ $| c$ $\subseteq$ failures($B$ $| c$)  
$\Rightarrow$ [Definition 7.1] $T \sqsubseteq^{bric} T''$

If a BRIC constituent refines a deadlock-free constituent then it is deadlock-free too. Lemma 7.3 formalises this idea.

**Lemma 7.3 (BRIC refinement preserves deadlock freedom)**
Consider two BRIC constituents $T : (\langle B, \mathcal{R}, I, C \rangle)$ and $T' : (\langle B', \mathcal{R}', I', C' \rangle)$. If $T \sqsubseteq^{bric} T'$ and $T$ is a deadlock-free contract, so is $T'$.

**Proof**

$[T$ is a deadlock-free contract$]$  
$\forall s \in \Sigma^*$ $(s, \Sigma') \notin$ failures($B$)  
$\Rightarrow$ $[T \sqsubseteq^{bric} T']$  
failures($B'$) $\subseteq$ failures($B$)  
$\Rightarrow$ [set theory]  
$\forall s \in \Sigma^*$ $(s, \Sigma') \notin$ failures($B'$)  
$\Rightarrow$ $T'$ is a deadlock-free contract

The behaviour of a BRIC constituent is given by an I/O process and, since our relation relies on the CSP failures model, it is important to establish what behavioural properties are preserved by refinement in this model. If an I/O process refines an I/O confluent process then it is I/O confluent too. Theorem 7.1 formalises this idea.

**Theorem 7.1 (Failures refinement preserves I/O confluence)**
Consider two I/O processes $T$ and $T'$. If $T$ is I/O confluent and $T \sqsubseteq^{bric} T'$ then so is $T'$.
Proof

\[
\{s \circ (c_1.a) \cap t, s \circ (c_2.b)\} \subseteq \text{traces}(T')
\]
\[
\Rightarrow [T \sqsubseteq T']
\]
\[
\{s \circ (c_1.a) \cap t, s \circ (c_2.b)\} \subseteq \text{traces}(T)
\]
\[
\Rightarrow [T \text{ is I/O confluent (Proposition [2] and [Ram11])}]
\]
\[
c_1.a \neq c_2.b
\]

(1) \(c_1.a \in \text{inputs}(T) \land \exists i : \text{inputs}(c_1, T)\)
\[
\Rightarrow (s \circ (c_2.b, i) \cap (t - (c_2.b))) \in \text{traces}(T) \lor
\]
(2) \(c_1.a \in \text{outputs}(T) \land \exists o : \text{outputs}(c_1, T)\)
\[
\Rightarrow (s \circ (c_2.b, o) \cap (t - (c_2.b))) \in \text{traces}(T) \lor
\]
(3) \(c_1 = c_2 \land (X \subseteq \text{inputs}(T) \lor X \subseteq \text{outputs}(T) \mid X = \{c_1.a, c_2.b\})\)
\[
\Rightarrow [(1) \land T \text{ is input deterministic (Definition [2.6])}]
\]
\[
(s \circ (c_2.b),\{i\}) \notin \text{failures}(T)
\]
\[
\Rightarrow [T \sqsubseteq T']
\]
\[
(s \circ (c_2.b),\{i\}) \notin \text{failures}(T')
\]
\[
\Rightarrow [\text{failures semantics } \land T' \text{ is input deterministic}]
\]
\[
s \circ (c_2.b, i) \in \text{traces}(T')
\]
\[
\Rightarrow \begin{bmatrix}
T \text{ is input deterministic } \land \\
T \text{ is strong output decisiveness (Definition [2.6]) } \land
\end{bmatrix}
\]
\[
(s \circ (c_2.b, i),\{c_0.v_0\}) \notin \text{failures}(T) \lor
\]
\[
((s \circ (c_2.b, i), \text{outputs}(c_0, T))) \notin \text{failures}(T) \land
\]
\[
(s \circ (c_2.b, i), \text{outputs}(c_0, T) \setminus \{c_0.v_0\}) \in \text{failures}(T))
\]
\[
\Rightarrow [T \sqsubseteq T']
\]
\[
s \circ (c_2.b, i, c_0.v_0) \in \text{traces}(T')
\]
\[
\Rightarrow [\text{by induction on elements of } t - (c_2.b)]
\]
\[
s \circ (c_2.b, i) \cap (t - (c_2.b)) \in \text{traces}(T')
\]
\[
\Rightarrow [(2) \land \text{ applying the same previous reasoning}]
\]
\[
s \circ (c_2.b, o) \cap (t - (c_2.b)) \in \text{traces}(T')
\]
\[
\Rightarrow [(3) \land \text{ choices between inputs and outputs are on different channels}]
\]
\[
X \subseteq \text{inputs}(T') \lor X \subseteq \text{outputs}(T'), \text{ where } X = \{c_1.a, c_2.b\}
\]
\[
\Rightarrow T' \text{ is I/O confluent}
\]

A communication protocol is a deadlock-free I/O process that is generally associated to the behaviour exhibited by some BRIC constituents when restricted to communicate through a particular channel. Two communication
protocols are conjugate if they can consume the outputs of each other when put in parallel. This is a relevant property to the composition rules (except interleaving) and it is preserved by failures refinement. Lemma 7.4 formalises this idea.

**Lemma 7.4 (Failures refinement and protocol conjugation)**

Consider $T$, $R$ and $T'$ three communication protocols. If $T$ and $R$ are conjugate and $T \preceq_f T'$ then $T'$ and $R$ are also conjugate.

**Proof**

\[
[T \text{ and } R \text{ are conjugate}]
\]

\[
\text{outputs}(T) \subseteq \text{inputs}(R) \land \text{outputs}(R) \subseteq \text{inputs}(T)
\]

\[
\Rightarrow [T \preceq_f T']
\]

\[
\text{failures}(T') \subseteq \text{failures}(T)
\]

\[
\Rightarrow [T \text{ and } T' \text{ are strong output decisive and input deterministic}]
\]

\[
\text{outputs}(T') \subseteq \text{outputs}(T) \land \text{inputs}(T') = \text{inputs}(T)
\]

\[
\Rightarrow [\text{set theory}]
\]

\[
\text{outputs}(T') \subseteq \text{inputs}(R) \land \text{outputs}(R) \subseteq \text{inputs}(T')
\]

$T'$ and $R$ are conjugate

Two communication protocols are strong compatible if, when put in parallel, the outputs of each process are accepted by the other in all scenarios. This is a relevant property to the composition rules (except interleaving) and it is preserved by failures refinement. Lemma 7.5 formalises this idea.

**Lemma 7.5 (Failures refinement and strong protocol compatibility)**

Consider $T$, $R$ and $T'$ three communication protocols. If $T \approx R$ (strong protocol compatible) and $T \preceq_f T'$ then $T' \approx R$.

**Proof**

\[
\Rightarrow [T \approx R]
\]

\[
\forall s : \text{traces}(T) \cap \text{traces}(R) \bullet (\text{outputs}(T/s) \cup \text{outputs}(R/s) \neq \emptyset) \land \text{outputs}(T/s) \subseteq \text{inputs}(R/s) \land \text{outputs}(R/s) \subseteq \text{inputs}(T/s))
\]

\[
\Rightarrow [T \preceq_f T']
\]

\[
\forall s \in \text{traces}(T') \cap \text{traces}(R) \bullet s \in \text{traces}(T)
\]

\[
\Rightarrow [T \preceq_f T' \land T \text{ and } T' \text{ are strong output decisiveness}]
\]
outputs(T/s) ⊇ outputs(T'/s)
⇒ [T ⪯F T' ∧ T and T' are input deterministic]
inputs(T/s) = inputs(T'/s)
⇒ [T ⪯F T' ∧ T and T' are strong output decisiveness]
outputs(T/s) ≠ ∅ ⇒ outputs(T'/s) ≠ ∅
⇒ ∀s : traces(T') ∩ traces(R) • (outputs(T'/s) ∪ outputs(R/s) ≠ ∅) ∧
(outputs(T'/s) ⊆ inputs(R/s) ∧ outputs(R/s) ⊆ inputs(T'/s))
⇒ [· ≈ ·] T' ≈ R

An I/O process has the Finite Output Property (FOP) when it cannot perform an infinite sequence of outputs without an input. This is a relevant property to the composition rules and it is preserved by failures refinement. Lemma 7.6 formalises this idea.

**Lemma 7.6 (Failures refinement preserves Finite Output Property)**
Consider T and T' two I/O processes and C and C' the set of channels used in T and T', respectively. If T satisfies the Finite Output Property and T ⪯F T' then T' also satisfies the Finite Output Property.

**Proof**
[T ⪯F T']
C' ⊆ C
⇒ [T satisfies FOP (Finite Output Property)]
∀ c : C • T \ outputs(T, c) is divergent free
⇒ [T ⪯F T' ∧ X ⊆ Σ ∧ hide semantics ]
T \ X ⪯F T' \ X
⇒ [X = outputs(T, c)]
∀ c : C • T \ outputs(T, c) ⪯F T' \ outputs(T, c)
⇒ [T ⪯F T' ∧ T and T' satisfy strong output decisiveness]
∀ c : C' • outputs(T', c) ⊆ outputs(T, c)
⇒ [D(c) = outputs(T, c) \ outputs(T', c) | dom D = C]
∀ c : C • T \ outputs(T, c) ⪯F T' \ outputs(T', c) ∪ D(c)
⇒ [P \ Y = P ⇔ αP ∩ Y = ∅ | P : CSPProcesses ∧ Y ⊆ Σ]
∀ c : C' • T' \ outputs(T', c) ∪ D(c) ⪯F T' \ outputs(T', c)
⇒ [P ⪯F Q ∧ P is divergent free ⇒ Q is divergent free | P, Q : CSP Processes]
∀ c : C' • T' \ outputs(T', c) is divergent free
⇒ T' satisfies the FOP

Two channels of a BRIC constituent are decoupled if the communication through them behaves as communication between channels of distinct processes. This is a necessary condition for feedback composition rule. Lemma 7.7 formalises the fact that our refinement relation for BRIC holds this property.

Lemma 7.7 (BRIC refinement and decoupled channels)
Consider T : ⟨B, R, I, C⟩ and T' : ⟨B', R', I', C'⟩ two BRIC constituents. If T ⊑ bric F T' and \{c₁, c₂\} ⊆ C are decoupled in T then \{c₁, c₂\} ⊆ C' are decoupled in T'.

Proof

\[ [T \sqsubseteq bric F T' \land \{c₁, c₂\} \subseteq C] \]
\[ C = C' \land \{c₁, c₂\} \subseteq C' \]
⇒ \[\{c₁, c₂\} are decoupled in T\]
B | \{c₁, c₂\} ≡ F B | {c₁} ||| B | {c₂}
⇒ [T \sqsubseteq bric F T']
B \sqsubseteq F B'
⇒ [failures refinement semantics]
B' | {c₁, c₂} ≡ F B' | {c₁} ||| B' | {c₂}
⇒ [decoupled channels definition]
\{c₁, c₂\} ⊆ C' are decoupled in T'

A buffering self injection I/O process can stablish a communication between two of its channels via an one-place-buffer without deadlock. This is a necessary condition for reflexive composition rule. Theorem 7.2 formalises the fact that our refinement relation for BRIC holds this property.

Theorem 7.2 (Failures refinement and self-injection compatibility)
Consider two deadlock-free I/O processes T and T'. If T | {c, z} is buffered self-injection compatible and T \sqsubseteq F T' then T' | {c, z} is also buffered self-injection compatible (c and z are I/O channels).
Proof

\[
\begin{align*}
& \begin{cases}
T \upharpoonright \{c, z\} \text{ is self-injection compatible} \\
O^T_z = \text{outputs}(c, T) \land O^T = \text{outputs}(z, T) \land \\
I^T_c = \text{inputs}(c, T) \land I^T_z = \text{inputs}(z, T)
\end{cases} \\
\forall(s, X) : \text{failures}(T \upharpoonright \{c, z\}) | \\
\begin{cases}
s \downarrow O^T = s \downarrow I^T \land s \downarrow O^T = s \downarrow I^T \land X \cap (O^T \cup O^T) = \emptyset \land \\
s \downarrow O^T > s \downarrow I^T \land (s \upharpoonright z, X \cup \{c\}) \in \text{failures}(T \upharpoonright z) \land \\
s \downarrow O^T > s \downarrow I^T \land (s \upharpoonright c, X \cup \{z\}) \in \text{failures}(T \upharpoonright c) \\
\Rightarrow [T \subset_F T' \land O^T = \text{outputs}(T' \cup c) \land O^T = \text{outputs}(T', z) \land \\
I^T_c = \text{inputs}(T' \cup c) \land I^T_z = \text{inputs}(T', z)] \\
O^T_c \subseteq O^T \land O^T_z \subseteq O^T \land I^T_c \subseteq I^T \land I^T_z \subseteq I^T \\
\Rightarrow [(s \in \text{traces}(T' \upharpoonright \{c, z\}) \Rightarrow s \upharpoonright (\Sigma \setminus \{c, z\}) = (\{\}) \land \text{set theory}] \\
(s \downarrow O^T = s \downarrow I^T \land s \downarrow O^T = s \downarrow I^T) \Rightarrow (s \downarrow O^T = s \downarrow I^T \land s \downarrow O^T = s \downarrow I^T) \land \\
(X \cap (O^T \cup O^T) = \emptyset) \Rightarrow (X \cap (O^T \cup O^T) = \emptyset) \\
\Rightarrow [s \downarrow O^T > s \downarrow I^T] \\
\downarrow O^T > s \downarrow I^T \\
\Rightarrow [\text{applying the same reasoning for } s \downarrow O^T > s \downarrow I^T] \\
\begin{cases}
s \downarrow O^T > s \downarrow I^T \land (s \upharpoonright c, X \cup \{z\}) \in \text{failures}(T' \upharpoonright c) \\
\Rightarrow [\text{definition of buffer self-injection compatibility}] \\
T' \upharpoonright \{c, z\} \text{ is self-injection compatible}
\end{cases}
\end{align*}
\]

In this section we have established how our $\texttt{BRIC}$ refinement relation relates with behavioural properties relevant for the composition rules. The next section systematically collects these results to prove that our relation is
monotonic.

7.3 Monotonicity of BRIC operators with respect to refinement

The BRIC composition operators are monotonic with respect to BRIC refinement. It is proved by the next lemmas.

Lemma 7.8 (Interleave composition is monotonic with respect to BRIC refinement)

Consider $T$, $R$ and $T'$ three constituent contracts, such that $C_T \cap C_R = \emptyset$. If $T$ and $R$ are deadlock free contracts and $T \sqsubseteq_F T'$, then $T \parallel R \sqsubseteq_F T' \parallel R$.

Proof

$[T \sqsubseteq_F T' \land \text{Lemma 7.3}]$

$T'$ is a deadlock free contract $\land (C_T = C_T' \Rightarrow C_T \cap C_R = \emptyset)$

$\Rightarrow$ [deadlock free interleave composition: Theorem 2.1 and Definition 2.7]

$T \parallel R$ and $T' \parallel R$ are deadlock free contracts

$\Rightarrow [B_T \parallel R = B_{T'} \parallel B_R \land B_{T'} \parallel B_R \land \text{monotonicity of } \sqsubseteq_F]$

$B_T \parallel R \sqsubseteq_F B_{T'} \parallel R$

$\Rightarrow T \parallel R \sqsubseteq_F T' \parallel R$

Lemma 7.9 (BRIC refinement is monotonic in communication composition)

Consider $T$, $R$ and $T'$ three constituent contracts, such that $C_T \cap C_R = \emptyset$ and $c \in C_T$ and $z \in C_R$. If we can compose $T$ and $R$ in communication through channels $c$ and $z$ and $T \sqsubseteq_F T'$, then $T[c \leftrightarrow z]R \sqsubseteq_F T'[c \leftrightarrow z]R$.

Proof

$[T \sqsubseteq_F T']$

$B_T \parallel c \sqsubseteq_F B_{T'} \parallel c \land (C_T = C_T' \Rightarrow C_T \cap C_R = \emptyset)$

$\Rightarrow [N_{10}^{c \leftrightarrow z} \text{ is a bijection}]$

$(B_T \parallel c)[[N_{10}^{c \leftrightarrow z}] \sqsubseteq_F (B_{T'} \parallel c) [[N_{10}^{c \leftrightarrow z}]]$
Lemma 7.10 (BRIC refinement is monotonic in feedback composition)

Consider $T$ and $T'$ two constituent contracts, such that $\{c, z\} \subseteq C_T$ are decoupled. If we can compose $T$ in feedback through channels $c$ and $z$ and $T \preceq F_T T'$, then $T[c \leftrightarrow z] \preceq F_T T'[c \leftrightarrow z]$.

Proof

$[T \preceq F_T T']$

$B_T \models c \sqsubseteq [N_{\mathcal{I}_0}^{-z}]$ and $N_{\mathcal{I}_0}^{-c}$ are bijections

$(B_T \models c)[[N_{\mathcal{I}_0}^{-z}]] \preceq_F (B_T' \models c)[[N_{\mathcal{I}_0}^{-z}]] \land (B_T \models z)[[N_{\mathcal{I}_0}^{-c}]] \preceq_F (B_T' \models z)[[N_{\mathcal{I}_0}^{-c}]]$

$[\text{Theorem 7.1}] \land (B_T \models c)[[N_{\mathcal{I}_0}^{-z}]]$ and $(B_T \models z)[[N_{\mathcal{I}_0}^{-c}]]$ are I/O confluent

$(B_T \models c)[[N_{\mathcal{I}_0}^{-z}]] \approx (B_T' \models z)[[N_{\mathcal{I}_0}^{-c}]]$

$[\text{Lemma 7.6}] \land (B_T \models c)[[N_{\mathcal{I}_0}^{-z}]]$ and $(B_T \models z)[[N_{\mathcal{I}_0}^{-c}]]$ satisfy FOP

$(B_T \models c)[[N_{\mathcal{I}_0}^{-z}]]$ and $(B_T' \models z)[[N_{\mathcal{I}_0}^{-c}]]$ satisfies FOP

$[\text{Lemma 7.4}] \land \{c, z\}$ are decoupled in $T$

$\{c, z\}$ are decoupled in $T'$

$[\text{deadlock free feedback composition: Theorem 2.1 and Definition 2.11}]$

$T[c \leftrightarrow z]$ and $T'[c \leftrightarrow z]$ are deadlock-free contracts
Lemma 7.11 (BRIC refinement is monotonic in reflexive composition)

Consider \( T \) and \( T' \) two constituent contracts, such that \( \{c, z\} \subseteq C_T \). If we can compose \( T \) reflexively through channels \( c \) and \( z \) and \( T \sqsubseteq_{\text{bric}} F_{T'} \) and \( T \) is a deadlock free contract, then \( T[c \leftrightarrow z] \sqsubseteq_{\text{bric}} T'[c \leftrightarrow z] \).

Proof

\[
\begin{align*}
B_T & \subseteq_{\text{F}} B_{T'} \quad \text{(Theorem 7.3)} \\
& \text{and buffering self injection compatible} \\
B_T & \subseteq_{\text{F}} B_{T'} \quad \text{(Lemma 7.6)} \\
& \text{satisfies FOP} \\
B_T & \subseteq_{\text{F}} B_{T'} \quad \text{(deadlock free reflexive composition: Theorem 2.1 and Definition 2.12)} \\
& \text{T'[c \leftrightarrow z] are deadlock-free contracts} \\
\Rightarrow & \quad \text{monotonicity of failures refinement} \\
B_T & \subseteq_{\text{F}} B_{T'} \quad \text{Bric composition is monotonic with respect to refinement)}
\end{align*}
\]
7.4 Abstracting Leadership Election

In the Leadership Election case study, developed in Section 3.2, we rely on the petition value held by each network’s node to elect a leader. In an early specification phase, one can develop a more abstract algorithm, where a leader is non-deterministically chosen, in such a way that makes the network’s users unable to tell any difference between the two approaches. Let’s analyse the main abstractions between this abstract specification and that of Section 3.2.

All petition values (DIST type) are non-deterministically selected and useless in the election mechanism. In this phase, we have no mechanism to proclaim a network’s leader. Furthermore, all actions take their jobs without considering petitions, for example in the actions: Follower, Leader and Undecided. Where a specific criteria (the higher petition) was applied, with a non-deterministic choice being taken: \( c := <leader> | ^c := <follower> \).

**Undecided** = \( s : \text{seq of NODES_IDS, p:DIST} @\)

\[
\text{node_mem.i.<out>.<reqLeaders> -> node_mem.i.<inn>?leaders:(leaders in set LEADERS_SET) -> node_mem.i.<out>.<reqHpetition> -> node_mem.i.<inn>?hpetition:(hpetition in set HPETITION_SET) -> node_mem.i.<out>.<reqHpetitionid> -> node_mem.i.<inn>?hpetitionid:(hpetitionid in set HPETITIONID_SET) -> (}
\]

\[
dcl c:CLAIM @
\]

\[
(\text{if leaders.n > 0 then c:= <follower>}
\text{else if (s = []) then c:= <leader> | ^c := <follower>}
\text{else c:= <undecided>})\text{Node'(tl(s),c,p)}\]

**Leader** = \( s : \text{seq of NODES_IDS, p:DIST} @\)

\[
\text{node_mem.i.<out>.<reqLeaders> -> node_mem.i.<inn>?leaders:(leaders in set LEADERS_SET) -> if (leaders.n > 0) then Node'(s,<undecided>,p)
\text{else Node'(s,<leader>,p)}\]

158
The final system now has the same interface of that presented in 3.2 with a more abstract behaviour. Using the same arguments of Section 3, we can translate this CML specification to BRIC (let’s call the final system $Ctr_{Abs_{System}}$) and by equality of interfaces and abstraction of behaviour we can expect that our BRIC refinement relation (Definition 7.1) holds for these specifications and, therefore $Ctr_{Abs_{System}} \sqsubseteq_{bric} Ctr_{System}$. We have checked refinement between CSP versions of these processes and it indeed holds.
8 Timed Behaviour

The focus of this deliverable is on compositional analysis involving untimed behaviour. In this section, we briefly consider a possible approach to analysis taking into account the time aspects of CML. In order to reuse all the analysis strategies developed so far, our approach to addressing timed behaviour is directly based on that presented in [SCHS10b], originally developed for a version of Circus, named Circus Time.

In this technique, a syntactic transformation strategy is used to split a timed program into two parallel components: an untimed program that uses timer events, and a collection of timers. Then it is shown that, with the timer events, it is possible to reason about time properties in the untimed language, and so untimed analysis can be directly applied. Soundness is established using a Galois connection between the untimed UTP theory of Circus and the proposed time theory.

Figure 15 illustrates the steps for using the framework for Circus Time, which can be summarised as follows.

1. One starts with a program $P$ and a time requirement $R$, both defined using Circus Time. The designer gives a complete description of the system and uses the same notation to describe the desired properties.

2. With the help of the function $\Phi$, one obtains a normal form program that has the same semantics as the original program. The normal form is composed of two parts: a set of interleaved timers and a program $P' = \Phi(P)$ with no time operators, but containing internal timer events. The structure of the normal form of the requirement $R$ is the same. Furthermore, the validation requires that the timers of the normal forms of $P$ and $R$ are equivalent. In practice, adjustments may be needed to the normal form of the specification to add extra timers that may be used in the implementation. The presence in the abstract model of timers that are not used does not affect its behaviour.

3. Next, one applies the function $L$, an abstraction function (mapping time into untimed programs) to the program $P'$ and to the requirements $R'$. The framework is actually parametrised by this mapping function.

4. Finally, one shows that the untimed program $L(P')$ satisfies the untimed requirements $L(R')$. This guarantees that the timed program $P$ satisfies the timed requirements $R$ as proved in [SCHS10b].

Based on the idea that timed programs are implemented with timers (the
Figure 15: A heterogeneous framework for analysis of timed programs in Circus

system clock or a dedicated timer) the strategy in [SCHS10b] defines a normal form for Circus Time actions: a parallel composition of a set of timers and an untimed program that synchronises on the timer events. Semantically, these events have a specialised behaviour when they appear in an external choice or in a parallel composition. Therefore, strictly, the language used to express the normal form is an extension of Circus Time with the timer events.

As defined in [SCHS10b], a normalised action takes the following form.

\[ \Phi(A) \parallel \text{Timers}(k, n) \]

The function \( \Phi \) maps an arbitrary Circus Time action to an action that relies on timer events to synchronise with timers, and does not include any of the time operators, whereas \( \text{Timers}(k, n) \) is a set of interleaved timer actions \( \text{Timer}(i) \) with indices \( i \) from \( k \) to \( n \). The \( \parallel \) operator is defined in terms of the parallel composition operator, but deals explicitly with termination of the synchronisation between \( \Phi(A) \) and \( \text{Timers}(k, n) \). The indices \( k \) to \( n \) are used for the timers because each normalised action has its own set of timers.
The specification of an indexed timer $i$, with a delay of $d$ time units, is defined in [SCHS10b] as follows.

$$
\text{Timer}(i) \equiv \mu X \bullet \left( \begin{array}{c}
\text{setup}.i?d \to \\
\text{halt}.i \to X \\
\text{wait}d; \\
\text{out}.i.d \to X \\
\text{terminate}.i \to \text{SKIP} \\
\text{terminate}.i \to \text{SKIP}
\end{array} \right)
$$

The event $\text{setup}.i.d$ initialises the times, which subsequently offers the event $\text{halt}.i$ while it waits for $d$ time units; at the end, the event $\text{out}.i.d$ is also offered. After engaging on either $\text{halt}.i$ or $\text{out}.i.d$, the timer is reset. The event $\text{terminate}.i$ is always offered, allowing termination of the timer execution. The events $\text{setup}$ and $\text{out}$ take $d$ as a parameter; these events have specialised behaviour that are sensitive to the delay.

The process $\text{Timers}(k,n)$ represents (the interleaving of) a set of timers, one for each time operator.

$$
\text{Timers}(k,n) \equiv \parallel i : k..n \bullet \text{Timer}(i)
$$

The special parallel composition operator $\text{par}$ applies to an arbitrary action $A$ and to an interleaving of timers $\text{Timers}(k,n)$; it is defined as follows.

$$
A \text{ par } \text{Timers}(k,n) \equiv (A; \text{Terminate}(k,n) \parallel \text{Timers}(k,n)) \setminus \text{TSet}
$$

The set $s_A$ contains all the variables changed by $A$ and $\text{TSet}$ is the set of timer events. $\text{Terminate}(k,n)$ ensures termination of the timers if and when $A$ terminates. The full details of this strategy can be found in [SCHS10b].

Although this framework has been originally developed for Circus Time, it can be used to reason about CML models. Particularly, Circus Time is a subset of the action sublanguage of CML. Nevertheless, more strictly, further work is necessary in order to justify the soundness when applied to CML. Particularly:

- The semantic models of the two languages are different. A timed trace in Circus Time is a sequence of pairs. Each pair records the interaction over a single time unit represented by the sequence index. The first component of the pair is the sequence of events (trace) that occurred during the time unit. The second component is the set of events that
can be refused at the end of the time unit. In CML, a timed trace is an heterogeneous sequence that has two types of elements: events and sets of events (the latter representing refusal experiments). The passage of time is observed through the refusal experiments. At the end of each time interval either a refusal experiment is made or the empty refusal set is recorded.

- The time operators of Circus Time are a subset of those of CML, which includes timeout, wait, timed interrupt and operators that determine when a process must start and when it must finish. Circus Time includes only timeout and wait. Also, whereas in Circus Time timeout is defined in terms of wait, in CML it is the other way round.

Therefore, in order to formally lift the approach from Circus Time to CML, a link must be established between the two models, as illustrated for CSP and Circus (as well as for Circus and CML) in the previous COMPASS deliverable on compositional analysis (D24.1 [OSA13]). This is one of the topics that we plan to explore as future research.
9 Conclusions

Compositional approaches provide mechanisms and tools for constructing systems by plugging components together. However, a safe construction of these systems is still a research challenge. Such trustworthiness is needed during many development activities, such as safe composition of third-party components or the correct adaptation of library components.

In the previous deliverable D24.1 [OSA13] we proposed a correct-by-construction approach for building trustworthy CML SoS, which contained a comprehensive set of composition rules that can be regarded as safe steps in the development. The application of the rules could be used to systematically develop a wide variety of trustworthy component systems, and guaranteed, by construction, the absence of deadlock. The approach covered not only tree-topologies, but also topologies with cycles in a compositional method, without needing to know the overall structure of the system. In [OSA13], we also proposed the use of metadata extending our approach for arbitrary components with improved and lightweight side conditions. Metadata were derived from component-contract elements and were used in substitution to heavier verifications in the version without metadata. Additionally, metadata of compositions can be easily derived from the metadata of its constituting components. As a result, the verification effort was reduced. The benefits of using metadata, however, were limited to the application of the communication and feedback composition rules. In other words, although systematic, our approach was not local for cyclic communicating systems, potentially presenting a state explosion in the verification of such systems.

In Section 4, we proposed a solution based on sophisticated architectural patterns using properties of these patterns that reduce the verification task. Our work was based on that done by Roscoe [Ros10], which considers architectural patterns to reduce the verification effort, by allowing a local analysis of deadlock, even for cyclic communication topologies. However, none of the existing architectural patterns in the literature (including those in [Ros10]) match directly the structure of the leadership election case study presented in Section 3. As a further contribution, we propose and formalise a new architectural pattern. By considering particular topologies, based on these patterns, we obtain a very efficient analysis strategy. In the particular case of deadlock freedom, considering the dining philosophers classical example, we achieved an efficiency comparable to tools like the deadlock checker [Mar95] and strategies as proposed in [RGG+95], dramatically increasing the size of networks we can consider.
As another major contribution, in Section 5, we provided side conditions to ensure livelock-free compositions in \textit{BRIC}. The traditional approach to prove that a concurrent system is livelock-free performs a global analysis in order to verify that a livelock state cannot be reached \cite{Ros10}. However, this verification is still costly and can become unfeasible, considering the number of components of a large system. This strategy is fully automatised by FDR \cite{For05}. On the other hand, we provided additional conditions that guarantee that the original composition rules ensure livelock freedom. Furthermore, we provide a definition of livelock at the level of \textit{BRIC} component contracts, in which we analyse the behaviour $B$ of the component, but we hide the communication channels that are not in the component’s set of visible channels $C$. The reason for this is because in \textit{BRIC} the channels used in the compositions are removed from the set of visible channels $C$, and, consequently, they cannot be used in future compositions; however, they are not hidden in the component’s behaviour. As result, our approach performs a local livelock analysis, which allows us to check only some parts of a system. This strategy requires less calculation, consequently, reducing time and effort in this verification.

In Section 6, we introduced a stronger notion of conformance, named \textit{behavioural conformance}, which guarantees that the overall behaviour of a component is maintained on a given set of communication channels $cs$. Further conformance notions might be interesting for more specific coordination purposes, but are not in the scope of this document.

Besides composition, substitution is another important aspect in the development of SoSs. We defined a refinement notion, at the \textit{BRICK} level, which allows a constituent to be replaced with a refined version, leading to refinement of the overall system. This was the subject of Section 7.

Finally, although the focus of this deliverable is on compositional analysis involving untimed behaviour, in Section 8 we considered analysis techniques for CML involving time. In order to reuse all the analysis strategies developed so far, our approach was directly based on that presented in \cite{SCHS10b}, originally developed for a version of \textit{Circus}, named \textit{Circus Time}. In this technique, a syntactic transformation strategy is used to split a timed program into two parallel components: an untimed program that uses timer events, and a collection of timers. We have briefly discussed how it is possible to reason about time properties in the untimed language, and so the untimed analysis techniques is directly applicable.

In this deliverable, we considered two different case studies, which were presented in Section 3. First, in Section 3.1 we presented the dining philoso-
phers, a classical concurrency problem in which \( n \) philosophers are seated at a round table with \( n \) forks and each fork is placed between each pair of philosophers. Next, in Section 3.2, we formalised a more complex and SoS related case study, the version of the leadership election algorithm used at B&O\(^4\) which is an example of a class of cyclic networks. This algorithm is used in B&O’s networks of Audio and Video (AV) systems with up to 32 systems.

The results presented here are an important step towards one of the COMPASS objectives: developing compositional design and analysis techniques, based on sophisticated architectural patterns (WP24), that will help to realise the potential and promise of SoS. They will foster reusability and substitutability (evolution) of components, by limiting impact and costs of changes. This also has an impact on cost of development, to ensure scalability.

The work reported here can evolve in several interesting directions.

- Concerning deadlock freedom, other behaviour patterns can be considered, possibly aiming at formalising a comprehensive set of patterns applicable to a large variety of systems.

- The approach to compositional livelock analysis seems very promising, but it has not yet been fully formalised, and some aspects like parameters and guards are not addressed. More elaborate case studies are also in our plans.

- The verification of service conformance ought to be exercised with more elaborated case studies, which might raise the need for optimisations like those proposed here for deadlock-freedom and livelock freedom verification.

- Time aspects were briefly addressed. As pointed out in the previous section, to effectively apply the strategy to CML models. Particularly, further work is necessary in order to justify the soundness of the approach; this requires establishing a formal link between Circus Time and CML.

- Some of the results presented here, like the BRIC model and the livelock analysis, are already supported by prototype tools that were used in the development of the case studies. An interesting piece of future work, specially from the practical point of view, is the implementation of such tools in the realm of CML and their insertion in the CML tool chain.

\[^4\]http://www.bang-olufsen.com/
A The Stable Revivals Model

This work uses a less conventional CSP model, proposed in [Ros09], called stable revivals, which has not been yet linked with a CML semantic model. It is used in the client/server pattern and here we present it. This model is more expressive than the stable failures one, which implies that it can capture some interesting properties that cannot be entirely captured using the stable failures model, an example of such property is provided later. This is a reason why the addition of this model, or a similar one, in the CML semantics and a link from CSP to CML seems as an interesting feature for the CML notation.

In the stable revivals model (denoted by $R$), a process is described by a triple $(Tr, Dead, Rev)$ containing its traces, its deadlocks and its stable revivals, respectively. The deadlocks of a process are given by the set of traces after which the process refuses all the events in its alphabet; this set is given by $\text{deadlock}(P)$. Lastly, the stable revivals of a process are given by the set of stable revivals. A stable revival is a triple containing a trace, denoted by $s$, a refusal set, denoted $X$, and a revival event, denoted $a$. The refusal set $X$, similarly to the one described in the stable failures model, describes the set of events that can be refused by the process after the trace $s$. The revival event $a$ represents an event that the process can offer after performing $s$ and refusing $X$. At the state where the revival is recorded, the process must not be able to perform an internal action, otherwise this state is unstable, not being taken into account. The function $\text{revivals}(P)$ gives the set of stable revivals of process $P$. Thus, the representation of a process $P$ in this model is given by $(\text{traces}(P), \text{deadlocks}(P), \text{revivals}(P))$.

The necessity of this model comes from the fact that some of the properties that we intend to capture cannot be captured using failures alone, or they can only be when restricted to maximal failures. Revivals allow to specify properties with a finer control of what a process is allowed to do when offering an event. This is useful to reason about responsiveness and stuck-freeness in distributed systems. An example of such a property states that a process must be able to perform an event of a certain set $S$ when offering events from set $T$. This is a property that cannot be formulated in terms of failures. For instance, we know the events that a process might offer after trace $s$, in the failures model, because of the traces component, and we know what are the maximal failures after this trace. However, we do not know generally if, after performing $s$ and rejecting $X$, it can offer a given event, or, the other way round, if $s$ is performed and event $a$ is offered, whether it can refuse $X$. 

167
As an example, from the semantics of \((a \rightarrow \text{STOP} [] b \rightarrow \text{STOP}) \mid \text{STOP} [] c \rightarrow \text{STOP}\), in the failures model, we are unable to express the property when \(a\) is offered, so must be event \(b\). We know from the traces that the process might perform \(a, b\) or \(c\) initially, it deadlocks after performing any of these events, and for the empty trace it has as failures \(\langle\rangle, R\) such that \(R \subseteq \{a, c\}\) or \(R \subseteq \{a, b\}\). On the other hand, it is easily expressed in the stable revivals model, since it can be expressed by the fact that a revival of the form \((s, \{b\}, a)\) must not be present.

The functions \(\text{traces}(P)\), \(\text{deadlock}(P)\) and \(\text{revivals}(P)\) are calculated inductively based on the constructs of the CSP language. This means that all these functions are defined with clauses for calculating the semantics of each CSP construct in terms of its arguments. The clauses for calculating the \(\text{traces}\) are broadly known and can be found in [Ros98]. The clauses for calculating \(\text{deadlock}\) and \(\text{revivals}\) are depicted in Table 3 and Table 4, respectively. Note that particularly for calculating deadlocks for the parallel operator, as this model is not congruent, we have to calculate the failures of both processes involved using their stable revivals semantics. This calculation is given by the following definition.

**Definition A.1 (failures from Stable Revivals semantics)** Let \(P = (\text{Tr}, \text{Dead}, \text{Rev})\) be a process represented in \(R\). We can calculate its failures as follows.

\[
\text{failures}(P) = \{(s, X) \mid X \subseteq \Sigma^\prime \land s \in \text{Dead}\} \\
\cup \{(s, X), (s, X \cup \{\checkmark\}) \mid (s, X, a) \in \text{Rev}\} \\
\cup \{(s, X) \mid s \langle \checkmark \rangle \in \text{Tr} \land X \subseteq \Sigma\} \\
\cup \{(s \langle \checkmark \rangle, X) \mid s \langle \checkmark \rangle \in \text{Tr} \land X \subseteq \Sigma^\prime\}
\]

Moreover, for calculating the revivals of the external choice and the parallel composition operators, one needs to calculate the failures of the process from its stable revivals just as before. Nevertheless, in this case, we only need to calculate the failures recording the refusals of subsets of \(\Sigma\).

**Definition A.2 (failures (\(\checkmark\)-instable) from Stable Revivals semantics)** Let \(P = (\text{Tr}, \text{Dead}, \text{Rev})\) be a process represented in \(R\). We can calculate its failures as follows.

\[
\text{failures}^b(P) = \{(s, X) \mid X \subseteq \Sigma \land s \in \text{Dead}\} \\
\cup \{(s, X) \mid (s, X, a) \in \text{Rev}\}
\]
\begin{table}
\centering
\begin{tabular}{ll}
\textbf{deadlock(STOP)} & = \{\langle \rangle \} \\
\textbf{deadlock(SKIP)} & = \emptyset \\
\textbf{deadlock(div)} & = \emptyset \\
\textbf{deadlock}(a \rightarrow P) & = \{\langle a \rangle \concat s \mid s \in \text{deadlock}(P)\} \\
\textbf{deadlock}(P ; Q) & = \{s \mid s \in \text{deadlock}(P)\} \\
& \cup \{s \concat t \mid s \concat (✓) \in \text{traces}(P) \land t \in \text{deadlock}(Q)\} \\
\textbf{deadlock}(P \parallel Q) & = (\text{deadlock}(P) \cup \text{deadlock}(Q)) \cap \{s \mid s \neq \langle \rangle \} \\
& \cup (\text{deadlock}(P) \cap \text{deadlock}(Q)) \langle \rangle \\
\textbf{deadlock}(P \mid\mid Q) & = \text{deadlock}(P) \cup \text{deadlock}(Q) \\
\textbf{deadlock}(P \mid\mid X \mid Q) & = \{u \mid \exists (s, Y) : \text{failures}(P), (t, Z) : \text{failures}(Q) \bullet Y \setminus (X \cup \{✓\}) = Z \setminus (X \cup \{✓\}) \\
& \land (s \parallel X \parallel t) \cap \Sigma^* \\
& \land u \in (s \parallel X \parallel t) \cap \Sigma^* \\
& \land \Sigma' = Y \cup Z\} \\
\textbf{deadlock}(P \setminus X) & = \{s \setminus X \mid s \in \text{deadlock}(P)\} \\
\textbf{deadlock}(P \llbracket[\llbracket R \rrbracket]\rrbracket) & = \{s' \mid \exists s \bullet sRs' \land s \in \text{deadlock}(P)\} \\
\end{tabular}
\caption{Deadlocks semantic clauses}
\end{table}

In addition, the triples representing valid processes in this model must satisfy some conditions. A triple \((Tr, Dead, Rev)\) is a valid representation of a process, i.e. belongs to this model, if and only if \(Tr \subseteq \Sigma^*\), \(Dead \subseteq \Sigma^*\) and \(Rev \subseteq \Sigma^* \times \mathcal{P}\Sigma \times \Sigma\) and the following healthiness conditions are respected:

T1: \((s \leq t \land t \in Tr \implies s \in Tr) \land \langle \rangle \in T\)

Dead1: \(Dead \subseteq Tr\)

Rev1: \((s, X, a) \in Rev \implies s \concat \langle a \rangle \in Tr\)

Rev2: \((s, X, a) \in Rev \land Y \subseteq X \implies (s, Y, a) \in Rev\)

Rev3: \((s, X, a) \in Rev \land b \in \Sigma \implies ((s, X, b) \in Rev \lor (s, X \cup \{b\}, a))\)

These sanity conditions state whether or not a given triple \((Tr, Dead, Rev)\) is a valid representation of a process. T1 states that the traces component of a process must be prefix closed and non-empty. Dead1 states that every trace leading to a deadlock is a trace of the process. Rev1 states that the trace formed of the concatenation of the trace of a revival and its revival event is
D2.4.4 - Comp. Anal. and Des. of CML Models (Public)

| revivals(STOP)          | = \emptyset          |
| revivals(SKIP)          | = \emptyset          |
| revivals(div)           | = \emptyset          |
| revivals(a \rightarrow P) | = \{((s, X, a) \mid a \notin X)
|                            | \cup \{(s) \sim s, X, b) \mid (s, X, b) \in \text{revivals}(P)\} |
| revivals(P ; Q)         | = \{(s, X, a) \mid (s, X, a) \in \text{revivals}(P) \}
|                            | \cup \{(s) \sim t, X, a) \mid (s) \sim t, X, a) \in \text{traces}(P) \}
|                            | \land (t, X, a) \in \text{revivals}(Q) \} |
| revivals(P \sqcup Q)    | = \{((s, X, a) \mid (s, X, a) \in \text{failures}^b(P) \cap \text{failures}^b(Q) \}
|                            | \land ((s, Y, a) \in \text{revivals}(P) \cup \text{revivals}(Q)) \}
|                            | \cup \{(s, X, a) \mid (s, X, a) \in \text{revivals}(P) \}
|                            | \cup \text{revivals}(Q) \land s \neq \langle \rangle \} |
| revivals(P \lceil \neg \rangle Q)  | = \text{revivals}(P) \cup \text{revivals}(Q) |
| revivals(P \lfloor \neg \lfloor \rangle Q) | = \{(u, Y \cup Z, a) \mid
|                            | \exists s, t \bullet (s, Y) \in \text{failures}^b(P) \land (t, Z) \in \text{failures}^b(Q) \}
|                            | \land u \in s \parallel X \parallel t \land Y \land X = Z \land X \}
|                            | \land ((a \in X \land (s, Y, a) \in \text{revivals}(P) \}
|                            | \land (t, Z, a) \in \text{revivals}(Q) \}
|                            | \lor a \notin X \land ((s, Y, a) \in \text{revivals}(P) \}
|                            | \lor (t, Z, a) \in \text{revivals}(Q)) \})) \}
|                            | \cup \{(u, Y \cup Z, a) \mid
|                            | \exists s, t \bullet (s, Y, a) \in \text{revivals}(P) \land t \sim (\neg \rangle) \in \text{traces}(Q) \}
|                            | \land Z \subseteq X \land a \notin X \land u \in s \parallel X \parallel t \} |
|                            | \cup \{(u, Y \cup Z, a) \mid
|                            | \exists s, t \bullet (t, Z, a) \in \text{revivals}(Q) \land s \sim (\neg \rangle) \in \text{traces}(P) \}
|                            | \land Y \subseteq X \land a \notin X \land u \in s \parallel X \parallel t \} |
| revivals(P \setminus X) | = \{(s \setminus X, Y, a) \mid (s, Y \cup X, a) \in \text{revivals}(P) \}
| revivals(P \llbracket R \rrbracket) | = \{(s', X, a') \mid \exists s, a \bullet sRs' \land aRa' \}
|                            | \land (s, R^{-1}(X), a) \in \text{revivals}(P) \} |

Table 4: Revivals semantic clauses

a trace of the process. Rev2 states that when a process refuses a set X after
s and offering a, it must refuse any subset of X after s and offering a. Rev3 states that a process either accepts or refuses an event after performing s and refusing X. In a certain way, these conditions represent a set of properties expected by “realistic” processes in this model.

This model has an associated refinement relation denoted by $\lbrack V= \rbrack$. A process $P$ is refined by process $Q$, $P \lbrack V= Q$, if and only if the traces of $Q$ are a subset of the traces of $P$, the deadlocks of $Q$ are a subset of the deadlocks of $P$ and the stable revivals of $Q$ are a subset of the stable revivals of $P$. This definition is formally defined as follows.

**Definition A.3 (Stable revivals refinement relation)** Let $P$ and $Q$ be two CSP processes.

$$P \lbrack V= Q \iff \text{traces}(Q) \subseteq \text{traces}(P) \land$$

$$\text{deadlocks}(Q) \subseteq \text{deadlock}(P) \land \text{revivals}(Q) \subseteq \text{revivals}(P)$$

The meaning of a recursive process has been intentionally delayed because it requires a particular treatment. The meaning of a recursion in CSP is given by the greatest fixed-point in the $\lbrack V= \rbrack$ order.

**Definition A.4 (Greatest fixed-point of $P$)** The greatest fixed-point of the recursive process $P = F(P)$, considering the refinement order in the stable revivals model, is given by the following expression.

$$P \equiv \bigsqcup \{F^n(\text{div}) \mid n \in \mathbb{N}\}$$
References


