Under consideration for publication in Formal Aspects of Computing

Rigorous Development of Component-based Systems using Component Metadata and Patterns

M. V. M. Oliveira[1], P. Antonino[2], R. Ramos[2], A. Sampaio[2], A. Mota[2], and A. W. Roscoe[3]

[1]Departamento de Informática e Matemática Aplicada, Universidade Federal do Rio Grande do Norte, Brazil
[2]Centro de Informática, Universidade Federal de Pernambuco, Brazil
[3]Department of Computer Science, University of Oxford, UK

Abstract. In previous work we presented a CSP-based systematic approach that fosters the rigorous design of component-based development (CBD). Our approach is strictly defined in terms of composition rules, which are the only permitted way to compose components. These rules guarantee the preservation of properties (particularly deadlock-freedom) by construction in component composition. Nevertheless, their application is allowed only under certain conditions whose verification via model checking turned out impracticable even for some simple designs, and particularly those involving cyclic topologies. In this paper, we address the performance of the analysis and present a significantly more efficient alternative to the verification of the rule side conditions, which are improved by carrying out partial verification on component metadata throughout component compositions and by using behavioural patterns. The use of metadata, together with behavioural patterns, demands new composition rules, which allow previous exponential time verifications to be carried out now in linear time. Two case studies (the classical dining philosophers, also used as a running example, and an industrial version of the leadership election algorithm) are presented to illustrate and validate the overall approach.

Keywords: Component-based development; correct by construction; designs; metadata; behavioural pattern; deadlock analysis; CSP.

1. Introduction

Although component-based development (CBD) has been around for a long time [Mah90], over the last decade it has 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

Correspondence and offprint requests to: Marcel Oliveira, Departamento de Informática e Matemática Aplicada, Universidade Federal do Rio Grande do Norte
Campus Universitário, Lagoa Nova, CEP: 59078-970, Natal - RN - Brazil.
e-mail: marcel@dimap.ufrn.br

paradigm is to replace conventional programming with the composition and configuration of reusable and independent units, called components.

Nevertheless, in order to ensure the success of the component-based method, it is essential that we trust the behaviour of the components and, furthermore, of the systems based on them. This is even more important in critical applications. For instance, avionics systems must have high reliability and continue to operate upon a failure [MJG+10], and autonomous agents in a manufacturing system must correctly obey their schedule [Weh00, BGL+08]. Errors in these systems are caused not only by failures of individual components, but by dysfunctional interactions between non-failed components.

The reason of dysfunctional interactions is that real industrial components do not always fit together like 'Lego Pieces', or just using a simple *glue code*. Integration solutions are often developed in an *ad hoc* manner, in which incompatibilities are not discovered until their side effects emerge during implementation [HGK+06]. Critical issues for system construction are related to the design of the communication-based interaction mechanisms that permit components to work together [Spi04]. The correct design of these elements is critical; otherwise the system may malfunction in subtle ways or may not work at all. This concern is even more acute when a group of components are put together and coordinated to accomplish a collective set of tasks [PA98]. Therefore, it is crucial to verify whether component based systems (CBS) satisfy some desired properties. In fact, most dysfunctional interactions are originated by classical problems in concurrent systems, such as deadlock and livelock.

Safety-related properties, including deadlock-freedom and livelock-freedom, are emergent system attributes [Lev95]. In other words, these are properties that emerge from the interactions among multiple system components, and their analysis might not reside in any component system in particular. For this reason, emergent properties cannot be checked directly in an efficient way. In [Min07, MCMM08], it is shown that deciding deadlock-freedom and liveness in interaction systems is NP-hard. Therefore, it is desirable to establish (stronger) conditions that are easier to check and entail the desired properties [GGMC+06].

To help development, these conditions should be intrinsic to the design and implementation rules used by component developers and application integrators [Wal03, MH05]. In this way, a system engineer, who is not an expert in analytic theory, can reason about properties of the design.

Unfortunately, it is at present difficult to verify important properties of component-based systems in industry. Most well known industrial component models, which define components and how they integrate, are widely based on simple, low-level granularity components (EJB [DK06] and COM/DCOM [Mic11]). These are represented by syntactic interfaces, which lack behavioural information and restrict component verifications [FG03].

Ironically, the idea of higher-level granularity component models, such as Wright [All97, ADG98], Fractal [BCL+06] and SOFA [BHP06], has been still waiting for full commercial exploitation [Pla05]. Higher-level granularity component models complement the syntactic information of a component with behaviour. The behaviour can be discriminated between different kinds, usually associated to the assembly behaviour [HJK10a], component, and port. The assembly behaviour is related to the established interaction of one component with another. The *protocol* represents the whole observable behaviour of the component. The behaviour associated to a port, also called *port-protocol*, is that observed from one point of interaction of the component. For the sake of clarity, since in our approach the context explicitly associates behaviour to a port, we call the port-protocol as just protocol.

Nevertheless, formal description methods are getting more and more attention in the development of critical systems because of their accuracy and the use of model checking and theorem proving support [Chi09, Geo86]. Much effort is devoted to the correctness of component-based systems [All97, BCD02, HLL06b, Sif10, CZ07]. These works define component models with precise meanings, or adopt formal specification notations. This makes it possible to analyse the systems and to provide tool support in verifications.

The practice to date has been to verify and validate the system after it has been built [HLL06b, PV02, CCH+09, Geo86] – the system is designed, implemented and then verified and validated. The major issue is the high cost to fix a problem that is found in a late stage in development, especially when the problem requires redesigning the system to meet reliability or some other quality attribute requirement.

Instead of verifying the entire system, other more promising approaches focus on iteratively identifying problems in compositions. However, in most approaches the cost of subsequent compositions is not alleviated by the results of the previous ones [ADG98, BCD02, CK96]. Every composition is taken as a monolithic system for verification, and properties of its constituent parts are not considered. Verification methods do not take advantage of the hierarchical structure of component-based systems. In other words, these methods are usually not compositional, and have scalability problems by not using local analysis when this is possible.

In [RSM10, RSM09], we presented a theoretical foundation for the development of trustworthy component-based systems. We proposed a correct by construction strategy for ensuring the preservation of properties of a CBS from proved properties of its interaction model and of its components. More specifically, we consider the freedom of deadlock. Although we focus on this property, the strategy can be applied to predict other safety and liveness properties. The overall approach is based on the CSP process algebra, which offers rich semantic models that support a wide range of process verification, and comparisons. In fact, CSP has shown to be very useful to support the rigorous development of component based systems, as a hidden formalism for modelling languages used in practice [RSM06, SNMI14]. Nevertheless, the same principles can be transferred to other formal models, and support the implementation of practical tools for component-based development.

In our component model, the necessary constraints for a safe interaction between the components is imposed on the development process. The process is strictly defined in terms of composition rules, meaning that this is the only way to compose components. These rules provide a systematic method that preserves properties (deadlock-freedom) by construction. For instance, the fact that each component to be composed is deadlock-free, as well as component interoperability are checked before composition. In [RSM10], we have investigated compatibility notions in the integration of heterogeneous software components. In [RSM09], we propose three basic composition rules, which can be regarded as safe steps to form a wide variety of trustworthy systems.

The approach is intended to address engineering concerns, and make the expertise on correctness available to engineers who are not experts in understanding the origin of dysfunctional interactions between non-failed components in the system. Moreover, we claim that a constructive approach, in opposition to *a posteriori* verification, is more suitable to component based systems. It preserves quality attributes of the system by construction, and identifies problems early in the design phase. Moreover, we use local analysis, when this is possible, to scale the verifications in our approach. Similarly, systematic approaches [LMC10, HJK10b, Zub11, All97] also propose the use of local analysis using port-protocol, however the calculation of such port-protocol in most systems always depend of the entire analysis of previous compositions. This reduces the practical applicability of such approaches and prevents scaling them.

To underpin this approach, we propose important design constraints. Satisfying these constraints at development, we can certainly trust on the resulting system. Part of these constraints comprise our model for components. They characterise which kinds of components, as well as interactions, are supported in this work. To allow further verifications, we focus on behavioural rich components, in which not only syntactic information about component operations is present, but also the behaviour with the possible valid sequences of operations that the component can perform. The other constraints are the constructive constraints for these components. They aimed to assist system evolution. We focus on notions that predict quality attributes of components in compositions, which is one of the main activities in Component-Based Development [Szy02]. These notions allow checking whether the behaviours of two components are compatible for them to interoperate.

Despite having proved to be a promising approach, since it supports the design (and verification) of the system by construction and helps the understanding of the overall conditions for components to interoperate, improvements are required in order to scale this approach and to reduce the cost of verification related to the side conditions imposed by the composition rules.

In this paper we propose variations of the composition rules presented in [RSM09], with a notion of metadata [LU05] and behavioural patterns [Mar96, Ros98], which allow us to carry out partial results of verification and, furthermore, to decrease the effort to check side conditions in composition rules. Moreover, we provide a fourth composition rule that complements our previous strategy, and allows us to construct more complex systems with safe cyclic dependencies among the components of the system. We also propose some guidelines on the order of application of rules with or without metadata, and formalise the fact that any sequence of rule applications produces deadlock-free components, provided the original components are deadlock-free. Finally, we mechanise the verification of the rule side conditions (by expressing them as refinement assertions that can be checked using tools like FDR [For12, GRABR14]); this allows us to model and automatically analyse design models to provide practical evidence of the gains obtained with our new strategy. This is illustrated with two case studies.

Component refinement and evolution in $\mathcal{BRICK}$ is addressed as an orthogonal issue [dSOSO15]. Other properties such as livelock freedom and a notion of service conformance are addressed in [OSA+13].

The remainder of this paper is structured as follows. First, we present a brief introduction to CSP, which is the underlying process algebra of our approach. In Section 3, we present the development approach

from [RSM09], together with a new composition rule that allows the construction of systems with cyclic topologies, such as the one in both case studies we present. In Section 4, we enrich our component model to include metadata information, and explore variations of the composition rules based on the notion of metadata. In Section 5 we show how the rule side conditions can be expressed as refinement assertions in CSP; these assertions can be mechanically checked using tools like FDR. Our first case study, also used as a running example, is fully discussed in Section 6, where we detail an application of metadata in the construction of the classical dinning philosophers problem, and discuss its benefits and associated cost compared with our previous approach. Section 6 demonstrates that the verification of a single side-condition of one of the rules we have introduced is inefficient because it is verified over the entire system causing an exponential growth in verification time. In Section 7 we present an optimisation to our approach that allows the integration of behavioural patterns into it and revisit the dinning philosophers case study to show the substantial reduction of verification effort. Furthermore, in Section 8, we discuss the results of further experiments we have carried out using a second case study (a leadership election protocol) that show a considerable improvement in the efficiency of our strategy using behavioral patterns; this gives some evidence that the strategy can be used in the development of industrial systems in a practical manner. Finally, in Section 9, we draw our concluding remarks and consider related and future work.

2. CSP

CSP is a process algebra that can be used to describe systems composed by interacting components, which are independent self-contained processes with interfaces that are used to interact with the environment [Ros98]. Most of the CSP tools, like FDR and ProBE [For98], accept a machine-processable CSP, called CSP$_M$. For the sake of presentation, in this paper we use the original CSP notation in the theoretical definitions and CSP$_M$ in the examples. In this section, we use the CSP$_M$ notation.

2.1. CSP Syntax

The two basic CSP processes are `STOP` and `SKIP`; the former deadlocks, and the latter does nothing but terminates. The prefixing `a -> P` is initially able to perform only the event `a`; afterwards it behaves like process `P`. A boolean guard may be associated with a process: `g \& P` behaves like `P` if the predicate `g` is true; it deadlocks otherwise. The operator `P1 ; P2` combines `P1` and `P2` in sequence. The external choice `P1 [] P2` initially offers events of both processes. The performance of the first event or termination resolves the choice in favour of the process that performs either of them. The environment has no control over the internal choice `P1 |~| P2`, in which the choice is resolved internally. The sharing parallel composition `P1 [| cs |] P2` synchronises `P1` and `P2` on the events in the synchronisation set `cs`; events that are not listed occur independently. The alphabetised parallel composition `P1 [| cs1 | cs2 |] P2` allows `P1` and `P2` to communicate in the sets `cs1` and `cs2`, respectively; however, they must agree on events in `cs1`∩`cs2`. Processes composed in interleaving `P1 ||| P2` run independently. The event hiding operator `P \ cs` encapsulates the events that are in `cs`. Next, the untimed time-out `P [> Q` is an operator in which the options of `P` are offered for a short time before it opts to behave like `Q`. For obvious reasons the representation of a time-out is very imperfect in a model without time like that of CSP, in which, `P [> Q` is defined as `(P |~| STOP) [] Q`. Finally, the renaming operator `P [[a <- b]]` behaves like `P` except that all occurrences of `a` in `P` are replaced by `b`. In Section 3, we use $P \llbracket F \rrbracket$, where P is a CSP process and F is a bijection between name prefixes, to denote the renaming of all names $n \in \mathrm{dom}(F)$ by $F(n)$.

CSP provides finite iterated operators that can be used to generalise the binary operators of sequence, external and internal choice, parallel composition, and interleaving. Apart from sequence, all the iterated operators are commutative and associative. For this reason, there is no concern about the order of the elements in the type of the indexing variable. However, for the sequence operator, we require this type to be a finite sequence. As expected, the process `; x:S @ P(x)` is the sequential composition of processes `P(v)`, with `v` taken from `S` in the order that they appear.

In what follows, we illustrate the CSP constructs using a classical concurrency problem, the dining philosophers [Ros98], which is used to validate the use of our approach in Section 6.

The dining philosophers is a classical concurrency problem: n philosophers are seated at a round table with n forks and each fork is placed between each pair of philosophers. In order to eat, a philosopher must

```
RANGE = {1..3}

-- Fork Data Types and Channels
datatype EVENTS = picksup | putsdown | picksack | putsack | thinks | sits | eats | getsup
subtype I_FORK = picksup | putsdown | picksack | putsack
channel fk1, fk2: I_FORK

-- PHIL Data Types and Channels
subtype I_LIFE = thinks | sits | eats | getsup
subtype I_PHFK = picksup | putsdown | picksack | putsack
channel pf1, pf2: I_PHFK
channel lf: I_LIFE

-- Forks
FORK  = (COMPFK(fk1) [] COMPFK(fk2)); FORK
COMPFK(f) = PICKUP_FK(f); PUTDOWN_FK(f)
PICKUP_FK(f) = f.picksup -> f.picksack -> SKIP
PUTDOWN_FK(f) = f.putsdown -> f.putsack -> SKIP

-- Philosophers
PHIL =  PREEAT; EAT; POSTEAT; PHIL
PREEAT =  lf.thinks -> lf.sits -> SKIP
POSTEAT = lf.getsup -> SKIP
EAT = PICKFORKSUP; (lf.eats -> PUTFORKSDOWN)
PICKFORKSUP = PICKUP_PH(pf1); PICKUP_PH(pf2)
PUTFORKSDOWN = PUTDOWN_PH(pf1); PUTDOWN_PH(pf2)
PICKUP_PH(c) = c.picksup -> c.picksack -> SKIP
PUTDOWN_PH(c) = c.putsdown -> c.putsack -> SKIP
```

Fig. 1. CSP behaviour of Philosophers and Forks

pick up the forks on either side. A philosopher who cannot pick up one or the other fork has to wait. However, since there is a limited number of forks, it is necessary to control the access to such resources. Otherwise, for instance, all philosophers might get hungry simultaneously and pick up one fork, then deadlock and starve to death. Even though this example is anthropomorphic, the behaviour associated to forks and philosophers can be represented by components, and their interaction by the composition among these components. Before presenting the actual components (forks and philosophers) that form the system, we explain the behaviour of a fork and a philosopher as simple CSP processes.

The process `FORK` ensures that two philosophers cannot hold a fork simultaneously. All events associated to forks are represented by the channels `fk1` and `fk2`, each one specific for the interactions with each philosopher. The processes `PICKUP_FK` and `PUTDOWN_FK` represent the actions associated with a fork. The former represents picking up a fork and the latter putting it down; picking up a fork always precedes putting it down (`COMPFK`). The external choice means that the first philosopher to pick a fork, holds it until putting it down. All events performed by these processes are defined by the type `I_FORK`. To pick up a fork, we use two events `picksup` and `picksack`. The former represents the intention to pick the fork, and the latter indicates that it has been performed. Similarly, `PUTDOWN_FK` uses `putsdown` and `putsack`. Using a pair of events we are able to ensure the necessary communication synchrony of this example, as, by default, the communication induced by our composition rules is asynchronous.

The events of a fork are a subset of those communicated by a philosopher, since the latter has a more complex life cycle. It is assumed that a philosopher thinks all the time, except when he gets hungry; in this case, he sits on his chair and picks up the necessary forks, eats, and then releases the forks, gets up and starts to think again. Of course, different philosophers might have different preferences about which order they keep or release such forks. We assume that each philosopher picks up and puts down the left fork first.

The process `PHIL` represents the life cycle of a philosopher. It uses the channels `pf1`, `pf2` and `life`, which

represent events associated to fork manipulations and other philosopher activities. The type of `pf1` and `pf2` are similar to `fk1` and `fk2`, and are used to pick up and put down forks. The main activity in the life of a philosopher is eating. Before eating, the philosopher thinks, sits and picks the forks up (`PREEAT`). After eating, the philosopher puts the forks down and gets up (`POSTEAT`). Philosopher actions on his left and right forks are represented by `fk1` and `fk2`, respectively. The processes `PICKFORKSUP` and `PUTFORKSDOWN` represent the elementary actions associated to fork manipulations; the former represents acquiring the two forks, and the latter represents releasing them.

2.2. CSP Semantic Models

CSP offers a number of approaches to formally define the behaviour of processes. A process written in CSP may be understood in terms of an operational semantics (where the process is represented as a labelled transition system, with transitions representing communications); or in terms of an 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 a denotational semantics (where the process corresponds to a denotation 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 major denotational models: traces, failures and failures-divergences [Ros98].

2.2.1. 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 $\mathcal{A}^{*\checkmark} = \Sigma^* \cup \{s \frown \langle\checkmark\rangle \mid s \in \Sigma^*\}$ be the alphabet of communications. Formally, in the traces model each process is identified by a set $T \subseteq \mathcal{A}^{*\checkmark}$ that satisfies the following healthiness condition:

T1. T is nonempty and prefix-closed. This means that it always contains the empty trace $\langle\rangle$ and if $s \frown t \in T$ then $s \in T$.

Here, Σ stands for the set of all visible events and $\frown$ for sequence concatenation.

Given a CSP process P, the traces of P are denoted as $traces(P)$. For example, `STOP` never communicates anything: its set of traces consists only of the empty trace, $traces(\texttt{STOP}) = \{\langle\rangle\}$. Furthermore, the traces of a prefix process `c -> P` are the traces of the prefixed process P, each prefixed with the event `a` first communicated and the empty trace added ($traces(\texttt{c -> } P) = \{\langle\rangle\} \cup \{\langle a\rangle \frown s \mid s \in traces(P)\}$). Details about the other constructors are presented in [Ros98].

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

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

Two processes P and Q are traces-equivalent, $P \equiv_{\mathrm{T}} Q$, if $P \sqsubseteq_{\mathrm{T}} Q$ and $Q \sqsubseteq_{\mathrm{T}} P$, i.e., $traces(P) = traces(Q)$. The process `STOP` is the most refined process in the traces model, i.e., $P \sqsubseteq_{\mathrm{T}} \texttt{STOP}$, 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 $P \,\texttt{|~|}\, Q$ (internal choice) and $P \,\texttt{[]}\, Q$ (external choice) are indistinguishable because both are defined as $traces(P) \cup traces(Q)$.

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., $traces(Q) \subseteq traces(P)$. Safety conditions are concerned with the exclusion of traces only.

2.2.2. Stable failures Model

The stable failures model $\mathcal{F}$ gives a 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 a *refusal* set; the process cannot perform any event in the set X no matter for how long it is offered.

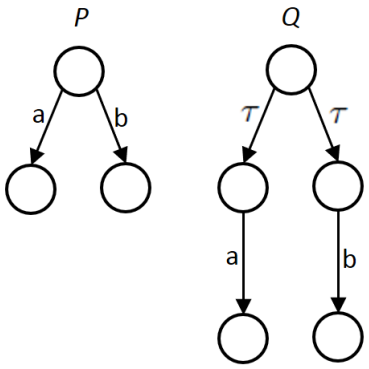


Fig. 2. Labelled Transition Systems for Processes P and Q

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 $\{\mathtt{a}, \mathtt{b}\}$:

```
P = a -> STOP []  b -> STOP
Q = a -> STOP |~| b -> STOP
```

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

$$\mathit{failures}(P) = \{(\langle\rangle, \{\checkmark\})\} \cup \{(\langle \mathtt{a}\rangle, X) \mid X \subseteq \{\mathtt{a}, \mathtt{b}, \checkmark\}\} \cup \{(\langle \mathtt{b}\rangle, X) \mid X \subseteq \{\mathtt{a}, \mathtt{b}, \checkmark\}\}$$
$$\mathit{failures}(Q) = \{(\langle\rangle, X) \mid X \subseteq \{\mathtt{a}, \checkmark\}\} \cup \{(\langle\rangle, X) \mid X \subseteq \{\mathtt{b}, \checkmark\}\}$$
$$\cup \{(\langle \mathtt{a}\rangle, X) \mid X \subseteq \{\mathtt{a}, \mathtt{b}, \checkmark\}\} \cup \{(\langle \mathtt{b}\rangle, X) \mid X \subseteq \{\mathtt{a}, \mathtt{b}, \checkmark\}\}$$

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 record that initially (after the trace $s = \langle\rangle$) the process cannot refuse either $\mathtt{a}$ or $\mathtt{b}$: it can only refuse to terminate ($\{\checkmark\}$). The process Q can perform either $\mathtt{a}$ or $\mathtt{b}$ separately, and refuse $\mathtt{b}$ or $\mathtt{a}$ respectively. The failures of Q do not record any information about the initial state, but only information about the stable states.

It might be convenient to understand the difference between P and Q from an operational perspective. In Figure 2 we present the labelled transition systems of the two processes. The process P is unable to refuse either a or b in its initial state, but can refuse both of these events after it has performed a or b. On the other hand, the process Q is initially unstable, as there are two internal transitions (τ) that are possible for it. Each of these leads to a stable state where either a or b is possible, and the other is refused. Although our focus in on the denotational models, sometimes we consider the operational semantics to help with intuition.

Observe that it is by no means inevitable that every trace of a process has failures: it may never stop performing τ 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 $T = \mathit{traces}(P)$ and $F = \mathit{failures}(P)$, where $T \subseteq \Sigma^{*\checkmark}$ and $F \subseteq \Sigma^{*\checkmark} \times \mathbb{P}(\Sigma^{*\checkmark})$, satisfying the following healthiness conditions (where s, t range over Σ^* and X, Y over $\mathbb{P}(\Sigma^{\checkmark})$):

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. $s \frown \langle\checkmark\rangle \in T \Rightarrow (s \frown \langle\checkmark\rangle, X) \in F$. If a trace terminates successfully by producing $\checkmark$, then it should refuse all events in $\Sigma^{\checkmark}$ at the stable state after $s \frown \langle\checkmark\rangle$.

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

F3. $(s, X) \in F \wedge \forall\, a : Y \bullet s \frown \langle a\rangle \notin T \Rightarrow (s, X \cup Y) \in F$. 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. $s \frown \langle \checkmark \rangle \in T \Rightarrow (s, \Sigma) \in F$. This asserts that if we have any terminating trace $s \frown \langle \checkmark \rangle$, these should refuse Σ at the stable state after s.

For example, `STOP` initially refuses to communicate anything.

$$\mathit{failures}(\texttt{STOP}) = \{(\langle\,\rangle, X) \mid X \subseteq \Sigma^{\checkmark}\}$$

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

$$\mathit{failures}(\texttt{a -> P}) = \{(\langle\,\rangle, X) \mid a \notin X\} \cup \{(\langle a \rangle \frown s, X) \mid (s, X) \in \mathit{failures}(P)\}$$

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 failures refinement). Let P, Q be CSP processes. P is a stable failures refinement of Q, written as $Q \sqsubseteq_{\mathrm{F}} P$, if, and only if: $\mathit{traces}(P) \subseteq \mathit{traces}(Q) \wedge \mathit{failures}(P) \subseteq \mathit{failures}(Q)$.

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 failures-equivalent, $P \equiv_{\mathrm{F}} Q$, if $P \sqsubseteq_{\mathrm{F}} Q$ and $Q \sqsubseteq_{\mathrm{F}} P$, i.e., $\mathit{traces}(P) = \mathit{traces}(Q)$ and $\mathit{failures}(P) = \mathit{failures}(Q)$. The bottom element in $\sqsubseteq_{\mathrm{F}}$ is $(\Sigma^{*\checkmark}, \Sigma^{*\checkmark} \times \mathbb{P}(\Sigma^{*\checkmark}))$, while its top element is $(\langle\,\rangle, \emptyset)$.

An important phenomenon captured by $\mathcal{F}$ is deadlock. Deadlock is a phenomenon pertaining to networks of communicating processes which occur when the processes in the network cannot agree to communicate among themselves nor with the environment, 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 deadlock-free if, after performing a trace s, it 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:

$$\forall\, s : \Sigma^* \bullet (s, \Sigma^{\checkmark}) \notin \mathit{failures}(P)$$

This definition is justified, as in the model $\mathcal{F}$ 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 failures refinement notion preserves the deadlock-freedom of a process. That is, if P is deadlock-free and $P \sqsubseteq_{\mathrm{F}} Q$, then Q is deadlock-free.

2.2.3. 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 livelocked 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 -> Q) \ {a}`. `Q` converts the external event `a` into an internal action τ. Therefore, `Q` indefinitely performs internal actions, which leads to a divergence. As a consequence, `Q` and `P` have the same behaviour in the failures-divergences model. The CSP process `DIV` (the same as `P` or `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 after which the process can perform an infinite sequence of internal (invisible) actions. So, each process P is modelled by the pair $(\mathit{failures}_{\perp}(P), \mathit{divergences}(P))$, where:

- $divergences(P)$ is the (extension-closed) set of traces s after which a process can diverge. Thus, the set $divergences(P)$ contains not only the traces s on which P can diverge, but also all extensions $s \frown t$ of such traces;
- $failures_\perp(P) = failures(P) \cup \{(s, X) \mid s \in divergences(P)\}$.

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 $\Sigma^{*\checkmark}$, and X, Y range over $\mathbb{P}(\Sigma^\checkmark)$:

F.1. $traces_\perp(P) = traces(P) \cup divergences(P)$ is non-empty and prefix closed.
F.2. $(s, X) \in F \land Y \subseteq X \Rightarrow (s, Y) \in F$.
F.3. $(s, X) \in F \land (\forall a \in Y \bullet s \frown \langle a \rangle \notin traces_\perp(P)) \Rightarrow (s, X \cup Y) \in F$.
F.4. $s \frown \langle \checkmark \rangle \in traces_\perp(P) \Rightarrow (s, \Sigma) \in F$.
D.1. $s \in D \cap \Sigma^* \land t \in \Sigma^{*\checkmark} \Rightarrow s \frown t \in D$.
D.2. $s \in D \Rightarrow (s, X) \in F$. This adds all divergences-related failures of F.
D.3. $s \frown \langle \checkmark \rangle \in D \Rightarrow s \in D$. 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 $Q \sqsubseteq_{\text{FD}} P$, if and only if:

$$failures_\perp(P) \subseteq failures_\perp(Q) \land divergences(P) \subseteq divergences(Q)$$

Two processes P and Q are failures-divergences equivalent, $P \equiv_{\text{FD}} Q$, if $P \sqsubseteq_{\text{FD}} Q$ and $Q \sqsubseteq_{\text{FD}} P$, i.e., $failures_\perp(P) = failures_\perp(Q)$ and $divergences(P) = divergences(Q)$. The process `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 `DIV`.

As already mentioned, this is the most appropriate model to reason about safety and liveness properties in CSP; 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 -> P [] b -> P) \ {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 the next section, 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 of 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.

3. Systematic Development of Component based Systems

Our approach is based on a component model that defines the relevant component aspects and constraints. Both components and connectors, as well as their interaction semantics, are characterised in this component

model – which we call $\mathcal{BRIC}$ – that defines the building blocks of our systematic development approach. A component contract encapsulates a component in our approach. It is defined in terms of the component behaviour (represented as a CSP process), its ports (represented as channels) and their respective types. The term port is more conventionally used to represent interaction points for components. In CSP, they are just channels. Here, we use these terms indistinguishably.

Definition 3.1 (Component contract). A component contract Ctr comprises an observational behaviour $\mathcal{B}$, a set of communication channels $\mathcal{C}$, a set of types $\mathcal{I}$, that stands for the sets of values communicated by the channels, and a function $\mathcal{R} : \mathcal{C} \to \mathcal{I}$ between channels and types ($Ctr : \langle \mathcal{B}, \mathcal{R}, \mathcal{I}, \mathcal{C} \rangle$), such that $\mathcal{B}$ is an I/O process (see Definition 3.2), dom $\mathcal{R} = \mathcal{C}$, and ran $\mathcal{R} = \mathcal{I}$.

We use $\mathcal{B}_{Ctr}$, $\mathcal{R}_{Ctr}$, $\mathcal{I}_{Ctr}$ and $\mathcal{C}_{Ctr}$ to denote the elements of the contract Ctr.

In our example, the contracts of a fork component and of a philosopher component are defined as follows.

$$Ctr_{FORK} = \langle\ \texttt{FORK}, \{\texttt{fk1} \mapsto \texttt{I\_FORK}, \texttt{fk2} \mapsto \texttt{I\_FORK}\}, \{\texttt{I\_FORK}\}, \{\texttt{fk1}, \texttt{fk2}\}\ \rangle$$
$$Ctr_{PHIL} = \langle\ \texttt{PHIL}, \{\texttt{lf} \mapsto \texttt{I\_LIFE}, \texttt{pf1} \mapsto \texttt{I\_PHFK}, \texttt{pf2} \mapsto \texttt{I\_PHFK}\}, \{\texttt{I\_PHFK}, \texttt{I\_LIFE}\}, \{\texttt{lf}, \texttt{pf1}, \texttt{pf2}\}\ \rangle$$

The behaviour of the fork component is defined by the process `FORK` (see Figure 1). This contract has two channels, `fk1` and `fk2`, both of type `I_FORK`. Similarly, the behaviour of the philosopher component is defined by the process `PHIL` (also in Figure 1). This contract, however, has three channels, `pf1`, `fk2`, both of type `I_PHFK`, and `lf` of type `I_LIFE`.

We follow approaches like that of [All97], in which component models have a higher-level granularity by complementing the syntactic information of a component with behaviour. In our case, we explicitly distinguish inputs and outputs. The behaviour of our components are represented by I/O processes, which are defined as follows.

Definition 3.2 (I/O Process). An I/O process is a CSP process P that satisfies five conditions:

(i) **I/O Channels.** Every event in P is either an input or an output. Formally, we say a channel c is an I/O channel if, for a process P:

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

where $\{\!|\ c\ |\!\}$ returns the set of all events on c, and $inputs(c, P)$ and $outputs(c, P)$ return all input and output events on c in process P, respectively.
The functions $inputs_P$ and $outputs_P$ returns all input events and all output events of a given process, respectively. In our example, the definitions of *inputs* and *outputs* are such that:

$$inputs_P(FORK) = \{\texttt{fk1.picksup}, \texttt{fk1.putsdown}, , \texttt{fk2.picksup}, \texttt{fk2.putsdown}\}$$
$$outputs_P(FORK) = \{\texttt{fk1.picksack}, \texttt{fk1.putsack}, \texttt{fk2.picksack}, \texttt{fk2.putsack}\}$$
$$inputs_P(PHIL) = \{\texttt{pf1.picksack}, \texttt{pf1.putsack}, \texttt{pf2.picksack}, \texttt{pf2.putsack}\}$$
$$outputs_P(PHIL) = \left\{ \begin{array}{l} \texttt{pf1.picksup}, \texttt{pf1.putsdown}, \texttt{pf2.picksup}, \texttt{pf2.putsdown}, \\ \texttt{lf.getsup}, \texttt{lf.eats}, \texttt{lf.sits}, \texttt{lf.getsup} \end{array} \right\}$$

Hence, no event is both an input and an output of `FORK` or `PHIL`.

(ii) **Non-terminating.** P is a non-terminating process (we consider, however, for practical purposes in model checking, that these processes have finite state-spaces). The processes `FORK` and `PHIL` are defined as infinite loops and also satisfy this condition.

(iii) **Divergence-freedom.** P is divergence-free. Clearly, since `FORK` and `PHIL` do not use any hiding nor infinite traces of internal events, they are divergence-free.

(iv) **Input Determinism.** If a set of input events in P is offered by the environment, none of them are refused by P. Formally, P is input deterministic if:

$$\forall\, e : \Sigma;\ s : \mathrm{seq}\,\Sigma \mid s \frown \langle e \rangle \in traces(P) \land e \in inputs_P(P) \bullet (s, \{e\}) \notin failures(P)$$

Here, seq stands for the sequence type constructor. The processes `FORK` and `PHIL` are deterministic processes. Consequently, they are input deterministic processes.

(v) **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, a process is strong output decisive if:

$$\forall\, e : \Sigma;\ s : \mathrm{seq}\,\Sigma \mid s \frown \langle e \rangle \in \mathit{traces}(P) \land e \in \mathit{outputs}_P(P)$$
$$\bullet\ (\exists\, c : \mathit{CHANNEL} \mid e \in \mathit{outputs}(c, P)$$
$$\bullet\ (s, \mathit{outputs}(c, P)) \notin \mathit{failures}(P) \land (s, \mathit{outputs}(c, P) \setminus \{e\}) \in \mathit{failures}(P))$$

Besides, in the definition above, we state output decisiveness based only on maximum failures (all outputs within c except e – represented by $(s, \mathit{outputs}(c, P) \setminus \{e\})$); the other failures (whose refusals are subsets of these ones) are implicit in the definition. According to the failures theory, $\forall\, X \mid X \subseteq (s, \mathit{outputs}(c, P) \setminus \{e\}) \bullet (s, X) \in \mathit{failures}(P)$. Our strong output decisiveness notion is similar to the equally named property from [Ros05], if we consider that channels communicate exclusively either inputs or outputs. Our notion, however, forbids all outputs on that channel to be refused at the same time ($(s, \mathit{outputs}(c, P)) \notin \mathit{failures}(P)$). In addition to being deterministic processes, `FORK` and `PHIL` are trivially strong output decisive processes because there are no choices on outputs.

As one of the contributions of this paper, in Section 5, we describe how these conditions can be encoded as CSP assertions that can be automatically verified using FDR.

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

Definition 3.3 (Component contract instantiation). Let Ctr be a component contract, and F a bijection between name prefixes, such that dom $F = \mathcal{C}_{\mathit{Ctr}}$ and $F(c)$ has the form $c.x...y$, where $x...y$ is a sequence of one or more integers combined by dots, playing the roles of indices. Then the instance of Ctr according to F, denoted by $\mathit{Comp}_{\mathit{INST}}(\mathit{Ctr}, F)$, is defined as follows.

$$\mathit{Comp}_{\mathit{INST}}(\mathit{Ctr}, F) = \langle \mathcal{B}_{\mathit{Ctr}}\ [\![F]\!], F^{-1};\ \mathcal{R}_{\mathit{Ctr}}, \mathcal{I}_{\mathit{Ctr}}, \mathrm{ran}\, F \rangle$$

In the definition above, all elements of the new component are derived from the bijection and from the older component contract. The behaviour of the new component contract is equivalent to the older one, except that it uses a new set of interaction points (the range of the bijection). The interfaces are still the same, and mapping from interaction points into interfaces are represented by a composition of the bijection (the inverse function F^{-1}) with the relation $\mathcal{R}$. Intuitively, all protocols of the new component are equivalent to the protocols of the original one. The interaction point takes the form $c.x...y$, in which c is a channel and $x..y$ are identifiers of components; usually one of these identifiers is assigned to the component itself and the others to components that it interacts with.

We represent each fork and philosopher behaviour by a process `FORKi` or `PHILi`, where $i \in \{1..3\}$, to ensure no fork can be held by two philosophers at the same time. The channels used by these components are `fk`, `pfk`, and `life`. In order to distinguish actions from each philosopher on each fork, there are two integers on the type of channels `fk` and `pfk` standing for the fork and for the philosopher. The channel `life` type represents the philosopher identifier.

```
channel fk: RANGE.RANGE.I_FORK
channel pfk: RANGE.RANGE.I_PHFK
channel life: RANGE.I_LIFE
```

The contracts of the three forks and philosophers are defined using the component instantiation, which returns a contract that is similar to the given contract but replaces occurrences of the channels in the domain of the given mapping by the channels they are mapped to.

$$\mathit{FORK}1 = \mathit{Comp}_{\mathit{INST}}(\mathit{Ctr}_{\mathit{FORK}}, \{\texttt{fk1} \mapsto \texttt{fk.1.1}, \texttt{fk2} \mapsto \texttt{fk.1.3}\})$$
$$\mathit{FORK}2 = \mathit{Comp}_{\mathit{INST}}(\mathit{Ctr}_{\mathit{FORK}}, \{\texttt{fk1} \mapsto \texttt{fk.2.2}, \texttt{fk2} \mapsto \texttt{fk.2.1}\})$$
$$\mathit{FORK}3 = \mathit{Comp}_{\mathit{INST}}(\mathit{Ctr}_{\mathit{FORK}}, \{\texttt{fk1} \mapsto \texttt{fk.3.3}, \texttt{fk2} \mapsto \texttt{fk.3.2}\})$$

$$\mathit{PHIL}1 = \mathit{Comp}_{\mathit{INST}}(\mathit{Ctr}_{\mathit{PHIL}}, \{\texttt{pf1} \mapsto \texttt{pfk.1.1}, \texttt{fk2} \mapsto \texttt{pfk.2.1}, \texttt{lf} \mapsto \texttt{life.1}\})$$
$$\mathit{PHIL}2 = \mathit{Comp}_{\mathit{INST}}(\mathit{Ctr}_{\mathit{PHIL}}, \{\texttt{pf1} \mapsto \texttt{pfk.2.2}, \texttt{fk2} \mapsto \texttt{pfk.3.2}, \texttt{lf} \mapsto \texttt{life.2}\})$$
$$\mathit{PHIL}3 = \mathit{Comp}_{\mathit{INST}}(\mathit{Ctr}_{\mathit{PHIL}}, \{\texttt{pf1} \mapsto \texttt{pfk.3.3}, \texttt{fk2} \mapsto \texttt{pfk.1.3}, \texttt{lf} \mapsto \texttt{life.3}\})$$

For example, the resulting `FORK1` contract is:

$$FORK1 = \left\langle \begin{array}{l} \texttt{FORK [[fk1 <- fk.1.1, fk2 <- fk.1.3]]}, \\ \{\texttt{fk.1.1} \mapsto \texttt{I\_FORK}, \texttt{fk.1.3} \mapsto \texttt{I\_FORK}\}, \{\texttt{I\_FORK}\}, \{\texttt{fk.1.1}, \texttt{fk.1.3}\} \end{array} \right\rangle$$

Although all forks and philosophers are represented by one process with indices on its channels, there is a separate definition for each component contract.

Observe that a component contract (or its instantiation) does not explicitly define the protocols that it satisfies. These protocols are usually an information given at deployment, used before component integration. In case a protocol is not provided at the integration phase, we automatically derive it from the component behaviour, since a protocol is in fact a specification that a component realises during the communication with other components.

Communication protocols are commonly associated to specifications of component behaviours at a specific abstraction level, with an exclusive focus on a portion of the communicated events. Despite its relation with components, these protocols are also studied apart to analyse the communications through a channel. For homogeneity, we consider communication protocols as regular I/O processes. However, we show the relationship between protocols and more complex I/O processes, which encompass such communication protocols.

Definition 3.4 (Communication protocol). We define an I/O process P as a communication protocol if $\exists\, c_1, c_2 \bullet \mathit{inputs}_P(P) \subseteq \{\!|\ c_1\ |\!\} \wedge \mathit{outputs}_P(P) \subseteq \{\!|\ c_2\ |\!\}$.

The definition above says that a communication protocol is an I/O process that inputs solely by a unique channel (c_1, for instance) and outputs solely by a unique channel (c_2, for instance). These channels can be the same, or be distinct channels (one channel for each direction). Note that any protocol that inputs and outputs via one channel has an isomorphic counterpart with two distinct channels, and vice-versa.

Based on the definition of communication protocols, we have two related and distinct concepts: implementing a protocol (Protocol Implementation) and satisfying a protocol (Protocol Satisfaction).

The function $\mathit{Prot}_{IMP}(\mathit{Ctr}, \mathit{ch})$ takes an I/O process (in our case the component behaviour) and a channel, and returns the protocol associated to them. We call protocol, or communication protocol, the protocol associated with a communication channel of a component.

Definition 3.5 (Protocol implementation). Let P be an I/O process, and ch a communication channel. The communication protocol, namely $\mathit{Prot}_{IMP}(P, ch)$, implemented by P over ch is a protocol that satisfies the following property:

$$\mathit{Prot}_{IMP}(P, ch) \equiv_{\mathrm{F}} P \upharpoonright \{ch\}$$

where the projection $P \upharpoonright C$ is defined as follows.

Definition 3.6 (Projection). Let P be a process, and C a set of communication channels. The *projection* of P over C (denoted by $P \upharpoonright C$) is defined as: $P \upharpoonright C = P \setminus (\Sigma \setminus \{\!|\ C\ |\!\})$

By way of illustration, the processes below represent parameterised protocols of forks and philosophers on a channel (port) `c` (`PROT_PH(c)`). Observe that a protocol is equivalent to the projection of the entire dynamic behaviour of `FORK` and `PHIL` (see Figure 1) over the corresponding port.

```
PROT_FK(c) = COMPFK(c); PROT_FK(c)
PROT_PH(c) = PICKUP_PH(c); PUTDOWN_PH(c); PROT_PH(c)
```

Based on these definitions, we may, for example, define the protocols of the $FORK1$ and the $PHIL1$, on `fk.1.1` and `pfk.1.1`, respectively, as follows:

$\mathit{Prot}_{IMP}(\mathcal{B}_{FORK1}, \texttt{fk.1.1}) = \texttt{PROT\_FK(fk.1.1)}$
$\mathit{Prot}_{IMP}(\mathcal{B}_{PHIL1}, \texttt{pfk.1.1}) = \texttt{PROT\_PH(pfk.1.1)}$

Generalising the definition above, we now define whether a process satisfies a protocol. Such a protocol is a given behaviour that satisfies its communication over a channel.

Definition 3.7 (Protocol satisfaction). Let P be an I/O process, ch a communication channel, and Q a communication protocol. The communication of P over ch satisfies the protocol Q, if, and only, if $Q \sqsubseteq_{\mathrm{F}} \mathit{Prot}_{IMP}(P, ch)$.

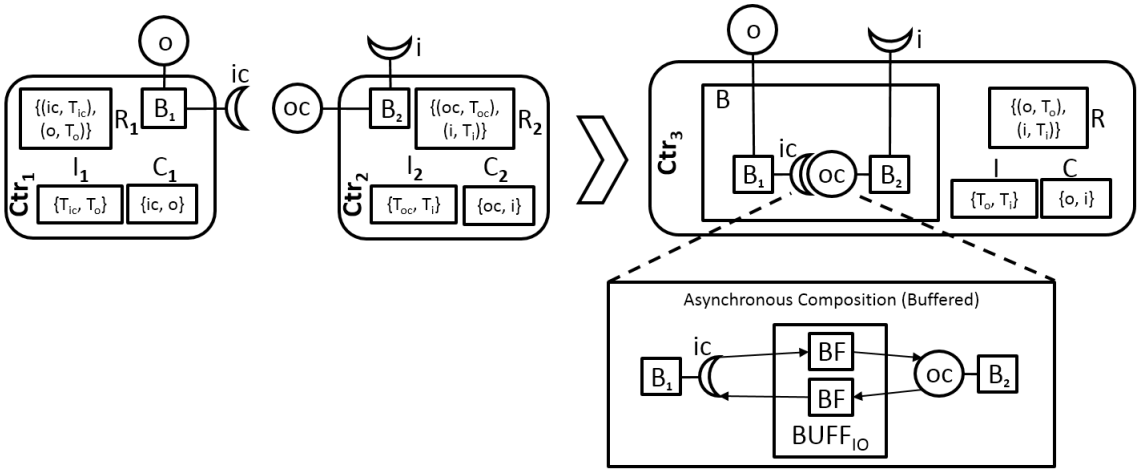


Fig. 3. Asynchronous Binary Composition ($Ctr_1 \langle ic \rangle \asymp \langle oc \rangle\ Ctr_2$)

Protocol satisfaction is an important notion in some architectures, since they may have a restriction about the communication protocols a process must satisfy in order to communicate with others.

Based on the work presented in [Ros05] on buffer tolerance, we intentionally adopt an asynchronous communication model in component interactions, which allows us to analyse systems with asynchronous communications. To represent asynchronous communication, we introduce buffers as intermediate elements of the composition. They copy information from one component channel to another. Information is always accepted, despite the other component being ready or not being ready to input. These buffers are not first-class elements, but are implicit to the component model. Furthermore, we consider infinite buffers here, as they represent the worst case scenario we can find. Nevertheless, in [RSM09] we demonstrate that all the important properties are preserved in the presence of finite buffers, which avoid the state explosion on model checkers and are used in our experiments.

Each composition creates a new component as a result, which includes the original ones. We define component composition in two modes: a binary operation on two components, and a unary operation over a single component. To help us in this definition, we first specify an auxiliary function *AsyncComp* that takes a set of processes S and a bijective function F among distinct sets of channels used by processes within S and yields the assembly of the processes within S, connecting each channel c to its respective channel represented by $F(c)$.

$$AsyncComp(S, F) = \left(|||_{P \in S}\, P \right) \parallel_{\mathrm{dom}\ F \cup \mathrm{ran}\ F} \left(|||_{c \in \mathrm{dom}\ F}\, BUFF^{\infty}_{IO}(c, F(c)) \right)$$

where $BUFF^{\infty}_{IO}(c, z)$ is an infinite buffer that copies information from c to z, and vice-versa. All formal definitions related to the buffers used in this paper can be found in Appendix A.

The binary composition operator $Ctr_1 {}_s \asymp {}_t Ctr_2$ provides an asynchronous interaction between components Ctr_1 and Ctr_2, mediated by infinite buffers, on the corresponding channels from two equally sized lists s and t, respectively. The binary composition rules of our approach, defined and illustrated later in this section, are based on this asynchronous binary composition.

Definition 3.8 (Asynchronous binary composition). Let Ctr_1 and Ctr_2 be two distinct component contracts, and $\langle c_1, .., c_n \rangle$ and $\langle z_1, .., z_n \rangle$ sequences of distinct channels within $\mathcal{C}_{Ctr_1}$ and $\mathcal{C}_{Ctr_2}$, respectively, such that $\mathcal{C}_{Ctr_1} \cap \mathcal{C}_{Ctr_2} = \emptyset$. Then, the asynchronous binary composition of Ctr_1 and Ctr_2 is given by:

$$Ctr_1 \langle c_1, .., c_n \rangle \asymp \langle z_1, .., z_n \rangle\ Ctr_2 = \\ \langle (AsyncComp(\{\mathcal{B}_{Ctr_1}, \mathcal{B}_{Ctr_2}\}, \{c_i \mapsto z_i \mid i \in 1..n\}), \mathcal{R}_{Ctr_3}, \mathcal{I}_{Ctr_3}, \mathcal{C}_{Ctr_3} \rangle$$

where $\mathcal{C}_{Ctr_3} = (\mathcal{C}_{Ctr_1} \cup \mathcal{C}_{Ctr_2}) \setminus \{c_1, .., c_n, z_1, .., z_n\}$, $\mathcal{R}_{Ctr_3} = \mathcal{C}_{Ctr_3} \lhd (\mathcal{R}_{Ctr_1} \cup \mathcal{R}_{Ctr_2})$, and $\mathcal{I}_{Ctr_3} = \mathrm{ran}\, \mathcal{R}_{Ctr_3}$.

In this definition, we assume each component has a distinct set of interaction points ($\mathcal{C}_{Ctr_1} \cap \mathcal{C}_{Ctr_2} = \emptyset$), and that their communication is asynchronous, mediated by buffers. The behaviour of the composition is

defined by the synchronisation of the components (Ctr_1 or Ctr_2) with an infinite buffer in all interactions of the channels mapped by F (this is expressed by the process *AsyncComp* described above). Any communication related to a channel mapped by F is not offered to the environment in further compositions ($\mathcal{C}_{Ctr_3}$). The operator $\lhd$ stands for domain restriction; it is used to restrict the mapping from channels into interfaces ($\mathcal{R}_{Ctr_3}$). It is important to observe that the sequences $\langle c_1, .., c_n\rangle$ and $\langle z_1, .., z_n\rangle$ have distinct channels; this helps to avoid that components share the same interaction points (channels).

By way of illustration, let us consider the two component contracts defined below:

$$Ctr_1 \mathrel{\widehat{=}} \langle B_1, \{ic \mapsto T_{ic}, o \mapsto T_o\}, \{T_{ic}, T_o\}, \{ic, o\}\rangle$$
$$Ctr_2 \mathrel{\widehat{=}} \langle B_2, \{oc \mapsto T_{oc}, i \mapsto T_i\}, \{T_{oc}, T_i\}, \{oc, i\}\rangle$$

The contract Ctr_1 has a CSP behaviour B_1 and two channels: ic of type T_{ic} and o of type T_o. The contract Ctr_2 has a similar definition. In Figure 3, we illustrate the composition of these two contracts on channels ic and oc (denoted as $Ctr_{1\langle ic\rangle} \asymp_{\langle oc\rangle} Ctr_2$). The resulting contract, Ctr_3, composes both channels asynchronously using one infinite buffer (depicted as BF in Figure 3) for each direction.

The binary composition behaves similarly to the piping (or chaining) CSP operator [Ros98]. However, it does not oblige pipelines of assembled processes to have the same interface type; the channels used in the composition are explicitly defined. Moreover, the buffer explicitly represents the bindings of different interface types, rather than the implicit renaming performed by such CSP operator.

Below we show how the composition can be used to construct well-formed component contracts (elements that satisfy the conditions to be a component contract in our component model).

Theorem 3.1 (Binary Composition Monotonicity). Let Ctr_1 and Ctr_2 be component contracts, and $\langle c_1, .., c_n\rangle$ and $\langle z_1, .., z_n\rangle$ sequences of distinct channels within $\mathcal{C}_{Ctr_1}$ and $\mathcal{C}_{Ctr_2}$, respectively, such that the behaviour ($\mathcal{B}$) of $Ctr_{1\langle c_1,..,c_n\rangle} \asymp_{\langle z_1,..,z_n\rangle} Ctr_2$ is deadlock-free. Then $Ctr_{1\langle c_1,..,c_n\rangle} \asymp_{\langle z_1,..,z_n\rangle} Ctr_2$ is a component contract.

Proof. In order to prove that the resulting tuple is a component contract, we have to show that its structure is compatible with Definition 3.1. This is provided by the use of the direct composition (see Definition 3.8), $P \,[\!|||]\, Q = P_{\langle\rangle} \asymp_{\langle\rangle} Q$. Moreover, the behaviour of the component is also an I/O process, since the composition does not introduce divergences (no hiding operation or undesired renaming is performed), the infinite behaviours of the original components result in a new infinite process, and the resulting process is input deterministic and output decisive with respect to the channels that remain in the contract (see theorems B.1 and B.2 in Appendix B). □

Unary compositions $Ctr \asymp\big|_s^t$ are needed when we want to assemble inner channels from two channel lists s and t of a single component Ctr.

Definition 3.9 (Asynchronous unary composition). Let Ctr be a component contract, and $\langle c_1, .., c_n\rangle$ and $\langle z_1, .., z_n\rangle$ sequences of distinct channels within $\mathcal{C}_{Ctr}$, such that $\{c_1, .., c_n\} \cap \{z_1, .., z_n\} = \emptyset$. Then, the asynchronous unary composition of Ctr is given by:

$$P \asymp\big|_{\langle z_1,..,z_n\rangle}^{\langle c_1,..,c_n\rangle} = \langle (AsyncComp(\{\mathcal{B}_{Ctr}\}, \{c_i \mapsto z_i \mid i \in 1..n\}), \mathcal{R}_{Ctr'}, \mathcal{I}_{Ctr'}, \mathcal{C}_{Ctr'}\rangle$$

where $\mathcal{C}_{Ctr'} = \mathcal{C}_{Ctr} \setminus \{c_1, .., c_n, z_1, .., z_n\}$, $\mathcal{R}_{Ctr'} = \mathcal{C}_{Ctr'} \lhd \mathcal{R}_{Ctr}$, and $\mathcal{I}_{Ctr'} = \operatorname{ran} \mathcal{R}_{Ctr'}$.

The definition above is similar to the one for binary composition. It differs on the number of processes passed to *AsyncComp*. This allows us to assemble channels of a same component, instead of two distinct components. Similar to the binary compositions, we are able to show how the composition can be used to construct well-formed component contracts (elements that satisfy the conditions to be a component contract in our component model); the corresponding theorem and proof are omitted as they are similar to the binary case.

We propose four composition rules: *interleave*, *communication*, *feedback* and *reflexive* compositions. Each one focuses on ensuring deadlock-freedom in a specific interaction pattern. Using the rules, developers may asynchronously assemble two channels of two components, or even of the same component. The first three rules have already been presented in [RSM09] (although the proof that they preserve deadlock freedom is a contribution of the current paper) and focus on the local analysis of *protocols*, which are the projections of the contract's behaviour on each port (channel). The *reflexive composition* is novel: it is a generalisation of the *feedback composition*. Both feedback and reflexive are unary compositions (they allow connecting channels

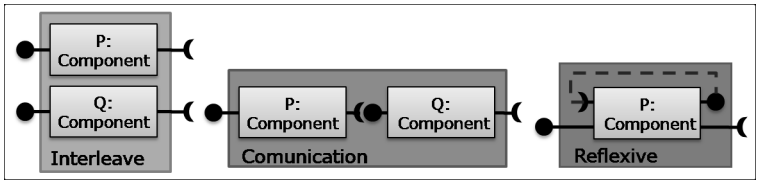


Fig. 4. Three Compositions Rules

of a same component). Feedback composition imposes strong restrictions to allow local analysis. Reflexive composition represents the general case where an overall analysis of the entire component is necessary, rather than a local analysis. The complexity of this analysis served as motivation for us to explore adherence of components to some behavioural patterns, which requires only local analysis. We illustrate three of these composition rules in Figure 4. Feedback composition is not present in the figure as it has the same graphical format as reflexive composition. Each of the four rules is detailed in the sequel.

The interleave composition rule aggregates two independent entities such that, after composition, these entities still do not interact. 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 3.10 (Interleave composition). Let Ctr_1 and Ctr_2 be two component contracts with disjoint channels, and $\mathcal{C}_{Ctr_1} \cap \mathcal{C}_{Ctr_2} = \emptyset$. The interleave composition of Ctr_1 and Ctr_2 is given by:

$$Ctr_1 \,[|||]\, Ctr_2 = Ctr_1 \,{}_{\langle\rangle} \asymp_{\langle\rangle}\, Ctr_2$$

Implicitly, Ctr_1 and Ctr_2 are assumed to be deadlock-free because their component behaviours are I/O processes. The interleave composition is a particular case of binary composition with an empty sequence of connecting channels.

The entire system of our example is formed of composing all the basic component contracts in parallel ($FORK1$ to $FORK3$ and $PHIL1$ to $PHIL3$). We can build this system in several steps, using the composition rules presented here. First, philosophers and forks can be interleaved separately as presented below, until we reach the configuration presented in Figure 5. For simplicity, we consider only 3 philosophers and 3 forks to illustrate the approach.

$$FORKS\_1\_2 = FORK1 \,[|||]\, FORK2$$
$$FORKS = FORKS\_1\_2 \,[|||]\, FORK3$$

$$PHILS\_1\_2 = PHIL1 \,[|||]\, PHIL2$$
$$PHILS = PHILS\_1\_2 \,[|||]\, PHIL3$$

This is clearly a valid composition since the contracts have disjoint channels.

The above composition form is, by definition, a particular kind of direct composition that involves no communication, resulting in a weakly cohesive entity, which performs all events defined in the original entities without any interference from each other.

Theorem 3.2 (Deadlock-free Interleave Composition). The *interleave composition* of two deadlock-free component contracts is also a deadlock-free component contract.

Proof. The statement that the resulting process is deadlock-free follows directly from the condition that the components do not share any channel. Furthermore, as showed below, $\mathcal{B}_{Ctr_1\,[|||]\,Ctr_2} = \mathcal{B}_{Ctr_1} \,|||\, \mathcal{B}_{Ctr_2}$. As $\mathcal{B}_{Ctr_1}$ and $\mathcal{B}_{Ctr_2}$ are deadlock-free, so is, from the semantics of the interleave operator, $\mathcal{B}_{Ctr_1\,[|||]\,Ctr_2}$. We start by expanding the direct composition of Ctr_1 and Ctr_2 (Definition 3.8).

$$\begin{aligned}
&\mathcal{B}_{Ctr_1\,[|||]\,Ctr_2} \\
&\equiv_{\mathrm{F}} (\mathcal{B}_{Ctr_1} \,|||\, \mathcal{B}_{Ctr_2}) \,[\![\, \{\} \,]\!]\, \left(|||_{c \in \{\}} BUFF^{\infty}_{IO}(c, F(c)) \right) && \text{[no buffer is used, } c \in \{\}\text{]} \\
&\equiv_{\mathrm{F}} (\mathcal{B}_{Ctr_1} \,|||\, \mathcal{B}_{Ctr_2}) \,[\![\, \{\} \,]\!]\, SKIP && \text{[rewriting]} \\
&\equiv_{\mathrm{F}} \mathcal{B}_{Ctr_1} \,|||\, \mathcal{B}_{Ctr_2} && \text{[CSP law]}
\end{aligned}$$

The proof that the resulting contract of this composition rule is a component contract follows directly from Theorem 3.1. □

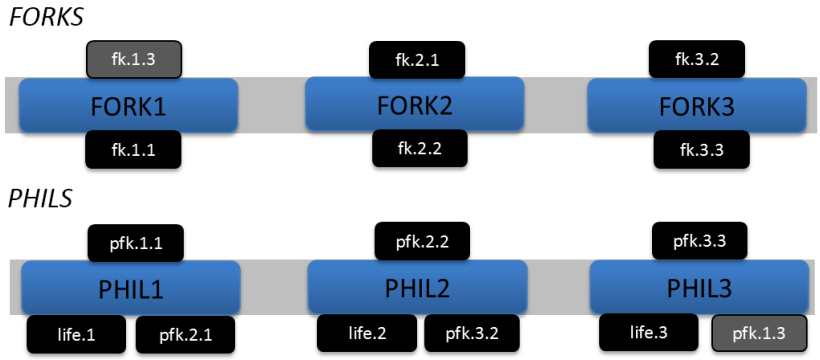


Fig. 5. Interleaving of Philosophers and Forks

In the composition rules that follow, the lists of connecting channels s and t do not have elements in common. For this reason, the intermediary buffer has to map outputs from s into inputs from t, and vice-versa. The conditions used to validate the composition takes this fact into consideration using a renaming function: $P \,[\![\, R_{IO}^{c_1 \to c_2}]\!]$ replaces, in process P, references to outputs on c_1 by references to c_2.

$$P \,[\![\, R_{IO}^{c_1 \to c_2}]\!] = P[\{c_1.x \mapsto c_2.x \mid c_1.x \in outputs_P(P)\}]$$

$$inputs_P(P \,[\![\, R_{IO}^{c_1 \to c_2}]\!]) = inputs_P(P)$$
$$outputs_P(P \,[\![\, R_{IO}^{c_1 \to c_2}]\!]) = outputs_P(P)[c_1 \mapsto c_2]$$

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

The next composition rule needs the properties below.

(vi) [**I/O Confluence**] Whenever a state has two alternative actions e_1 and e_2, then performing either of them does not preclude the other, unless it is a choice among inputs or outputs of the same channel. Formally, an I/O process P is I/O confluent if, and only if:

$$\begin{array}{l} \forall\, e_1, e_2 : \Sigma;\ \forall\, s, t : \operatorname{seq} \Sigma \\ \quad \mid e_1 \neq e_2 \wedge \{s \frown \langle e_1 \rangle \frown t, s \frown \langle e_2 \rangle\} \subseteq traces(P) \\ \quad \bullet \left(\begin{array}{l} e_1 \in inputs_P(P) \\ \wedge\, \exists\, i : inputs(channel(e_1), P) \bullet s \frown \langle e_2, i \rangle \frown (t - \langle e_2 \rangle) \in traces(P) \end{array} \right) \\ \qquad \vee \left(\begin{array}{l} e_1 \in outputs_P(P) \\ \wedge\, \exists\, o : outputs(channel(e_1), P) \bullet s \frown \langle e_2, o \rangle \frown (t - \langle e_2 \rangle) \in traces(P)) \end{array} \right) \\ \qquad \vee \left(\begin{array}{l} channel(e_1) = channel(e_2) \\ \wedge\, \{e_1, e_2\} \subseteq outputs_P(P) \vee \{e_1, e_2\} \subseteq inputs_P(P) \end{array} \right) \end{array}$$

where the function $channel(e)$ returns the communication channel of a given event e.
The protocol `PROT_FK(fk.1.1)` is an example of an I/O confluent process. It is categorised in the simplest case, in which there is no choice among events of different channels, neither choices among inputs and outputs.

(vii) [**Finite Output Property**] A process P cannot perform an infinite number of outputs without an input. As I/O processes are divergence-free, the absence of divergence after hiding the outputs guarantees this property. Formally, an I/O processes satisfies the Finite Output Property if, and only if:

$$divergences(P \setminus outputs_P(P)) = \emptyset$$

In our example, `FORK`, `PHIL`, and their protocols satisfy this property.

(viii) [**Conjugate Protocols**] Two communication protocols P and Q are conjugate if, and only if, the outputs of P are understood as inputs of Q, and vice-versa:

$$\begin{array}{l} outputs_P(P) \subseteq inputs_P(Q) \wedge outputs_P(Q) \subseteq inputs_P(P) \\ \wedge\, outputs_P(P) \cap outputs_P(Q) = \emptyset \wedge inputs_P(P) \cap inputs_P(Q) = \emptyset \end{array}$$

(ix) [**Strong Compatibility**] There must always be an output event to be performed and the outputs of

each process must be accepted by the other, in all scenarios. Formally, two deadlock-free conjugate communication protocols P and Q are strong compatible (denoted $P \approx\!\!\!\approx Q$) if, and only if:

$$\forall\, s : traces(P) \cap traces(Q) \bullet (O_P^s \neq \emptyset \vee O_Q^s \neq \emptyset) \wedge O_P^s \subseteq I_Q^s \wedge O_Q^s \subseteq I_Q^s$$

where I_P^s and O_P^s stands for the inputs and outputs performed by a process P after a trace s, respectively.

$$I_P^s = \{a : inputs_P(P) \mid s \frown \langle a \rangle \in traces(P)\}$$
$$O_P^s = \{a : outputs_P(P) \mid s \frown \langle a \rangle \in traces(P)\}$$

The first two properties deal with buffering concerns and allow mechanical verification on the system without state explosion [Ros05]. The third and forth properties guarantee the interoperability of the two components.

By way of illustration, the protocols $Prot_{IMP}(\mathcal{B}_{FORKS}, \texttt{fk.1.1})$ and $Prot_{IMP}(\mathcal{B}_{PHILS}, \texttt{pfk.1.1})$ previously presented are not conjugate protocols (hence, they are not strong compatible) because they clearly work on completely different channels. However, we may use the renaming function R_{IO} as follows to make both protocols strong compatible.

$$Prot_{IMP}(\mathcal{B}_{FORKS}, \texttt{fk.1.1}) \ [\![R_{IO}^{fk.1.1 \to pfk.1.1}]\!]$$
$$Prot_{IMP}(\mathcal{B}_{PHILS}, \texttt{pfk.1.1}) \ [\![R_{IO}^{pfk.1.1 \to fk.1.1}]\!]$$

To demonstrate this, using the definitions previously presented, we may calculate the following sets of inputs and outputs of the renamed protocols:

$$inputs_P(Prot_{IMP}(\mathcal{B}_{FORKS}, \texttt{fk.1.1}) \ [\![R_{IO}^{fk.1.1 \to pfk.1.1}]\!]) = \{\texttt{fk.1.1.picksup}, \texttt{fk.1.1.putsdown}\}$$
$$outputs_P(Prot_{IMP}(\mathcal{B}_{FORKS}, \texttt{fk.1.1}) \ [\![R_{IO}^{fk.1.1 \to pfk.1.1}]\!]) = \{\texttt{pfk.1.1.picksack}, \texttt{pfk.1.1.putsack}\}$$

$$inputs_P(Prot_{IMP}(\mathcal{B}_{PHILS}, \texttt{pfk.1.1}) \ [\![R_{IO}^{pfk.1.1 \to fk.1.1}]\!]) = \{\texttt{pfk.1.1.picksack}, \texttt{pfk.1.1.putsack}\}$$
$$outputs_P(Prot_{IMP}(\mathcal{B}_{PHILS}, \texttt{pfk.1.1}) \ [\![R_{IO}^{pfk.1.1 \to fk.1.1}]\!]) = \{\texttt{fk.1.1.picksup}, \texttt{fk.1.1.putsdown}\}$$

We demonstrate that the protocols are strong compatible because they satisfy the conditions of strong compatibility stated above.

Before formalising the verification of *Strong Compatibility*, we define an auxiliary notion: the dual protocol of P, $DProt(P)$, is a protocol with the same traces of P, but whose inputs are the outputs of P, and vice-versa. Formally:

Definition 3.11 (Dual protocol). Let P be a deadlock-free communication protocol. The dual protocol of P is defined as a deadlock-free communication protocol DP, such that:

$$inputs_P(P) = outputs_P(DP) \wedge outputs_P(P) = inputs_P(DP) \wedge traces(DP) = traces(P)$$

The formal verification of *Strong Compatibility* is characterised as assertions on simple failures refinement described in Section 2. In summary, two protocols P and Q are strong compatible if $DProt(P) \sqsubseteq_{\mathrm{F}} Q$.

The protocols of our example include no choice operators. For this reason, their dual protocols are the same, but the definitions of their *inputs* and *outputs* are interchanged. For example, we present below the definition of the dual protocol of the protocol implementation of $FORK1$ on `fk.1.1`.

$$DProt(Prot_{IMP}(\mathcal{B}_{FORK1}, \texttt{fk.1.1})) = Prot_{IMP}(\mathcal{B}_{FORK1}, \texttt{fk.1.1})$$
$$inputs_P(DProt(Prot_{IMP}(\mathcal{B}_{FORK1}, \texttt{fk.1.1}))) = outputs_P(Prot_{IMP}(\mathcal{B}_{FORK1}, \texttt{fk.1.1}))$$
$$outputs_P(DProt(Prot_{IMP}(\mathcal{B}_{FORK1}, \texttt{fk.1.1}))) = inputs_P(Prot_{IMP}(\mathcal{B}_{FORK1}, \texttt{fk.1.1}))$$

We are now able to present the second composition rule, *communication composition*.

Definition 3.12 (Communication composition). Let Ctr_1 and Ctr_2 be two component contracts, and ic and oc two channels, such that $ic \in \mathcal{C}_{Ctr_1} \wedge oc \in \mathcal{C}_{Ctr_2}$, $\mathcal{C}_{Ctr_1} \cap \mathcal{C}_{Ctr_2} = \emptyset$, and their port protocols $Prot_{IMP}(\mathcal{B}_{Ctr_1}, ic) \ [\![R_{IO}^{ic \to oc}]\!]$ and $Prot_{IMP}(\mathcal{B}_{Ctr_2}, oc) \ [\![R_{IO}^{oc \to ic}]\!]$ are I/O confluent strong compatible and satisfy the finite output property. The communication composition of Ctr_1 and Ctr_2 via ic and oc is defined as $Ctr_1[ic \leftrightarrow oc]Ctr_2 = Ctr_{1\,\langle ic \rangle} \asymp_{\langle oc \rangle} Ctr_2$.

Besides having disjoint channels, further restrictions apply to the divergence free process implementation protocols on ic and oc. They, however, apply to renamed versions of these protocols.

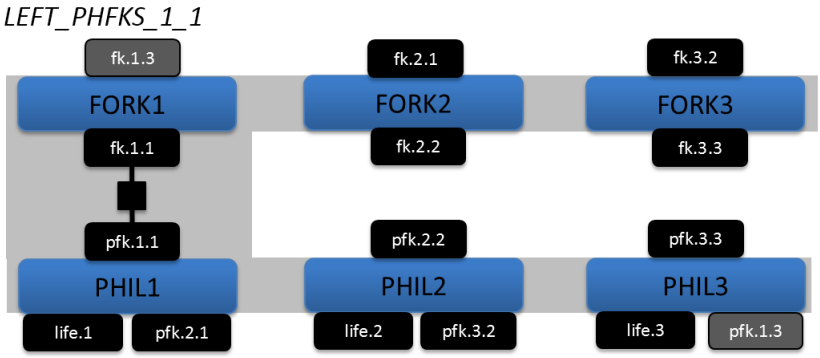


Fig. 6. Result of Communication Composition

In our example, we compose the resulting contracts *FORKS* and *PHILS* using the communication composition, yielding the configuration presented in Figure 6.

$$LEFT\_PHFKS\_1\_1 = FORKS[fk.1.1 \leftrightarrow pfk.1.1]PHILS$$

This composition is allowed because the contracts work on different channels and the renamed versions of their protocols $Prot_{IMP}(\mathcal{B}_{FORKS}, \texttt{fk.1.1}) \,[\![R_{IO}^{fk.1.1 \to pfk.1.1}]\!]$ and $Prot_{IMP}(\mathcal{B}_{PHILS}, \texttt{pfk.1.1}) \,[\![R_{IO}^{pfk.1.1 \to fk.1.1}]\!]$ are I/O confluent and strong compatible and satisfy the finite output property. The following theorem guarantees that deadlock-freedom is preserved by construction.

Theorem 3.3 (Deadlock-free Communication Composition). The *communication composition* of two deadlock-free component contracts is also a deadlock-free component contract.

Proof. The *communication composition* of two components Ctr_1 and Ctr_2 is formed, in fact, of two parallel synchronisations. The composition operator $\asymp$ implicitly introduces a buffer that always accepts any communication from Ctr_1, and forwards to Ctr_2, and vice-versa. As these components have strong compatible protocols, they always accept communications from each other. The proof of this theorem follows directly from Theorem B.4 (see Appendix B).

To prove that a divergent-free component Ctr is deadlock-free, we have to prove that $\mathcal{B}_{Ctr} \setminus \Sigma$ diverges [Ros98]. In the statements below we rewrite the behaviour of a composition of two component contracts Ctr_1 and Ctr_2 using the channels ic and oc.

$$\begin{aligned}
&\mathcal{B}_{Ctr_1[ic \leftrightarrow oc]Ctr_2} \setminus \Sigma \\
&= \left((\mathcal{B}_{Ctr_1} \,|||\, \mathcal{B}_{Ctr_2}) \,[\![\{\!| ic, oc |\!\}]\!] \left(\Big|||_{c \in \{ic\}} BUFF_{IO}^{\infty}(c, F(c)) \right) \right) \setminus \Sigma \\
&= ((\mathcal{B}_{Ctr_1} \,|||\, \mathcal{B}_{Ctr_2}) \,[\![\{\!| ic, oc |\!\}]\!] \, BUFF_{IO}^{\infty}(ic, oc)) \setminus \Sigma \\
&= (\mathcal{B}_{Ctr_1} \,[\![\{\!| ic |\!\}]\!] \, BUFF_{IO}^{\infty}(ic, oc) \,[\![\{\!| oc |\!\}]\!] \, \mathcal{B}_{Ctr_2}) \setminus \Sigma \\
&= (\mathcal{B}_{Ctr_1} \,[\![\{\!| ic |\!\}]\!] \, BUFF_{IO}^{\infty}(ic, oc) \,[\![\{\!| oc |\!\}]\!] \, \mathcal{B}_{Ctr_2}) \setminus (\Sigma \setminus \{ic, oc\}) \setminus \Sigma \\
&= (\mathcal{B}_{Ctr_1} \upharpoonright \{\!| ic, oc |\!\} \,[\![\{\!| ic |\!\}]\!] \, BUFF_{IO}^{\infty}(ic, oc) \upharpoonright \{\!| ic, oc |\!\} \,[\![\{\!| oc |\!\}]\!] \, \mathcal{B}_{Ctr_2} \upharpoonright \{\!| ic, oc |\!\}) \setminus \Sigma \\
&= (P \upharpoonright ic \,[\![\{\!| ic |\!\}]\!] \, BUFF_{IO}^{\infty}(ic, oc) \,[\![\{\!| oc |\!\}]\!] \, Q \upharpoonright oc) \setminus \Sigma
\end{aligned}$$

If either $Ctr_1 \upharpoonright ic$ or $Q \upharpoonright oc$ diverges, then the synchronisation above diverges, proving that the composition is deadlock-free. So, we continue the proof assuming that these projections are divergence-free.

$$\begin{aligned}
&\mathcal{B}_{Ctr_1[ic \leftrightarrow oc]Ctr_2} \setminus \Sigma \\
&= (Ctr_1 \upharpoonright ic \,[\![\{\!| ic |\!\}]\!] \, BUFF_{IO}^{\infty}(ic, oc) \,[\![\{\!| oc |\!\}]\!] \, Ctr_2 \upharpoonright oc) \setminus \Sigma && [Ctr_1 \upharpoonright ic \text{ and } Ctr_2 \upharpoonright oc \text{ are divergence-free}] \\
&= (Prot_{IMP}(P, ic) \,[\![\{\!| ic |\!\}]\!] \, BUFF_{IO}^{\infty}(ic, oc) \,[\![\{\!| oc |\!\}]\!] \, Prot_{IMP}(Q, oc)) \setminus \Sigma && [\text{Theorem B.4}] \\
&= \mathsf{div}
\end{aligned}$$

The proof that the resulting composition is a component contract follows directly from Theorem 3.1. □

The next two composition rules allow assembling two channels of the same component. Since we build systems by composing components pairwisely, a set of assembled components can always be taken as a

single large grain entity. In doing so, composing outputs and inputs of this large grain component allows introducing dependency cycles in the flattened structure.

In our example, the contract *LEFT_PHFKS*_1_1 depicted in Figure 6 encompasses all basic contracts (forks and philosophers) and their channels. Any further composition like, for instance, `fk.2.2` with `pfk.2.2` requires assembling two channels of the same component, *LEFT_PHFKS*_1_1.

This connection may introduce cycles of requests that have been dealt with in more general approaches to ensure deadlock-freedom [MW97, Ros98], which identify that deadlocks arise in complex (graph) topologies by the presence of undesirable cycles, called *cycles of ungranted requests* [Ros98]. This requires new conditions to be satisfied in order to preserve deadlock-freedom after the composition.

The third composition rule (which we call feedback composition) deals with pseudo cyclic topologies, which are behaviourally equivalent to systems with tree-topologies. It does have some cycles, but none of them introduces deadlocks. However, it cannot express all possible topologies. For this reason, verification on this topology is simpler than in arbitrary complex topologies. The feedback composition is aligned with the incremental nature of our strategy, dealing with one problem at a time. A rule (reflexive composition) for more complex topologies is presented in Definition 3.14.

The feedback composition rule requires the two linked channels to be decoupled.

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

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

This means that the projected behaviour of *P* over the channels *cs* must coincide with the interleaving of the projections of *P* over each channel *c* in *cs*. As an example, the channels `pfk.2.2` and `fk.2.2` are decoupled in *LEFT_PHFKS*_1_1 as they do not interfere in the behaviour of each other.

Definition 3.13 (Feedback composition). Let *Ctr* be a component contract, and *ic* and *oc* two channels, such that the port protocols $Prot_{IMP}(\mathcal{B}_{Ctr}, ic) \,[\![R_{IO}^{ic \to oc}]\!]$ and $Prot_{IMP}(\mathcal{B}_{Ctr}, oc) \,[\![R_{IO}^{oc \to ic}]\!]$ are I/O confluent strong compatible and satisfy the finite output property, and $\{ic, oc\} \subseteq \mathcal{C}_{Ctr}$ are decoupled in *Ctr*. The feedback composition is defined as $Ctr[oc \hookrightarrow ic] = Ctr \succ\!\!\mid_{\langle oc \rangle}^{\langle ic \rangle}$.

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.

In our example, since the channels `pfk.2.2` and `fk.2.2` are decoupled in *LEFT_PHFKS*_1_1, we may connect these channels using the feedback composition.

$$LEFT\_PHFKS = LEFT\_PHFKS\_1\_1[fk.2.2 \hookrightarrow pfk.2.2]$$

For the same reason, the feedback composition may also be used to connect `fk.3.3` to `pfk.3.3`, `fk.2.1` to `pfk.2.1`, and `fk.3.2` to `pfk.3.2` until we reach the configuration in Figure 7. Nevertheless, as we explain in the sequel, the channels `fk.1.3` and `pfk.1.3` are not decoupled and the use of the feedback composition is not valid.

Before presenting a theorem related to this kind of composition, we present the reflexive composition rule, which is more general. The proof of these two rules are related; in fact, as we show later, the feedback composition is a particular case of the reflexive composition.

The composition rules presented so far deal with systems with a tree topology. In practice, there are more complex systems with cycles of dependency. The last composition rule, reflexive composition, is more general than the feedback one. However, it is also more costly regarding verification. The reflexive composition requires that the projection on the two linked channels ($\mathcal{B}_{Ctr} \upharpoonright \{ic, oc\}$) satisfies the finite output property and is buffering self-injection compatible.

(xi) [**Buffering Self-injection Compatibility**] This notion of compatibility is very similar to the notion of strong compatibility presented on Page 16, except for the fact that we do not compare the communication between two simple processes (protocols) but the communication between events of the same process.
Let *P* be a deadlock-free I/O process, and *c* and *z* channels. Then $Pj = P \upharpoonright \{\!|\ c, z\ |\!\}$ is buffering self-injection compatible if, and only if:

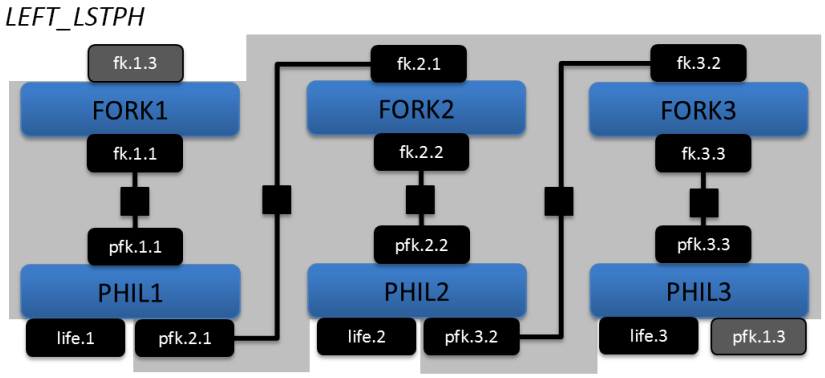


Fig. 7. Philosophers and their forks

- $\forall\, s : \operatorname{seq}\Sigma;\ X : \mathbb{P}\,\Sigma \bullet (s, X) : \mathit{failures}(Pj) \land s \downarrow O_c = s \downarrow I_z \land s \downarrow O_z = s \downarrow I_c \bullet X \cap (O_c \cup O_z) = \emptyset$
- $\forall\, s : \operatorname{seq}\Sigma;\ X : \mathbb{P}\,\Sigma \bullet (s, X) : \mathit{failures}(Pj) \land s \downarrow O_c > s \downarrow I_z \bullet (s \upharpoonright z, X \cup \{\!|\ c\ |\!\}) \in \mathit{failures}(Pj \upharpoonright z)$
- $\forall\, s : \operatorname{seq}\Sigma;\ X : \mathbb{P}\,\Sigma \bullet (s, X) : \mathit{failures}(Pj) \land s \downarrow O_z > s \downarrow I_c \bullet (s \upharpoonright c, X \cup \{\!|\ z\ |\!\}) \in \mathit{failures}(Pj \upharpoonright c)$

where $O_c = \mathit{outputs}(c, P)$, $O_z = \mathit{outputs}(z, P)$, $I_c = \mathit{inputs}(c, P)$, $I_z = \mathit{inputs}(z, P)$, and $s \downarrow A$ returns the number of occurrences of elements of the set of events A in the trace s.
Formally, a buffering self-injection compatible process can establish a communication between its channels via a one-place buffer without deadlock.

Definition 3.14 (Reflexive composition). Let Ctr be a component contract, and ic and oc two channels, such that $\{ic, oc\} \subseteq \mathcal{C}_{Ctr}$, and $\mathcal{B}_{Ctr} \upharpoonright \{ic, oc\}$ is buffering self-injection compatible and satisfies the finite output property. The reflexive composition is defined as $Ctr[ic \hookrightarrow oc] = Ctr \asymp\!\Big|_{\langle oc \rangle}^{\langle ic \rangle}$.

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

The composition in Figure 7 is still deadlock-free, but the system development is unfinished. All philosophers are assembled to two other forks, except $PHIL3$, which is assembled to just $FORK3$. The next step would be to assemble the philosopher to his right fork ($FORK1$). As previously explained, we are not able to use the feedback composition because the channels `fk.1.3` and `pfk.1.3` are not decoupled. An alternative is to apply the reflexive composition presented above. However, its application is forbidden because its dependence protocols are not buffering self-injection compatible. In fact, by making this connection we introduce deadlock scenarios. In this way, our strategy supports an incremental design, providing feedback on attempts to reach configurations that might lead to deadlock.

A well-known solution is to break the existing symmetry by changing the preferences of a philosopher on the table. For instance, we can replace $PHIL3$ by another philosopher $PHIL3'$ presented below, which always picks up and puts down the right fork first.

$$PHIL3' = Comp_{INST}(Ctr_{PHIL}, \{\texttt{pf1} \mapsto \texttt{pfk.1.3}, \texttt{fk2} \mapsto \texttt{pfk.3.3}, \texttt{lf} \mapsto \texttt{life.3}\})$$

The final system can be obtained by a reflexive composition, assembling the remaining channels of $FORK1$ and $PHIL3'$, which are also not decoupled (and so feedback composition could not be applied), as presented below, reaching the final configuration presented in Figure 8.

$$TABLE = LEFT\_PHFKS[fk.1.3 \hookrightarrow pfk.3.3]$$

The following theorem guarantees that deadlock-freedom is preserved by reflexive composition.

Theorem 3.4 (Deadlock-free Reflexive Composition). The *reflexive composition* of two deadlock-free component contracts is also a deadlock-free component contract.

Proof. This proof is similar to the one for Theorem 3.3. Buffers are also implicitly introduced in the communication. To prove a divergent-free component Ctr is deadlock-free, we have to prove that $\mathcal{B}_{Ctr} \setminus \Sigma$

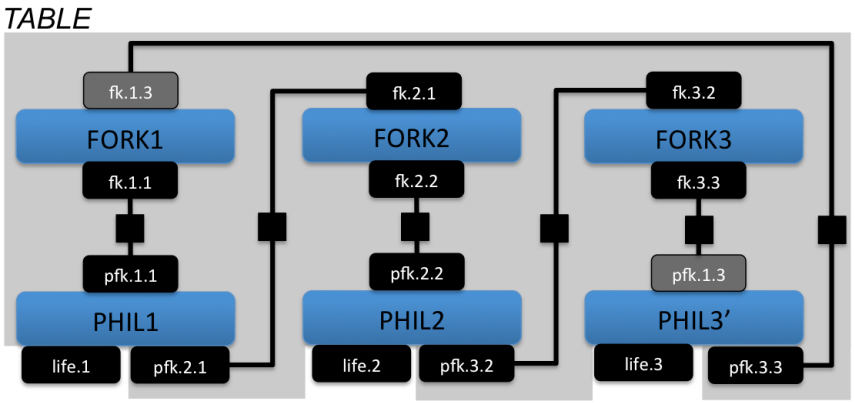


Fig. 8. Dining Philosophers Final Configuration

diverges [Ros98]. In the statements below we rewrite the behaviour of a composition of a component contract *Ctr* using the channels *ic* and *oc*.

$$
\begin{aligned}
& P[ic \hookrightarrow oc] \setminus \Sigma \\
&= \left(\mathcal{B}_{Ctr} \,[\![\, \{\!|\, ic, oc \,|\!\}\,]\!]\, \left(|||_{c \in \{ic\}}\, BUFF^{\infty}_{IO}(c, F(c))\right)\right) \setminus \Sigma \\
&= (\mathcal{B}_{Ctr} \,[\![\, \{\!|\, ic, oc \,|\!\}\,]\!]\, BUFF^{\infty}_{IO}(ic, oc)) \setminus \Sigma \\
&= (\mathcal{B}_{Ctr} \,[\![\, \{\!|\, ic, oc \,|\!\}\,]\!]\, BUFF^{\infty}_{IO}(ic, oc)) \setminus (\Sigma \setminus \{\!|\, ic, oc \,|\!\}) \setminus \Sigma \\
&= (\mathcal{B}_{Ctr} \upharpoonright \{\!|\, ic, oc \,|\!\} \,[\![\, \{\!|\, ic, oc \,|\!\}\,]\!]\, BUFF^{\infty}_{IO}(ic, oc)) \setminus \Sigma \\
&= (P \upharpoonright \{\!|\, ic, oc \,|\!\} \,[\![\, \{\!|\, ic, oc \,|\!\}\,]\!]\, BUFF^{\infty}_{IO}(ic, oc)) \setminus \Sigma
\end{aligned}
$$

According to Theorem B.5 the process in parentheses is deadlock-free, so hiding the whole alphabet results in divergence.

$$
\begin{aligned}
& P[ic \hookrightarrow oc] \setminus \Sigma \\
&= \mathsf{div}
\end{aligned}
$$

The proof that the resulting composition is a component contract follows directly from Theorem B.6. □

A similar theorem captures *feedback composition*.

Theorem 3.5 (Deadlock-free Feedback Composition). The *feedback composition* of a deadlock-free component contract is also deadlock-free.

Proof. This theorem follows directly from Theorem B.7. The decoupled property in the *feedback composition* implies in *self-injection compatibility*. As a result, according to Theorem 3.4, the feedback composition is also deadlock-free. □

The composition rules can also be applied in different orders to obtain the same system. For example, philosophers (and forks) have been interleaved, and then assembled by communication, feedback and reflexive compositions. We could, however, achieve the same final configuration of Figure 8 using the communication rule to connect forks to philosophers, and then assembling the resulting contracts with the use of communication, feedback and reflexive compositions.

All side conditions have been formalised using a mathematical notation. Their mechanical verification can be achieved by defining refinement assertions, presented in Section 5, which can be checked using FDR. Using these assertions, we were able to apply (and automatically verify) the systematic development approach to the case studies, as detailed in Sections 6 and 8.

4. Enriched Components with Metadata

A major contribution of this paper is the enriched component contract ($\mathcal{BRICK}$) that enriches the original contract ($\mathcal{BRIC}$) with metadata, which progressively records information that can be used to alleviate some verification conditions during component composition. Such metadata enriches contracts with static information that assist the environment with additional properties. The main metadata information selected in our approach are decoupled channels and protocol implementations. In addition, we add metadata associated to dual protocols and context processes in order to help protocol compatibility verification. We enrich our previous notion of component contract with a new metadata element.

Definition 4.1 (Enriched component contract). Let Ctr be a component contract, and $\mathcal{K}$ a metadata derived from its elements. An enriched component contract that includes Ctr is $\langle \mathcal{B}_{Ctr}, \mathcal{R}_{Ctr}, \mathcal{I}_{Ctr}, \mathcal{C}_{Ctr}, \mathcal{K} \rangle$ where $\mathcal{B}_{Ctr}$, $\mathcal{R}_{Ctr}$, $\mathcal{I}_{Ctr}$, and $\mathcal{C}_{Ctr}$ are the $\mathcal{BRIC}$ elements (as presented in the previous section), and $\mathcal{K}$ is defined as $\langle Prot^{\mathcal{K}}, CTX^{\mathcal{K}}, DProt^{\mathcal{K}}, Dec^{\mathcal{K}} \rangle$, such that:

- $\text{dom } Prot^{\mathcal{K}} \subseteq \mathcal{C}_{Ctr} \ \land \forall\, c : \text{dom } Prot^{\mathcal{K}} \bullet Prot^{\mathcal{K}}(c) \sqsubseteq_{\text{F}} Prot_{IMP}(Ctr, c)$
- $\text{dom } DProt^{\mathcal{K}} \subseteq \mathcal{C}_{Ctr} \ \land \forall\, c : \text{dom } DProt^{\mathcal{K}} \bullet DProt^{\mathcal{K}}(c)$ is the dual protocol of $Prot^{\mathcal{K}}(c)$
- $\text{dom } CTX^{\mathcal{K}} \subseteq \mathcal{C}_{Ctr} \ \land \forall\, c : \text{dom } CTX^{\mathcal{K}} \bullet CTX^{\mathcal{K}}(c)$ is the context process of $Prot^{\mathcal{K}}(c)$
- $\text{dom } Dec^{\mathcal{K}} \subseteq \mathcal{C}_{Ctr} \ \land\ \text{ran } Dec^{\mathcal{K}} \subseteq \mathcal{C}_{Ctr}$
- $\forall\, c_1, c_2 : \mathcal{C}_{Ctr} \bullet c_1\ Dec^{\mathcal{K}}\ c_2 \Rightarrow \{c_1, c_2\}\ DecoupledIn\ Ctr\ \land\ c_2\ Dec^{\mathcal{K}}\ c_1$

The element $Prot^{\mathcal{K}}$ is a mapping from channels to protocols, which represents the protocols of the component on its channels. If a protocol within $Prot^{\mathcal{K}}$ satisfies a property, then, by refinement, this property also holds for the protocol of the component. Similarly, the elements $DProt^{\mathcal{K}}$ and $CTX^{\mathcal{K}}$ map channels into context processes and dual protocols, respectively. Finally, the element $Dec^{\mathcal{K}}$ is a relation among decoupled channels of the component.

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

Definition 4.2 (Communication context process). Let P be a deadlock-free communication protocol. The communication context process of P (denoted by $CTX(P)$) is defined as a deadlock-free deterministic process, such that $traces(CTX(P)) = traces(P)$.

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

In our example, we alleviate the verification cost of the compositions by using the metadata of forks and philosophers, which are very simple and only consider information about their protocols; there are no decoupled channels in these components. All verifications about the protocols of the fork can be performed on the same protocol `PROT_FK`, which is the same for all of them. Similarly, the behaviour (protocols) that philosophers use to communicate with the forks can be represented by `PROT_PH`. These protocols are deterministic, and furthermore they have behavioural equivalent context processes and dual protocols. The enriched versions of these components use these protocols.

$$FORK1^e = Enrich(FORK1, MetaProt(\texttt{PROT\_FK(fk1)}, \texttt{fk1}, \texttt{fk.1.1}, \texttt{fk.1.3}))$$
$$FORK2^e = Enrich(FORK2, MetaProt(\texttt{PROT\_FK(fk1)}, \texttt{fk1}, \texttt{fk.2.2}, \texttt{fk.2.1}))$$
$$FORK2^e = Enrich(FORK3, MetaProt(\texttt{PROT\_FK(fk1)}, \texttt{fk1}, \texttt{fk.3.3}, \texttt{fk.3.2}))$$

$$PHIL1^e = Enrich(PHIL1, MetaProt(\texttt{PROT\_PH(pf1)}, \texttt{pf1}, \texttt{pfk.1.1}, \texttt{pfk.2.1}))$$
$$PHIL2^e = Enrich(PHIL2, MetaProt(\texttt{PROT\_PH(pf1)}, \texttt{pf1}, \texttt{pfk.2.2}, \texttt{pfk.3.2}))$$
$$PHIL3^e = Enrich(PHIL3, MetaProt(\texttt{PROT\_PH(pf1)}, \texttt{pf1}, \texttt{pfk.3.3}, \texttt{pfk.1.3}))$$

The function *Enrich* is used to construct enriched component contracts from regular component contracts and their already known metadata.

$$Enrich(Ctr, \mathcal{K}) = \langle \mathcal{B}_{Ctr}, \mathcal{R}_{Ctr}, \mathcal{I}_{Ctr}, \mathcal{C}_{Ctr}, \mathcal{K} \rangle$$

Furthermore, the function $MetaProt(\mathtt{P},\mathtt{c},\mathtt{a},\mathtt{b})$ is a function that builds a metadata with protocols similar to `P` substituting the channel `c` by the channels `a` and `b`, and with an empty set of decoupled channels.

$$MetaProt(\mathtt{P},\mathtt{c},\mathtt{a},\mathtt{b}) = \langle MProts(\mathtt{P},\mathtt{c},\mathtt{a},\mathtt{b}), MProts(\mathtt{P},\mathtt{c},\mathtt{a},\mathtt{b}), MProts(\mathtt{P},\mathtt{c},\mathtt{a},\mathtt{b}), \{\}\rangle$$
$$MProts(\mathtt{P},\mathtt{c},\mathtt{a},\mathtt{b}) = \{\mathtt{a} \mapsto \texttt{P [[c <- a]]}, \mathtt{b} \mapsto \texttt{P [[c <- b]]}\}$$

By way of illustration, the resulting enriched contract of $FORK1$, $FORK1^e$, is defined as:

$$FORK1^e = \left\langle \begin{array}{l} \texttt{FORK [[fk1 <- fk.1.1, fk2 <- fk.1.3]]}, \\ \{\texttt{fk.1.1} \mapsto \texttt{I\_FORK}, \texttt{fk.1.3} \mapsto \texttt{I\_FORK}\}, \\ \{\texttt{I\_FORK}\}, \\ \{\texttt{fk.1.1}, \texttt{fk.1.3}\}, \\ \langle Meta\_FORK1^e, Meta\_FORK1^e, Meta\_FORK1^e, \{\}\rangle \end{array} \right\rangle$$

where

$$Meta\_FORK1^e = \left\{ \begin{array}{l} \texttt{fk.1.1} \mapsto \texttt{PROT\_FK(fk1)[[fk1 <- fk.1.1]]}, \\ \texttt{fk.1.3} \mapsto \texttt{PROT\_FK(fk1)[[fk1 <- fk.1.3]]} \end{array} \right\}$$

To potentially increase the practical applicability of our compositional approach, we derive composition metadata from the metadata of the original components, without needing to build them from scratch. After each composition rule is applied, the metadata of the resulting component is obtained by simple calculations that consider the effect of the rule.

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 (except for interleaving) require a notion of protocol compatibility, which we call matching compatibility.

(xii) [**Matching Compatibility**] Two protocols P and Q are compatible if, and only if, the dual protocol of P is failure equivalent to Q ($DProt(P) \equiv_{\mathrm{F}} Q$).

This kind of compatibility is stronger than strong compatibility. The advantage of compositions in which the protocols are matching compatible is that they preserve local progress and, furthermore, other protocols (not involved in the composition) are preserved.

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

Definition 4.3 (Enriched interleaving composition). Let Ctr_1^e and Ctr_2^e be two enriched component contracts, such that $\mathcal{C}_{Ctr_1^e} \cap \mathcal{C}_{Ctr_2^e} = \emptyset$. Then, the enriched interleaving composition of Ctr_1^e and Ctr_2^e is given by:

$$Ctr_1^e \,[\!|||\!]^e\, Ctr_2^e = Enrich(\langle \mathcal{B}_{Ctr_1^e}, \mathcal{R}_{Ctr_1^e}, \mathcal{I}_{Ctr_1^e}, \mathcal{C}_{Ctr_1^e}\rangle \,[\!|||\!]\, \langle \mathcal{B}_{Ctr_2^e}, \mathcal{R}_{Ctr_2^e}, \mathcal{I}_{Ctr_2^e}, \mathcal{C}_{Ctr_2^e}\rangle, \langle Prot_3^{\mathcal{K}}, CTX_3^{\mathcal{K}}, DProt_3^{\mathcal{K}}, Dec_3^{\mathcal{K}}\rangle)$$

where

$$Prot_3^{\mathcal{K}} = Prot_{Ctr_1^e}^{\mathcal{K}} \cup Prot_{Ctr_2^e}^{\mathcal{K}}$$
$$CTX_3^{\mathcal{K}} = CTX_{Ctr_1^e}^{\mathcal{K}} \cup CTX_{Ctr_2^e}$$
$$DProt_3^{\mathcal{K}} = DProt_{Ctr_1^e}^{\mathcal{K}} \cup DProt_{Ctr_2^e}^{\mathcal{K}}$$
$$Dec_3^{\mathcal{K}} = Dec_{Ctr_1^e}^{\mathcal{K}} \cup Dec_{Ctr_2^e}^{\mathcal{K}} \cup \{(c_1, c_2) \mid (c_1 \in \mathcal{C}_{Ctr_2^e} \wedge c_2 \in \mathcal{C}_{Ctr_1^e}) \vee (c_1 \in \mathcal{C}_{Ctr_1^e} \wedge c_2 \in \mathcal{C}_{Ctr_2^e})\}$$

The result of this composition is similar to the one in Definition 3.10. In addition, we show here the metadata associated to the interleaving. At this stage, 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 Ctr_1 and Ctr_2, except that it also records the fact that all channels of one component are decoupled from the other; this is a direct result of the interleaved behaviour of the composition.

Theorem 4.1 (Enriched Interleaving Composition Compatibility). An enriched interleaving composition is an enriched component contract.

Proof. The proof of this theorem is presented in Appendix C. □

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, presented in Section 3.

The *TABLE* configuration presented in Figure 8 can be achieved using the corresponding enriched versions of the composition rules used in Section 3. For example, philosophers and forks can be interleaved separately as presented below, until we reach the configuration presented in Figure 5.

$$FORKS\_1\_2^e = FORK1^e \,[\![\!|\!|]\!]^e\, FORK2^e$$
$$FORKS^e = FORKS\_1\_2^e \,[\![\!|\!|]\!]^e\, FORK3^e$$

$$PHILS\_1\_2^e = PHIL1^e \,[\![\!|\!|]\!]\, PHIL2^e$$
$$PHILS^e = PHILS\_1\_2^e \,[\![\!|\!|]\!]\, PHIL3^e$$

As a simple example that illustrates the calculation of metadata, we present below the resulting enriched contract $FORKS\_1\_2^e$. The behaviour of the resulting contract is the interleaving of the behaviours of the basic enriched contracts. The remaining contract components are the union of the corresponding components of the basic contracts.

$$FORKS\_1\_2^e = \left\langle \begin{array}{l} \texttt{FORK[[fk1 <- fk.1.1, fk2 <- fk.1.3]] ||| FORK[[fk1 <- fk.2.2, fk2 <- fk.2.1]]}, \\ \{\texttt{fk.1.1} \mapsto \texttt{I\_FORK}, \texttt{fk.1.3} \mapsto \texttt{I\_FORK}, \texttt{fk.2.2} \mapsto \texttt{I\_FORK}, \texttt{fk.2.1} \mapsto \texttt{I\_FORK}\}, \\ \{\texttt{I\_FORK}\}, \\ \{\texttt{fk.1.1}, \texttt{fk.1.3}, \texttt{fk.2.2}, \texttt{fk.2.1}\}, \\ \langle Meta\_FORKS\_1\_2^e, Meta\_FORKS\_1\_2^e, Meta\_FORKS\_1\_2^e, \{\}\rangle \end{array} \right\rangle$$

where

$$Meta\_FORKS\_1\_2^e = \left\{ \begin{array}{l} \texttt{fk.1.1} \mapsto \texttt{PROT\_FK(fk1)[[fk1 <- fk.1.1]]}, \\ \texttt{fk.1.3} \mapsto \texttt{PROT\_FK(fk1)[[fk1 <- fk.1.3]]}, \\ \texttt{fk.2.2} \mapsto \texttt{PROT\_FK(fk1)[[fk1 <- fk.2.2]]}, \\ \texttt{fk.2.1} \mapsto \texttt{PROT\_FK(fk1)[[fk1 <- fk.2.1]]} \end{array} \right\}$$

Similar calculations yield the resulting enriched contracts $FORKS^e$ and $PHILS^e$, which are composed using communication composition of enriched contracts presented in the sequel.

Definition 4.4 (Enriched communication composition). Let Ctr_1^e and Ctr_2^e be two enriched component contracts, and ic and oc two channels, such that $ic \in \mathcal{C}_{Ctr_1^e} \wedge oc \in \mathcal{C}_{Ctr_2^e}$, $\mathcal{C}_{Ctr_1^e} \cap \mathcal{C}_{Ctr_2^e} = \emptyset$, and $Prot^{\mathcal{K}}_{Ctr_1^e}(ic) \,[\![R_{IO}^{ic \to oc}]\!]$ and $Prot^{\mathcal{K}}_{Ctr_2^e}(oc) \,[\![R_{IO}^{oc \to ic}]\!]$ are I/O confluent matching compatible protocols and satisfy the finite output property. The enriched communication composition of Ctr_1^e and Ctr_2^e via ic and oc is defined as:

$$Ctr_1^e[ic \leftrightarrow oc]^e\, Ctr_2^e = Enrich \left(\begin{array}{l} \langle \mathcal{B}_{Ctr_1^e}, \mathcal{R}_{Ctr_1^e}, \mathcal{I}_{Ctr_1^e}, \mathcal{C}_{Ctr_1^e} \rangle [ic \leftrightarrow oc] \langle \mathcal{B}_{Ctr_2^e}, \mathcal{R}_{Ctr_2^e}, \mathcal{I}_{Ctr_2^e}, \mathcal{C}_{Ctr_2^e} \rangle, \\ \langle Prot_3^{\mathcal{K}}, CTX_3^{\mathcal{K}}, DProt_3^{\mathcal{K}}, Dec_3^{\mathcal{K}} \rangle \end{array} \right)$$

where

$$Prot_3^{\mathcal{K}} = \{(c, Prot^{\mathcal{K}}_{Ctr_1^e}(c)) \mid c \in \mathrm{dom}\, Prot^{\mathcal{K}}_{Ctr_1^e} \setminus \{ic\}\} \cup \{(c, Prot^{\mathcal{K}}_{Ctr_2^e}(c)) \mid c \in \mathrm{dom}\, Prot^{\mathcal{K}}_{Ctr_2^e} \setminus \{oc\}\}$$
$$CTX_3^{\mathcal{K}} = \{(c, CTX^{\mathcal{K}}_{Ctr_1^e}(c)) \mid c \in \mathrm{dom}\, CTX^{\mathcal{K}}_{Ctr_1^e} \setminus \{ic\}\} \cup \{(c, CTX^{\mathcal{K}}_{Ctr_2^e}(c) \mid c \in \mathrm{dom}\, CTX^{\mathcal{K}}_{Ctr_2^e} \setminus \{oc\}\}$$
$$DProt_3^{\mathcal{K}} = \{(c, DProt^{\mathcal{K}}_{Ctr_1^e}(c)) \mid c \in \mathrm{dom}\, DProt^{\mathcal{K}}_{Ctr_1^e} \setminus \{ic\}\}$$
$$\cup \{(c, DProt^{\mathcal{K}}_{Ctr_2^e}(c) \mid c \in \mathrm{dom}\, DProt^{\mathcal{K}}_{Ctr_2^e} \setminus \{oc\}\}$$
$$Dec_3^{\mathcal{K}} = \left\{ \begin{array}{l} (c_1, c_2) \mid \{c_1, c_2\} \cap \{ic, oc\} = \emptyset \\ \qquad \wedge \left(\begin{array}{l} ((c_1\ Dec^{\mathcal{K}}_{Ctr_1^e}\ ic \vee ic\ Dec^{\mathcal{K}}_{Ctr_1^e}\ c_1) \wedge (c_2 \in \mathcal{C}_{Ctr_2^e} \vee c_1 Dec^{\mathcal{K}}_{Ctr_1^e} c_2)) \\ \vee ((oc\, Dec^{\mathcal{K}}_{Ctr_2^e}\, c_2 \vee c_2\, Dec^{\mathcal{K}}_{Ctr_2^e}\, oc) \wedge (c_1 \in \mathcal{C}_{Ctr_1^e} \vee c_1 Dec^{\mathcal{K}}_{Ctr_2^e} c_2)) \end{array} \right) \end{array} \right\}$$

The result of this composition is similar to that of Definition 3.12. However, instead of checking compatibility among port protocols of the original components, we check it on port protocols within their metadata. The composition also does not have to take into account the complexity of its components, since no 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. The calculation of metadata is very simple and

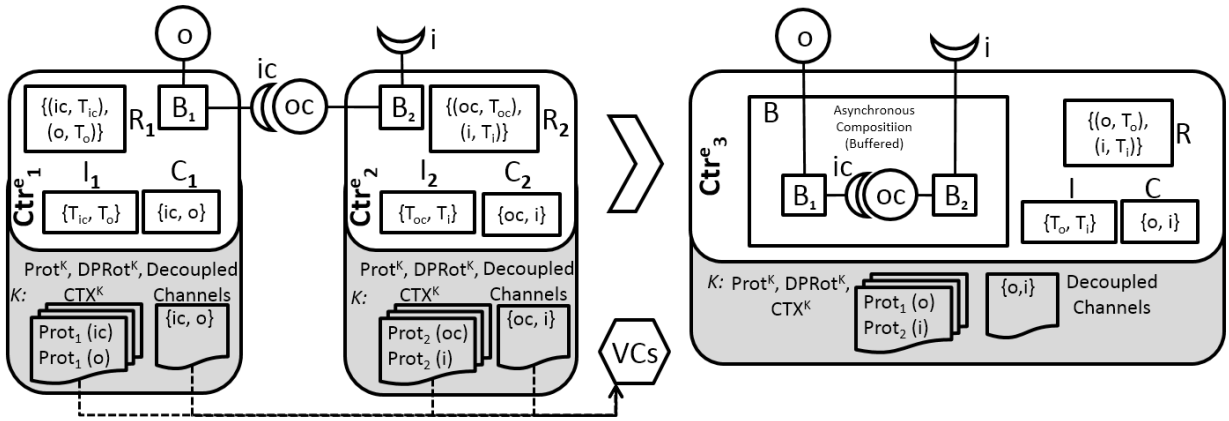


Fig. 9. Enriched Communication Composition Application

includes all information of the metadata of Ctr_1^e and Ctr_2^e, 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 $Dec^{\mathcal{K}}$ is a symmetric relation and this has to be handled in its calculation.

In Figure 9 we illustrate the application of Definition 4.4 to two enriched component contracts Ctr_1^e and Ctr_2^e with interaction points $\{ic, o\}$ and $\{oc, i\}$, respectively, which results in the component Ctr_3^e. The metadata (marked in gray) is used both to generate verification conditions (VCs) and to calculate the metadata of the resulting components. As we demonstrate in Section 6, this considerably alleviates the verification cost.

As for the interleaving composition, the communication composition of enriched components is also an enriched component.

Theorem 4.2 (Enriched Communication Composition Compatibility). An enriched communication composition is an enriched component contract.

Proof. The proof of this theorem is presented in Appendix C. □

The enriched contracts $FORKS^e$ and $PHILS^e$ may be composed using the enriched communication composition yielding the configuration presented in Figure 6.

$$LEFT\_PHFKS\_1\_1^e = FORKS^e[fk.1.1 \leftrightarrow pfk.1.1]^e PHILS^e$$

The resulting enriched contract $LEFT\_PHFKS\_1\_1^e$ has the same $\mathcal{BRIC}$ components as the original contract $LEFT\_PHFKS\_1\_1$. Its metadata includes all information of the metadata of $FORKS^e$ and $PHILS^e$, excluding information about `fk.1.1` and `pfk.1.1`, which do not belong to the resulting enriched contract. As we do not have decoupled channels, the resulting decoupled channels set remains empty.

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

Definition 4.5 (Enriched feedback composition). Let Ctr^e be an enriched component contract, and ic and oc two channels such that $\{ic, oc\} \subseteq \mathcal{C}_{Ctr^e}$, the port protocol $Prot^{\mathcal{K}}_{Ctr^e}(ic) \,[\![\, R^{ic \to oc}_{IO}]\!]$ and the port protocol $Prot^{\mathcal{K}}_{Ctr^e}(oc) [\![R^{oc \to ic}_{IO}]\!]$ are I/O confluent matching compatible and satisfy the finite output property, and $\{ic, oc\}$ are decoupled in Ctr^e. The feedback composition is defined as:

$$Ctr^e[oc \hookrightarrow ic]^e = Enrich(\langle \mathcal{B}_{Ctr^e}, \mathcal{R}_{Ctr^e}, \mathcal{I}_{Ctr^e}, \mathcal{C}_{Ctr^e} \rangle [oc \hookrightarrow ic], \langle Prot^{\mathcal{K}}_S, CTX^{\mathcal{K}}_S, DProt^{\mathcal{K}}_S, Dec^{\mathcal{K}}_S \rangle)$$

where

$$Prot^{\mathcal{K}}_3 = \{(c, Prot^{\mathcal{K}}_{Ctr^e}(c)) \mid c \in \operatorname{dom} Prot^{\mathcal{K}}_{Ctr^e} \setminus \{ic, oc\}\}$$

$$DProt^{\mathcal{K}}_3 = \{(c, DProt^{\mathcal{K}}_{Ctr^e}(c)) \mid c \in \operatorname{dom} DProt^{\mathcal{K}}_{Ctr^e} \setminus \{ic, oc\}\}$$

$$CTX^{\mathcal{K}}_3 = \{(c, CTX^{\mathcal{K}}_{Ctr^e}(c)) \mid c \in \operatorname{dom} CTX^{\mathcal{K}}_{Ctr^e} \setminus \{ic, oc\}\}$$

$$Dec^{\mathcal{K}}_3 = \left\{ \begin{array}{l} (c_1, c_2) \mid \{c_1, c_2\} \cap \{ic, oc\} = \emptyset \wedge c_1 \; Dec^{\mathcal{K}}_{Ctr^e} \; c_2 \\ \qquad \wedge \; \big(\; (c_1 \; Dec^{\mathcal{K}}_{Ctr^e} \; ic \wedge c_1 \; Dec^{\mathcal{K}}_{Ctr^e} \; oc) \vee (ic \; Dec^{\mathcal{K}}_{Ctr^e} \; c_2 \wedge oc \; Dec^{\mathcal{K}}_{Ctr^e} \; c_2) \; \big) \end{array} \right\}$$

The result of this composition is similar to the one from Definition 3.13, except that most provisos use the metadata of its original components directly. Instead of having to check compatibility among port protocols of P, 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 includes 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 $Dec_S^{\mathcal{K}}$, since after the composition new communications are established.

Theorem 4.3 (Enriched Feedback Composition Compatibility). An enriched feedback composition is an enriched component contract.

Proof. The proof of this theorem is presented in Appendix C. □

In our example, the connection of channels `pfk.2.2` and `fk.2.2` in $LEFT\_PHFKS\_1\_1^e$ may be achieved using the enriched feedback composition. The resulting enriched contract $LEFT\_PHFKS^e$ has the same $\mathcal{BRIC}$ components as the original contract $LEFT\_PHFKS$. Its metadata includes all information of the metadata of $LEFT\_PHFKS\_1\_1^e$ excluding information about `fk.2.2` and `pfk.2.2`, which do not belong to the resulting enriched contract.

$$LEFT\_PHFKS^e = LEFT\_PHFKS\_1\_1^e[fk.2.2 \hookrightarrow pfk.2.2]^e$$

Similar to the development presented in Section 3, the feedback composition may also be used to connect `fk.3.3` to `pfk.3.3`, `fk.2.1` to `pfk.2.1`, and `fk.3.2` to `pfk.3.2` until we reach the configuration in Figure 7.

The last rule is the reflexive composition of enriched compositions.

Definition 4.6 (Enriched reflexive composition). Let Ctr^e be a component contract, and ic and oc two communication channels, such that $\{ic, oc\} \subseteq \mathcal{C}_{Ctr^e}$, and $Ctr^e \upharpoonright \{ic, oc\}$ is buffering self-injection compatible and satisfies the finite output property. The reflexive composition is defined as:

$$Ctr^e[ic \hookleftarrow oc]^e = Enrich(\langle \mathcal{B}_{Ctr^e}, \mathcal{R}_{Ctr^e}, \mathcal{I}_{Ctr^e}, \mathcal{C}_{Ctr^e} \rangle [ic \hookleftarrow oc], \langle Prot_S^{\mathcal{K}}, CTX_S^{\mathcal{K}}, DProt_S^{\mathcal{K}}, Dec_S^{\mathcal{K}} \rangle)$$

where

$$Prot_3^{\mathcal{K}} = \{(c, Prot_{Ctr^e}^{\mathcal{K}}(c)) \mid c \in \operatorname{dom} Prot_{Ctr^e}^{\mathcal{K}} \setminus \{ic, oc\}\}$$

$$DProt_3^{\mathcal{K}} = \{(c, DProt_{Ctr^e}^{\mathcal{K}}(c)) \mid c \in \operatorname{dom} DProt_{Ctr^e}^{\mathcal{K}} \setminus \{ic, oc\}\}$$

$$CTX_3^{\mathcal{K}} = \{(c, CTX_{Ctr^e}^{\mathcal{K}}(c)) \mid c \in \operatorname{dom} CTX_{Ctr^e}^{\mathcal{K}} \setminus \{ic, oc\}\}$$

$$Dec_3^{\mathcal{K}} = \left\{ \begin{array}{l} (c_1, c_2) \mid \{c_1, c_2\} \cap \{ic, oc\} = \emptyset \wedge c_1 \ Dec_{Ctr^e}^{\mathcal{K}} \ c_2 \\ \qquad \wedge \ (\ (c_1 \ Dec_{Ctr^e}^{\mathcal{K}} \ ic \wedge c_1 \ Dec_{Ctr^e}^{\mathcal{K}} \ oc) \vee (ic \ Dec_{Ctr^e}^{\mathcal{K}} \ c_2 \wedge oc \ Dec_{Ctr^e}^{\mathcal{K}} \ c_2)\ ) \end{array} \right\}$$

The result of this composition is similar to the one from Definition 3.14. 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. As discussed in Section 7 we propose an alternative for the application of reflexive composition, based on communication patterns that supports an entirely local analysis. 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.

Theorem 4.4. An enriched reflexive composition is an enriched component contract.

Proof. The proof of this theorem is presented in Appendix C. □

The discussion on the final composition of our example using the reflexive composition presented in Section 3 applies at this point using the enriched reflexive composition. The final enriched system can be obtained by an enriched reflexive composition, assembling the remaining channels of $FORK1^e$ and $PHIL3'^e$, which are also not decoupled.

$$TABLE^e = LEFT\_PHFKS^e[fk.1.3 \hookleftarrow pfk.3.3]^e$$

Finally, we present the theorem that guarantees that all enriched rules presented in this section also guarantee deadlock-freedom.

Table 1. Mechanisation of Side Conditions in CSP for Interleave Composition

Alphabets	`assert STOP [T= RUN(inter(events(P),events(Q)))`
I/O Processes (i): I/O Channels	`assert not Test(inter(inputs(P),outputs(P)) == {}) [T= ERROR`
I/O Processes (ii): Infinite Traces	`assert not HideAll(P):[divergence free [FD]]`
I/O Processes (iii): Divergence Free	`assert P:[divergence free [FD]]`
I/O Processes (iv): Input Determinism	`assert LHS_InputDet(P) [F= RHS_InputDet(P)`

I/O Processes (v): Strong Output Decisive	`assert LHS_OutputDec_A(P) [F= RHS_OutputDec_A(P)`
	`assert LHS_OutputDec_B(P,c1) [F= RHS_OutputDec_B(P,c1)`
	`assert LHS_OutputDec_B(P,c2) [F= RHS_OutputDec_B(P,c2)`

Theorem 4.5 (Enriched Rules and Deadlock-Freedom). The enriched composition of deadlock-free component contracts is also a deadlock-free component contract.

Proof. This proof follows directly from Theorems 4.1 to 4.4 and from Theorems 3.2 to 3.5. □

Using the enriched approach, we obtain the same final configuration as that presented in Figure 8. Nevertheless, using enriched components, we reduce the verification cost. This is demonstrated in the experiments presented in Section 6.

5. Mechanising the Composition Rule Side Conditions in CSP

In Sections 3 and 4, we present a formalisation of all side conditions using a mathematical notation. Here, we define CSP assertions for all these conditions, which allow an automatic verification using a tool like FDR as we demonstrate in our case study presented in Section 6.

By way of illustration, Table 1 presents some of the mechanisation of the side conditions in CSP for the interleave composition of two processes P ($\alpha P = \{c1, c2\}$) and Q described in the sections that follow. Counterparts of the assertions presented in these sections are also needed for process Q. A summary of the complete mechanisation of the composition rule side conditions in CSP is presented in [OSA+13].

5.1. Alphabets

The first assertion in Table 1 guarantees that the channels of the processes are disjoint by checking that offering (`RUN`) all events of the intersection (`inter`) between both process events is a refinement of `STOP`. Since `STOP` offers no events, this is only possible if the intersection is empty.

5.2. I/O Channels

The assertion related to I/O channels is similar but is characterised in a different manner because functions *inputs* and *outputs* return channels, not events, and hence cannot be used in `RUN`. Its characterisation test uses two auxiliary processes: `ERROR = error -> SKIP` and `Test(c) = not c & ERROR`. This assertion is only satisfied if the condition is true.

5.3. Infinite Traces and Divergence-Freedom

Infinite traces are checked by asserting that hiding all events (`HideAll`) introduces divergence. Both this check and the one that checks if the process itself is divergence-free are achieved using FDR's built-in divergence check.

The next two assertions proved to be much more difficult to encode as a refinement assertion, and deserve special attention.

5.4. Input Determinism

We formally define input determinism as follows:

Definition 5.1 (Input determinism). We say a process P is input deterministic if

$$\forall\, s \frown \langle c.a \rangle : traces(P) \mid \; c.a \in inputs(c, P) \bullet (s, \{c.a\}) \notin failures(P)$$

Informally, this means that if a set of input events in P are offered to the environment, none of them are refused. As a consequence, the process is defined to be deterministic on the inputs.

In [Ros10], Roscoe presents a refinement check for **divergence-free processes** in FDR that is based on Lazić's Algorithm [Laz99].

The approach is to run two copies of the process synchronising on a newly introduced special event `clunk`. Furthermore, the set `AllButClunk` includes all events that `P` uses, but not the special event `clunk`.

```
channel clunk
AllButClunk = diff(Events,{clunk})
```

This special event is used to synchronise both copies of the process after any event. First, we enforce that the process synchronises in this special event after any other events. This is achieved by running the process in parallel with a watchdog process that produces a `clunk` after any event, as follows.

```
Clunking(P) = P [| AllButClunk |] Clunker
Clunker = [] x:AllButClunk @ x -> clunk -> Clunker
```

`Clunking(P)` behaves exactly like `P`, except that it communicates `clunk` between each pair of other events.

Next, we run both controlled copies of the process in parallel, but synchronising only on `clunk`. It follows that `(Clunking(P) [|{clunk}|] Clunking(P))\{clunk}` allows both copies of `P` to proceed independently, except that their individual traces never differ in length by more than one.

If `P` is deterministic, then, whenever one copy of `P` performs an event, the other one cannot refuse it provided they have both performed the same trace to date. It follows that if we run

```
RHS_InputDet(P) = (Clunking(P)[|{clunk}|]Clunking(P)) \ {clunk}
                  [|AllButClunk|]
                  Repeat
Repeat = [] x:AllButClunk @ x -> x -> Repeat
```

then the result will never deadlock after a trace with odd length. Such a deadlock can only occur if, after some trace of the form `<a,a,...,d,d>` in which each `P` has performed `<a,...,d>`, one copy of `P` accepts some event `e` and the other refuses it. This exactly corresponds to `P` not being $\mathcal{F}$-deterministic.

We can thus check determinism by testing whether the process `RHS_InputDet(P)` refines the following process over $\mathcal{F}$.

```
Deterministic(S) = STOP
                   |~|
                   ([] x:AllButClunk @ x -> (if member(x,S)
                                             then x -> Deterministic(S)
                                             else (STOP |~| x -> Deterministic(S))))
LHS_InputDet(P) = Deterministic(inputs(P))

assert LHS_InputDet(P) [F= RHS_InputDet(P)
```

The process `LHS_InputDet(P)` specifies a deterministic behaviour of the set of input events of a given process (`inputs(P)`). Notice that using `AllButClunk` bring us back to the original Lazić's algorithm, in which `LHS_InputDet = STOP |~| ([] x:AllButClunk @ x -> x -> LHS_InputDet)` and checks determinism in all events. We are, however, interested in a particular set of events `S`, namely the inputs.

Because it runs `P` in parallel with itself, Lazić's algorithm is at worst quadratic in the state space of `P`. (In other words, the number of states can be as many as the square of the state space of `P`.) In most cases, however, it is much better than this, but not as efficient as the FDR check.

Lazić's algorithm works (in the respective models) to determine whether a process is deterministic over $\mathcal{FD}$ or $\mathcal{F}$.

The fact that this algorithm is implemented by the user in terms of refinement checking means that it is easy to vary, and in fact many variations on this check have been used when one wants to compare the different ways in which a process `P` can behave on the same or similar traces. We use this idea for Strong Output Decisiveness as we explain in the sequel.

5.5. Strong Output Decisiveness

Strong Output Decisiveness if formally defined in Page 10; it is repeated here to facilitate references to the formal definition.

$$\forall\, e : \Sigma;\ s : \operatorname{seq}\Sigma \mid s \frown \langle e \rangle \in traces(P) \land e \in outputs_P(P)$$
$$\bullet\ (\exists\, c : CHANNEL \mid e \in outputs(c, P)$$
$$\bullet\ (s, outputs(c, P)) \notin failures(P) \land (s, outputs(c, P) \setminus \{e\}) \in failures(P))$$

Informally, this means that 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. Hence, the choice between output channels is external.

In [Ros05], processes are output decisive on a channel `c` if every maximal refusal of the process omits at most one member of `{|c|}`. This definition, however, differs from ours in three main aspects:

- **Channel based definition**: In [Ros05], a channel-based approach is presented. Channels are considered to be unidirectional for each process; hence, within each process, a channel is either input or output. For this reason, in [Ros05], processes are not allowed to offer an external choice between an input and an output on the same channel as they are here.
- **Single event outputs**: In [Ros05], process `P = c -> P` (with `c` $\in outputs_P(P)$) is not strong output decisive. In our definition, though, it is.
- **Refusing all outputs**: In [Ros05], a strong output decisive process might refuse all outputs on a given channel at once. Here, we reject such processes as strong output decisive; if a process might offer an output on `c1`, it might not refuse all outputs on `c1` at once. So, both processes below are Strong Output Decisive according to [Ros05], but they are not according to our definition.

We may, therefore, state our notion of Strong Output Decisiveness as follows: a process `P` is Strong Output Decisive if choices between outputs on different channels are external and choices between outputs on the same channel are internal.

The characterisation of strong output decisiveness as assertions will be divided into two parts:

- The **Part A** verifies that after a trace `s^<c.x>` (see definition of Strong Output Deciseveness), the process cannot refuse all events on `{|c|}`. This verification, however, does not guarantee that choices are non-deterministic.
- The **Part B** verifies that in a trace `s^<c.x>`, the process might refuse all events on `{|c|} \ {c.x}`. Hence, the process is non-deterministic for outputs on that channel.

5.5.1. Part A - Inter-channel Determinism.

Let `GET_CHANNELS(P)` be a set of distinct channels used in process `P`. Using the same `Clunker(p)` as previously described in Section 5.4, we now use two copies of the clunking version of `P` synchronising on `clunk` and everything except members of the channels we are worrying about, the outputs of `P`.

```
(Clunking(P)[|diff(Events, outputs(P))|]Clunking(P)) \ {clunk}
```

Furthermore, we consider a process `One2Many(S)`, which simply repeats events that are not communication on the channels in `S`; otherwise, it offers any other communication on that channel.

```
One2Many(S) =
    ([] x:diff(Events,union(S,{clunk})) @ x -> One2Many(S))
    [] ([] c:S @ [] x:{|c|} @ x -> One2Many'(S,c,x))
One2Many'(S,c,x) = [] y:chan(c,P) @ y -> if x==y then One2Many(S) else STOP
```

Here, we use an auxiliary function `chan(ev,P)`, which returns the outputs events on the same channel of a given event.

```
chan(ev,P) = inter(outputs(P),{| c | c <- GET_CHANNELS(P), member(ev,{|c|})|})
```

We put the process `One2Many` in parallel with the above to get the right-hand side implementation of the assertion.

```
RHS_OutputDec_A(P) = (Clunking(P)[|diff(Events, outputs(P))|]Clunking(P))\{clunk}
                     [| AllButClunk |]
                     One2Many(outputs(P))
```

This process expects the second copy of `P` to respond with a member of the same channel when an output has occurred. Importantly, it only continues the test when both copies have performed the same event: so at all times both copies of `P` have performed the same trace.

We test this implementation against the specification below.

```
LHS_OutputDec_A(P) =
    STOP
    |~| ([] x:diff(Events,union(outputs(P),{clunk})) @ x -> LHS_OutputDec_A(P))
    [] ([] x:outputs(P) @ x -> (|~| y:chan(x,P) @ y -> LHS_OutputDec_A(P)))
```

where

```
chan(ev,P) = inter(outputs(P), {| c | c <- GET_CHANNELS(P), member(ev,{|c|})|})
```

This will allow any trace that `RHS_OutputDec_A` can make, and only insists on some member of the same channel occurring after an output.

It is important to note that this certainly tests all traces of `P` since whenever one copy of `P` performs an event after trace t, it is certain that the other one can perform it, even though it may also be capable of refusing that event.

Thus, the refinement check below checks that, after every trace t of `P` after which an output can happen, the process cannot refuse the whole of the corresponding channel.

```
LHS_OutputDec_A [F= RHS_OutputDec_A(P)
```

This verification, however, does not check that the process can refuse all but one member of that channel. Hence, it does not check that the process is non-deterministic for a given output channel. For this reason, this verification accepts processes that offer an external choice on the outputs of a same channel. Hence, a further check is needed to guarantee the non-deterministic choice on the outputs of the same channel.

5.5.2. Part B - Intra-channel Non-determinism.

At this part of the verification, we need to guarantee that every single output of `P` can, if blocked, deadlock it on the same trace. To do this we need a Lazić construction [Laz99] on the left-hand side of the refinement check, along the lines of the process below.

```
LHS_OutputDec_B(P,c) = (FirstCopy(P)[|{clunk}|]SecondCopy(P))\{clunk}
                       [|Events|]
                       LHS_Test(inter({|c|},outputs(P)))
```

where

```
FirstCopy(P) = P [| AllButClunk |] DoubleClunker
SecondCopy(P) = P [| AllButClunk |] clunk -> DoubleClunker
DoubleClunker = [] x:AllButClunk @ x -> clunk -> clunk -> DoubleClunker

LHS_Test(S) = [] x:S @ x -> (x -> LHS_Test(S) [> ([] y:diff(S,{x}) @ y -> STOP)
                                                 []
                                                 ([] y:diff(Events,S) @ y -> STOP))
                            [] ([] y:diff(Events,S) @ y -> y -> LHS_Test(S))
```

The process `LHS_OutputDec_B(P,c)` strictly alternates events of the two copies of the `P`, and as long as they have performed the same trace to date can, after a `c.x`, offer everything other than that event itself. So it ought, under what we want, to be able to refuse the whole of `{|c|}` after the first of each pair of `c.x` events.

We then check if `LHS_OutputDec_B` is refined by `RHS_OutputDec_B`, which is similar to `LHS_OutputDec_B` but uses `RHS_Test` rather than `LHS_Test`.

```
RHS_OutputDec_B(P,c) = (FirstCopy(P)[|{clunk}|]SecondCopy(P))\{clunk}
                       [|Events|]
                       RHS_Test(inter({|c|}, outputs(P)))

RHS_Test(S) = [] x:S @ x -> (([] y:S @ y -> if x==y then RHS_Test(S) else STOP)
                               [> ([] y:diff(Events,S) @ y -> STOP))
                            [] ([] y:diff(Events,S) @ y -> y -> RHS_Test(S))
```

Now, process `RHS_OutputDec_B(P,c)` behaves in the same way except that it can prevent the second process from performing the second of a pair of `c.x`'s. In both cases, the untimed time-out is used to create the possibility of repeating the event already input, without offering it in a stable way.

5.5.3. Combining Assertions.

The final verification of Strong Output Decisiveness is then achieved in two parts. First, we verify the Part A, which is done for the whole process at once.

```
assert LHS_OutputDec_A [F= RHS_OutputDec_A(P)
```

If the assertion fails, the process is not Strong Output Decisive and the verification finishes. If, however, the process passes Part A, we need to check the Part B, which is done individually for every channel within the processes alphabet. For instance, supposing process α`P` = `{c1,c2}` we need the following assertions.

```
assert LHS_OutputDec_B(P,c1) [F= RHS_OutputDec_B(P,c1)
assert LHS_OutputDec_B(P,c2) [F= RHS_OutputDec_B(P,c2)
```

5.6. Further Side Conditions in CSP

Similar tricks were used to encode similar side conditions like checking if a channel is in the alphabet of a process. The assertions for decoupled channels (Prop. x) *ic* and *oc* in P is encoded as a bi-directional refinement between the projection of P over both channels and the interleaving of the protocol implementation of P over each individual channel.

The finite output property (Prop. vii) has been characterised as an assertion that hiding all outputs of the protocol `P` does not introduce divergence:

```
assert P \ allOutputs:[divergence free [FD]]
```

Furthermore, the work presented in [Ros05, Ros06] presents a test characterisation of confluence, by checking if placing inwards buffers of size one from the environment to a process P, the resulting process is deterministic.

Theorem 5.1 (I/O Confluence Assertion). A process P is I/O confluent if, and only if, the process in which a one-place inwards-pointing buffer is placed on every individual event of $P \,[\![R]\!]$ (where R is a forgetful renaming that removes the data components of all channels but preserves their direction), is deterministic.

Proof. Details about the proof of this theorem can be obtained in [Ros05]. □

Based on this theorem, our characterisation for processes P, like our implementation protocols, that work in a single event `c` are defined as:

```
assert InBufferProt(P,c) :[deterministic [F]]
```

The theorem below states that protocols are strong compatible (Prop. ix) if one of them is a failures refinement of the dual protocol of the other.

Theorem 5.2. Let P and Q be deadlock-free communication protocols, and DP the dual protocol of P, such that $DP \sqsubseteq_{\mathrm{F}} Q$. Then P and Q are strong compatible.

Proof. This Theorem follows directly from Lemma 5.1 (introduced next), since if Q refines the dual protocol, then Q can substitute it in all possible scenarios, based on Lemma 5.1.

$$\begin{aligned}
&\forall\, s : \mathit{traces}(P) \cap \mathit{traces}(DP) \bullet (O^s_P \neq \emptyset \vee O^s_{DP} \neq \emptyset) \wedge O^s_P \subseteq I^s_{DP} \wedge O^s_{DP} \subseteq I^s_P \quad [\mathit{traces}(Q) \subseteq \mathit{traces}(DP)] \\
&\Rightarrow \forall\, s : \mathit{traces}(P) \cap \mathit{traces}(Q) \bullet (O^s_P \neq \emptyset \vee O^s_{DP} \neq \emptyset) \wedge O^s_P \subseteq I^s_{DP} \wedge O^s_{DP} \subseteq I^s_P \\
&\qquad\qquad [\mathit{failures}(Q) \subseteq \mathit{failures}(DP) \text{ and protocols are input deterministic, } I^s_{DP} = I^s_Q \cap O^s_P] \\
&\Rightarrow \forall\, s : \mathit{traces}(P) \cap \mathit{traces}(Q) \bullet (O^s_P \neq \emptyset \vee O^s_{DP} \neq \emptyset) \wedge O^s_P \subseteq I^s_Q \wedge O^s_{DP} \subseteq I^s_P \\
&\qquad\qquad [\mathit{failures}(Q) \subseteq \mathit{failures}(DP) \text{ and protocols are output decisive}] \\
&\qquad\qquad [O^s_Q \subseteq O^s_{DP},\ O^s_{DP} \neq \emptyset \Rightarrow O^s_Q \neq \emptyset] \\
&\Rightarrow \forall\, s : \mathit{traces}(P) \cap \mathit{traces}(Q) \bullet (O^s_P \neq \emptyset \vee O^s_Q \neq \emptyset) \wedge O^s_P \subseteq I^s_Q \wedge O^s_Q \subseteq I^s_P \\
&\qquad\qquad [\text{Lemma 5.1 and Definition of Strong Protocol Compatibility (Page 16)}] \\
&\Rightarrow P \text{ and } Q \text{ are strong compatible}
\end{aligned}$$

□

Its proof is based on the following lemma:

Lemma 5.1. Let P be a deadlock-free communication protocol, and DP its dual protocol. Then P and DP are strong compatible.

Proof. This proof follows directly from the Definition of Dual Protocol (Definition 3.11) and the definition on Strong Protocol Compatibility (Page 16). We start the proof by a true statement.

$$\begin{aligned}
&\forall\, t : \mathit{traces}(P) \bullet I^s_P \subseteq I^s_P \wedge O^s_P \subseteq O^s_P \qquad [P \text{ is deadlock-free}] \\
&\Rightarrow \forall\, t : \mathit{traces}(P) \bullet (O^s_P \neq \emptyset \vee I^s_P \neq \emptyset) \wedge I^s_P \subseteq I^s_P \wedge O^s_P \subseteq O^s_P \\
&\qquad\qquad [\text{Definition 3.11, } O^s_P = I^s_{DP},\ I^s_P = O^s_{DP}] \\
&\Rightarrow \forall\, t : \mathit{traces}(P) \cap \mathit{traces}(DP) \bullet (O^s_P \neq \emptyset \vee O^s_{DP} \neq \emptyset) \wedge O^s_{DP} \subseteq I^s_P \wedge O^s_P \subseteq I^s_{DP} \\
&\qquad\qquad [\text{Definition of Strong Protocol Compatibility (Page 16)}] \\
&\Rightarrow P \text{ and } DP \text{ are strong compatible}
\end{aligned}$$

□

Based on Theorem 5.2, we may characterise strong compatibility check as assertions on simple failures refinement.

Finally, a third theorem states that a buffering self-injection compatible (See Prop. xi) process can establish a communication between its channels via a one-place buffer without deadlock.

Theorem 5.3. Let P be an deadlock-free I/O process, c and z communication channels, and LR_1 and LR_2 bijections, such that:

(i) $LR_1 : \mathit{outputs}(P, c) \leftrightarrow \mathit{inputs}(P, z)$ and $LR_2 : \mathit{outputs}(P, z) \leftrightarrow \mathit{inputs}(P, c)$
(ii) $\mathit{Prot}_{IMP}(P, c) \,[\![\, LR_1 \,]\!]\, \approx \mathit{Prot}_{IMP}(Q, z) \,[\![\, LR_2]\!]$
(iii) $\mathit{Prot}_{IMP}(P, c)$ and $\mathit{Prot}_{IMP}(Q, z)$ satisfy the finite output property.

Then, $P \upharpoonright \{\!|\, c, z \,|\!\}$ is *buffering self-injection compatible* if, and only if, the following process is deadlock-free:

$$P \upharpoonright \{\!|\, c, z \,|\!\} \,[\![\, \{\!|\, c, z \,|\!\} \,]\!]\, BUFF^1_{IO}(LR_1, LR_2)$$

Proof. The proof of this lemma is carried by case analysis and by contradiction, showing that for each possible case where the process deadlocks there is a possible communication to perform.

The process deadlocks when P and the two buffers within $BUFF^1_{IO}$ are stuck. Therefore, in this proof, we analyse each case where $BUFF^1_{IO}$ may deadlock. These are: (1) when the buffers are empty and no output of P comes out and (2) when at least one buffer is full.

In the first case, every time both buffers are empty, the process P has communicated the same number of events through c and z. In this case, the last event communicated by P cannot be an output; if so, the

buffer would not be empty. Therefore, since P is buffering self-inject compatible, P may output something in this case. These processes do not deadlock.

In the second case, at least one buffer is full. We may, therefore, show that if it deadlocks, then we have a contradiction.

□

Based on Theorem 5.3, we may characterise the assertion for buffering sel-injection compatibility as the following deadlock check.

```
assert not PROJ(P,{i, o}) [| {| i, o |} |] BUFFIO(LR1, LR2):[deadlock free [F]]
```

`LR1` and `LR2` provide the necessary renaming for communicating with `BUFFIO`.

Using these assertions, we were able to rigourously apply (and automatically verify) the systematic development approach to our example as we discuss in the next section.

6. Experiments on the Case Study

The experiments consisted in verifying the CSP scripts[1] of the dining philosophers using FDR2, and collecting the overall verification time. The experiment was executed on an Intel Core i7-2600K, 3.40GHz, with 16Gb RAM, 160 GB SSD disk, running Ubuntu 9.10 (64 bits). The data were collected for both development approaches: standard deadlock check and verifying all side conditions required to apply the composition rules. Furthermore, the data for the standard deadlock check was collected for two different views: checking for deadlock after each composition (STEP), and checking for deadlock only at the final composition (GLOBAL). Also, we consider the proposed rule-based strategy both with metadata (METADATA) and without metadata (NO METADATA).

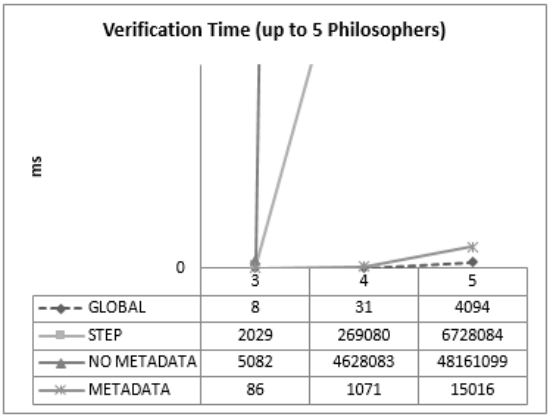
Our experiment was executed in two phases. The first phase considered a network of up to 5 philosophers (see Figure 10a). It aimed to demonstrate the improvement in the verification time by using metadata. In this phase, the time without the use of metadata proved to be much higher than that with the use of metadata. This result demonstrated the infeasibility of the approach if metadata is not considered. Furthermore, the time for standard verification of a step-by-step view was also very high. Based on the results of the first phase, we focused on the most efficient verifications of both approaches in the second phase of the experiment, which considered a network of up to 7 philosophers. In Figure 10b we present the results of the original verification of the whole system and the systematic development with the use of metadata.

The effort for checking the conditions for the rule-based application was alleviated by removing some of the side conditions in both phases of the experiment. Simple conditions based on set theory were verified using the SAT solver MiniSat 2.2. The cost of this verification, although insignificant in most cases, was added to the cost of each experiment. Furthermore, since deadlock-freedom is guaranteed by construction [OSA+13], further application of composition rules to components that result from previous compositions do not need to check for their deadlock-freedom. Further theorems also guarantee deadlock-freedom of protocol implementations of deadlock-free processes [OSA+13]. In [Ros98], it is demonstrated that if a process has no hiding and no unguarded recursion (*i.e* syntactic restrictions), it is divergence free. In [Ros98], it is also demonstrated that the checking for finite output property is irrelevant if we are using finite buffers. Finally, we also considered optimisations based on properties guaranteed by the enriched rules that use process metadata and optimisations based on theorems [RSM09] for systems with replicated components.

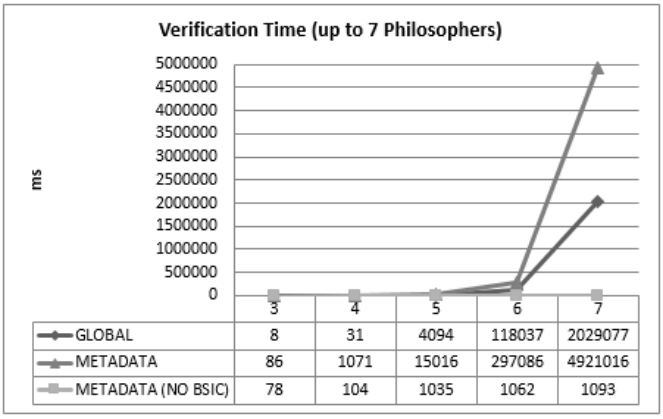
The results of the first phase are presented in Figure 10a: the results of the analysis related to Levels STEP and NO METADATA proved to be very high. The second phase of the experiment focused on the most optimised GLOBAL and METADATA (Figure 10b).

With the use of metadata and the optimisations presented above, the METADATA approach presented a gain of 99% against the sum of the verification time of each individual composition in a 5 philosophers network (see Figure 10a). It, however, presented a loss of 266% if compared with the GLOBAL approach. The loss decreases as we increase the number of instantiations of the parameterised protocol implementations. In Figure 10b, the METADATA approach presented a loss of 142% for networks with 7 philosophers. Furthermore,

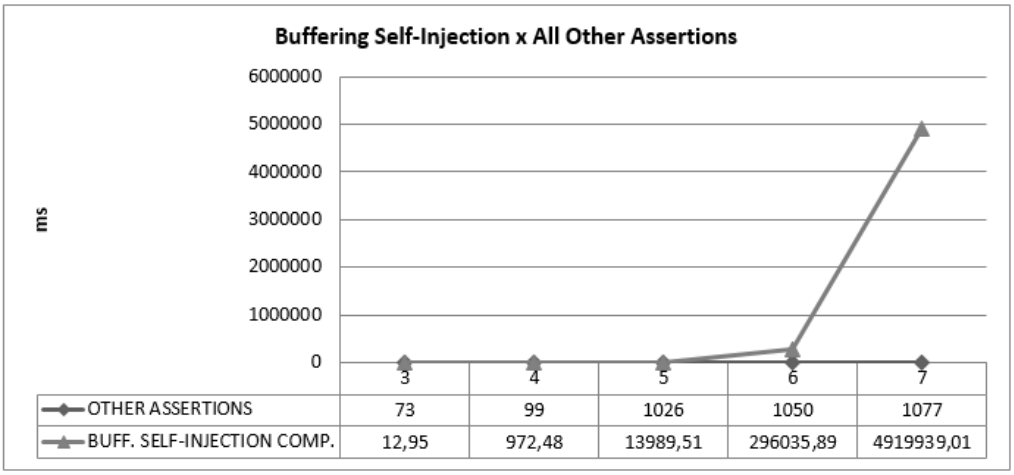
[1] The CSP scripts of all experiments can be downloaded from `http://www.dimap.ufrn.br/~marcel/research/compass/facj2016`



(a) Phase 1



(b) Phase 2



(c) Costs of Buffering Self-Injection

Fig. 10. Summary of Experiment Results

the systematic approach provides a better understanding induced by an incremental and systematic system construction.

Nevertheless, it is possible to observe in Figure 10b that both approaches present an exponential growth, which indicates scalability issues. Further investigation identified one of the side-conditions of the *reflexive composition* (buffering self-injection compatibility), the only non-compositional one, as the bottleneck of the approach, representing 99.98% of the overall cost in the verification of a network of 7 philosophers with metadata. Figure 10c presents the exponential growth of the verification time of buffering self-injection compatibility and a linear growth of the remaining assertions. As a matter of fact, however, if we do not consider this last non-compositional side-condition on buffering self-injection compatibility (BSIC), presented as METADATA (NO BSIC) in Figure 10b, our systematic approach presents a linear growth of the analysis cost.

These results demonstrate that the side condition on buffering self-injection compatibility of the *reflexive composition* needs to be avoided. One possible optimisation is based on the use of properties that are inherent to particular architectural styles, as we discuss in the next section.

7. Revisiting $\mathcal{BRICK}$: efficiency through behavioural pattern

As explained, 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 performing local analyses to guarantee deadlock freedom [Mar96, Ros98, AOS+14, ASW14].

Behavioural patterns have been introduced in the CSP context by Martin and Roscoe [Mar96, Ros98]. In [AOS+14, 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 components 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 components behave as users and which components behave as resources. This classification differentiates the components and enables the application of specific restrictions, depending on whether a component is a resource or a user. In the case of this pattern, the behaviour of a user and of a resource are restricted differently.

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 component 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 network might have and their alphabets.

In this section, we present an extension to $\mathcal{BRICK}$ that allows the integration of these behavioural patterns into it. 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 elements of the system that are relevant for pattern conformance. The new metadata, enriched with the relevant pattern elements, is denoted by the K^{+}_{PATT} tuple: it keeps track of pattern conformance throughout compositions. Next, the extended metadata needs to be accommodated by the contracts. This requires a small change to definition of contracts: we introduce a function to enrich a contract with this new metadata. The new function, $Enrich_{PATT}$, takes an enriched $\mathcal{BRICK}$ contract and enriches it further with this new metadata. This function can be seen as a variation of the *Enrich* function previously presented. Note that the steps described need to be concretely described (or specialised) for the actual pattern to be introduced.

In the next step to introduce a pattern in $\mathcal{BRICK}$, an initial validation step is required. This step is needed for guaranteeing that the initial contracts are behavioural and structural compliant to the pattern. This initial step is denoted by the predicate $INIT_{PATT}$. Finally, the rules must be extended to deal with the pattern conformance. For this matter, the side conditions of the rules must be modified to guarantee that the compositions preserve pattern adherence. Also, as the metadata is extended to deal with the elements of the pattern, the clauses for calculating the metadata resulting from the composition must be modified to handle the new element of metadata. In summary, the steps to introduce a pattern in $\mathcal{BRICK}$ are:

- Add metadata to handle pattern elements: K^{+}_{PATT}
- Add a function to further enrich the contract with the new elements of metadata: $Enrich_{PATT}$
- Add initial validation: $INIT_{PATT}$
- Extend rules modifying side conditions and adding clauses to calculate the new elements of metadata.

As we describe later, the $PATT$ subscript needs to be instantiated to denote the particular pattern adopted. For instance, here we introduce the resource allocation pattern, which is represented by the RA subscript.

To illustrate how this framework can be used to introduce an actual pattern, in what follows, we introduce the resource allocation pattern in $\mathcal{BRICK}$. The choice of the pattern is in no form arbitrary. This pattern can be used to efficiently verify that our case study, the asymmetric dining philosophers system, is deadlock free. This intended choice of the pattern should not be seen as a lack of expressiveness of our framework, rather we make this intended choice to demonstrate how efficient this integration can be in practice, by verifying

our case study. As a matter of fact, the other patterns can be integrated to $\mathcal{BRICK}$ using this framework in a very similar fashion.

Before the integration of the pattern into $\mathcal{BRICK}$, we introduce the pattern itself. The resource allocation pattern can be used to model systems where components are competing for some resources. The version that we introduce here is a slight modification from the one presented in [ASW14]. 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 signalling acquisition between components *CtrSource* and *CtrTarget*
- *rel*(*CtrSource*, *CtrTarget*): represents the event used for signalling release between components *CtrSource* and *CtrTarget*
- *ack*(*event*): the event used to acknowledge an event of either acquisition or a release
- *resource*(*CtrUser*): the sequence of resources representing the order in which the user *CtrUser* acquires them
- *user*(*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 constrain the behaviour of the user and resource components. This restriction is not imposed on the complete behaviour of the components but in a particular subset. As deadlocks can only happen in $\mathcal{BRICK}$ components due to some kind of miscommunication, the behaviour that needs to be restricted to avoid this problem is the one related with communication. Hence, we use an abstraction function in the behaviour of the components to conceal the events that are not linked with synchronisation and therefore cannot contribute to a deadlock. This abstraction function is given by *Abs*(*Ctr*), which is defined below as the abstraction of the behaviour of *Ctr* ($\mathcal{B}_{Ctr}$) on all events on the communication channels in $\mathcal{C}_{Ctr}$.

Definition 7.1 (Abstraction function). Let *Ctr* be an I/O contract. The abstraction of the behaviour of *Ctr* considering the behaviour related to communication is given as follows.

$$Abs(Ctr) \mathrel{\widehat{=}} \mathcal{B}_{Ctr} \upharpoonright (\bigcup_{c:\mathcal{C}_{Ctr}} \{\!|\ c\ |\!\})$$

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 *resources*(*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.

```
UserSpec(Ctr) =
    let Acquire(s) = if s != <> then acquire(Ctr,head(s)) -> ack(acquire(Ctr,head(s))) ->
                                                            Acquire(tail(s))
                     else SKIP
        Release(s) = if s != <> then release(Ctr,head(s)) -> ack(release(Ctr,head(s))) ->
                                                            Release(tail(s))
                     else SKIP
        User(s) = Acquire(s);Release(s);User(s)
    within User(resources(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.

```
ResourceSpec(Ctr) =
    let CtrUsers = users(Ctr)
        Resource = [] CtrU : CtrUsers @ acquire(Ctr,CtrU) -> ack(acquire(Ctr,CtrU)) ->
                                        release(Ctr,CtrU) -> ack(release(Ctr,CtrU)) ->
```

```
                                    Resource
within 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 $\sqsubseteq_{\mathrm{F}}$. The predicate used to represent the conformance relation applied to each component is given as follows. Note that, even though this restriction constraints 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 7.2 (Resource allocation behavioural restriction). Let *Users* and *Resources* be the sets of user and resource contracts, respectively.

$$BehaviourRA \mathrel{\widehat{=}} Behaviour(Users, \texttt{UserSpec}, \sqsubseteq_{\mathrm{F}}) \wedge Behaviour(Resources, \texttt{ResourceSpec}, \sqsubseteq_{\mathrm{F}})$$
where
$$Behaviour(S, Spec, \oplus) = \forall\, Ctr : S \bullet Spec(Ctr) \ \oplus \ Abs(Ctr)$$

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

$$partitions(S, T, U) \mathrel{\widehat{=}} S = (T \cup U) \wedge (T \cap U) = \emptyset$$

In the conditions that follow, we constraint the channels of the initial contracts. For the sake of simplicity, as $\mathcal{BRICK}$ 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 Ctr_1 to interact with contract Ctr_2 is denoted by the function $RACh(Ctr_1, Ctr_2)$, which yields all events used for interactions between components Ctr_1 and Ctr_2: the events from contract Ctr_1, used for acquisition, release, and their respective acknowledgement events in the interaction with Ctr_2.

The next condition, *controlledAlphabet*, imposes that the alphabet used for communication by a given set of contracts *Ctrs* must be composed by resource allocation channels.

$$controlledAlphabet(Ctrs) \mathrel{\widehat{=}}$$
$$\forall\, Ctr \bullet \bigcup_{c:\mathcal{C}_{Ctr_1}} \{\!|\ c\ |\!\} = \bigcup_{Ctr':Ctrs\setminus\{Ctr\}} \{RACh(Ctr, Ctr') \mid (Ctr, Ctr') \in \operatorname{dom} RACh\}$$

The three conditions that follow 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 $\mathcal{BRICK}$ model, connections are made by a composition, which links different channels via buffers.

The condition *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*.

$$paired(Ctrs) \mathrel{\widehat{=}} \forall\, Ctr_1, Ctr_2 : Ctrs \bullet Ctr_1 \in users(Ctr_2) \Leftrightarrow Ctr_2 \in \operatorname{ran} resources(Ctr_1)$$

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

$$\begin{aligned} consistent(Users, Resources) \equiv\ & \forall\, Ctr : Users \bullet \operatorname{ran} resources(Ctr) \subseteq Resources \wedge \\ & \quad \operatorname{ran} resources(Ctr) = \{Ctr' \mid (Ctr, Ctr') \in \operatorname{dom} RACh\} \\ & \wedge \forall\, Ctr : Resources \bullet users(Ctr) \subseteq Users \wedge \\ & \quad users(Ctr) = \{Ctr' \mid (Ctr, Ctr') \in \operatorname{dom} RACh\} \end{aligned}$$

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

channels that have been connected.

$$connected(connections) \mathrel{\widehat{=}} connections = \{(RACh(Ctr_1, Ctr_2), RACh(Ctr_2, Ctr_1)) \mid (Ctr_1, Ctr_2) \in \operatorname{dom} RACh\}$$

Finally, the condition *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.

$$strictOrdered(Users, S, \preceq) \mathrel{\widehat{=}} \forall\, Ctr : Users \bullet resources(Ctr) \preceq S$$

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

Definition 7.3 (Resource allocation structural restriction). 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 respects the strict order.

$$\begin{aligned} StructureRA \mathrel{\widehat{=}}\ & partitions(Ctrs, Users, Resources) \land controlledAlphabet(Ctrs) \\ & \land paired(Ctrs) \land consistent(Users, Resources) \land connected(connections) \\ & \land strictOrdered(Users, S, \preceq) \end{aligned}$$

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 *StructureRA* and *BehaviourRA* predicates.

This pattern prevents deadlock for a simple reason. The only way of attaining a deadlock state is by a cycle of ungranted requests between components, and if a system meets the aforementioned restrictions such a cycle is not possible. Note that the only way of this happening for this system is a path 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 a cycle cannot occur, since if it could it would violate the requirement of S being a strict order. The reader should refer to [ASW14] for more details.

We now demonstrate how the proposed framework can be integrated to $\mathcal{BRICK}$. We begin by extending the metadata to deal with the pattern requirements. The only information that needs to be kept throughout the composition process is the connection obligations that each component have. Note that, from all the conditions that the pattern imposes the only one that is not imposed on the initial components is the one restricting the connections made. This implies that the metadata must be enriched with some information to disable undesired compositions, i.e. the compositions that are not allowed by the pattern. Hence, we extend each of the initial component metadata with a set of pairs of channels, denoted by *Con* (Connections), which represents the expected future compositions of the component. This extension is defined as follows.

Definition 7.4 (K^+_{RA}). Let *Con* be a set of pairs of channels. The extended metadata for the resource allocation pattern, denoted by K^+_{RA}, is defined as:

$$K^+_{RA} \mathrel{\widehat{=}} \langle Prot^{\mathcal{K}}, CTX^{\mathcal{K}}, DProt^{\mathcal{K}}, Dec^{\mathcal{K}}, Con^{\mathcal{K}} \rangle$$

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

Next, we introduce the function to further enrich an enriched I/O Contract, as presented in Section 4. The function $Enrich_{RA}$ constructs resource allocation components from regular enriched components and their already known resource allocation metadata element, *Con*. It simply replaces the metadata element of the contract by the new metadata structure defined. It is formally defined as follows.

Definition 7.5 (Resource allocation enriched component contract). Let Ctr^e be an enriched component contract and K^+_{RA} as previously defined. A resource allocation enriched component contract that includes Ctr^e is defined as $Enrich_{RA}(Ctr^e, Con^{\mathcal{K}})$, where:

$$Enrich_{RA}(Ctr^e, Con^{\mathcal{K}}) = \langle \mathcal{B}_{Ctr^e}, \mathcal{R}_{Ctr^e}, \mathcal{I}_{Ctr^e}, \mathcal{C}_{Ctr^e}, \mathcal{K}^+_{RA} \rangle$$

The next step of this extension to $\mathcal{BRICK}$ introduces the initial validation required by the pattern. 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 *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 7.6 ($INIT^{+}_{RA}$).

$$INIT^{+}_{RA} \mathrel{\widehat{=}} BehaviourRA \land partitions \land disjointEvents \land controlledAlphaphabet \land paired \land consistent \land strictOrdered \land connectionObligations$$

where

$$connectionObligations \mathrel{\widehat{=}} \forall\, Ctr : Ctrs \bullet Con_{Ctr} = \{(RACh(Ctr, Ctr'), RACh(Ctr', Ctr)) \mid (Ctr, Ctr') \in \operatorname{dom} RACh\}$$

Finally, we present the extensions of the rules that adds a clause for calculating the new metadata element and further side conditions that accommodate the demands of the pattern. For this purpose, 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 new metadata element. This extension is a mere union of the *Con* structures for each of the contracts participating in this composition. This extension is formalised as follows.

Definition 7.7 (Resource allocation enriched interleaving composition). Let the contracts $Ctr_1^{e^+} = Enrich_{RA}(Ctr_1^e, Con_1^{\mathcal{K}})$ and $Ctr_2^{e^+} = Enrich_{RA}(Ctr_2^e, Con_2^{\mathcal{K}})$ be two resource allocation enriched component contracts, such that Ctr_1^e and Ctr_2^e satisfy the condition of the enriched interleaving composition rule. Then, the resource allocation enriched interleaving composition of $Ctr_1^{e^+}$ and $Ctr_2^{e^+}$ is given by:

$$Ctr_1^{e^+} \,[\!|||\!]^{e^+}\, Ctr_2^{e^+} = Enrich_{RA}(Ctr_1^e \,[\!|||\!]^e\, Ctr_2^e, Con_1^{\mathcal{K}} \cup Con_2^{\mathcal{K}})$$

Note that we use the $Enrich_{RA}$ 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 *Con* for each rule, based on the enriched operators presented in Section 4.

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 participating are allowed to engage in this connection. As explained, this is expressed by the *Con* metadata, so if component Ctr_1 is being composed using channel ch_1 with component Ctr_2 using channel ch_2, they are both willing to engage in this composition if $(ch_1, ch_2) \in Con_1$ and $(ch_2, ch_1) \in Con_2$. 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 *Con* structures of each component but excluding the pair of channels used for composition.

Definition 7.8 (Resource allocation enriched communication composition). Let the component contracts $Ctr_1^{e^+} = Enrich_{RA}(Ctr_1^e, Con_1^{\mathcal{K}})$ and $Ctr_2^{e^+} = Enrich_{RA}(Ctr_2^e, Con_2^{\mathcal{K}})$ be two resource allocation enriched ones, and ic and oc two channels, such that $(ic, oc) \in Con_1^{\mathcal{K}} \land (oc, ic) \in Con_2^{\mathcal{K}}$, and Ctr_1^e and Ctr_2^e satisfy the condition of the enriched communication composition rule for ic and oc. Then, the resource allocation enriched communication composition of $Ctr_1^{e^+}$ and $Ctr_2^{e^+}$ via ic and oc is defined as:

$$Ctr_1^{e^+}[ic \leftrightarrow oc]^{e^+}\, Ctr_2^{e^+} = Enrich_{RA}(Ctr_1^e[ic \leftrightarrow oc]^e\, Ctr_2^e, (Con_1^{\mathcal{K}} \setminus \{(ic, oc)\}) \cup (Con_2^{\mathcal{K}} \setminus \{(oc, ic)\}))$$

For the feedback rule, the side conditions must be modified and the metadata calculation for the new metadata element must be defined. Concerning the side conditions, the change needed is very similar to the one presented in the communication rule. For a feedback composition to be valid in this new settings, the component participating in this composition must be allowed to make a connection in the designate channels. This means that the pairs of channels denoting this connection must belong to *Con*. Considering the clause for calculating the *Con* for this rule, it is also very similar to the one defined for the communication rule. After the composition has been established, the resulting component is no longer allowed to make the just established connection. Therefore, the pairs of channels representing this connection are removed from its *Con*.

Definition 7.9 (Resource allocation enriched feedback composition). Let the component contract $Ctr^{e^+} = Enrich_{RA}(Ctr^e, Con^{\mathcal{K}})$ be a resource allocation one, and ic and oc two channels, such that $\{(ic, oc), (oc, ic)\} \subseteq Con^{\mathcal{K}}$, and Ctr^e satisfies the conditions of the enriched feedback composition rule via ic and oc. Then, the resource allocation enriched feedback composition of Ctr^{e^+} via ic and oc is defined as:

$$Ctr^{e^+}[ic \hookrightarrow oc]^{e^+} = Enrich_{RA}(Ctr^e[ic \hookrightarrow oc]^e, Con^{\mathcal{K}} \setminus \{(ic, oc), (oc, ic)\})$$

The extension of the reflexive composition rule is the most interesting case. It has its side conditions completely modified (rather than just adding new conditions and for the previous rules) and a clause for the calculation of the new metadata element defined. The reason for changing the side conditions is the need for removing the buffering self-injection compatibility from the side condition of this rule. This is the motivation for the use of behavioural patterns. For this rule, the side condition becomes the following. The component must be willing to connect the two channels involved in the composition, according to the *Con* element (this part is very similar to the other rules) and, in addition, this connection must finish the design of the system, i.e. there must be no intentions of connections left in the *Con* after this composition. Hence, the *Con* after this composition must be the empty set. The clause for calculating *Con* for the composition is exactly the one used in the feedback rule. After the composition takes place, the pairs of channels representing the intent of connection must be removed from *Con*. Note that, there is no more need for verifying buffering self-injection compatibility when using this rule. Henceforth, by the use of the pattern, there is only a need for verifying whether the connections are valid and it finishes the intended design. Therefore, no global analysis is needed; instead, only local verifications are carried out.

Definition 7.10 (Resource allocation enriched reflexive composition). Let the component contract $Ctr^{e^+} = Enrich_{RA}(Ctr^e, Con^{\mathcal{K}})$ be a resource allocation enriched one, and ic and oc two channels, such that $\{(ic, oc), (oc, ic)\} \subseteq Con^{\mathcal{K}}$, and $(Con^{\mathcal{K}} \setminus \{(ic, oc), (oc, ic)\}) = \emptyset$. Then, the resource allocation enriched reflexive composition of Ctr^{e^+} via ic and oc is defined as:

$$Ctr^{e^+}[ic \hookleftarrow oc]^{e^+} = Enrich_{RA}(Ctr^e[ic \hookleftarrow oc]^e, Con^{\mathcal{K}} \setminus \{(ic, oc), (oc, ic)\})$$

The extended rules provide $\mathcal{BRICK}$ with a fully local analysis strategy for systems that are compliant with the resource allocation pattern. This impacts the applicability of the rules for a set of larger systems. The verification of the side conditions of the reflexive rule, which was intractable for large set of systems using the original enriched rules, becomes tractable to those adherent to this behavioural pattern. This is further discussed later based on a practical experiment conducted.

Our extension to $\mathcal{BRICK}$ to accommodate the resource allocation pattern indeed prevents deadlock of arising. First of all, considering the first three rules (interleaving, communication and feedback), we have only strengthen the side conditions. As we have already demonstrated that the original rules with their side conditions avoid deadlocks, and these proposed new rules still meet their side conditions, it is clear that the new rules with their strengthened side conditions still avoid deadlocks of arising. Considering the reflexive rule, it also prevents the creation of a deadlocked system. As demonstrated in [ASW14], the design of a system according to the resource allocation pattern prevents a deadlock of arising. After the reflexive composition is made, the final system complies to the resource allocation pattern. Note that, after this composition, the system must have satisfied all the restrictions imposed by the behavioural and structural condition with the exception of the connected clause, since it must have passed the initial step of validation. Additionally, it must have performed only expected connections as imposed by the extension of the side conditions that verify whether a connection is a valid one. Furthermore, all the expected connections must have been performed, since the side conditions of the new reflexive rule states that the resulting contract must have no connections obligations, i.e. an empty *Con*. Hence, this final system obeys to all the restrictions of the pattern; hence, by consequence, it is compliant to the pattern.

7.1. Revisiting the dining philosophers case-study

After the introduction of the resource allocation pattern in $\mathcal{BRICK}$, we revisited the case study to demonstrate how this new version of $\mathcal{BRICK}$ could be applied to it. Moreover, we conducted a practical experiment to evaluate the benefits on efficiency that it brings to the analysis of this case study.

First of all, we use CSP_M to implement the elements of the $\mathcal{BRICK}$ strategy and the elements of the pattern. In order to manipulate the elements of the contract and the elements of the pattern, 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 since we use FDR2's pattern matching.

In order to use the strategy, one has to identify 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.

```
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.picksack
ack(pfk.idf.idp.picksup) = pfk.idf.idp.picksack
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) = <>

users(FK.id) = {PH.id,PH.prev(id)}
users(Other) = {}
```

After defining the elements of the pattern, we are able to start using $\mathcal{BRICK}$ incremented with the resource allocation pattern. As we plan to build the same dining philosophers built in Section 3, we define the initial contracts exactly as presented there. The *Con*, which is the new piece of information that needs to be defined on top of it is defined as follows.

```
Con(FK.id) = {(RACh(FK.id,PH.idp),RACh(PH.idp,FK.id)) | idp <- {id,prev(id)}}
Con(PH.id) = {(RACh(PH.id,FK.idf),RACh(FK.idf,PH.id)) | idf <- {id,next(id)}}
```

This are set comprehension expressions that define all the connections expected by the forks and philosophers components, 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 $INIT_{RA}$. 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 *controlledAlphabet* clause is satisfied. This makes 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 resource allocation enriched rules for composition, we are able to build our system using the same strategy adopted in Section 3. The only modification here is the use of the resource allocation enriched rules presented in this section instead of the use of the original enriched rules presented in Section 4. For conciseness, as the compositions are basically the same as those presented in Section 3, we do not present their details here. Using the resource allocation 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 conducted a practical experiment using the same approach as that presented in Section 6. As in that experiment, the verification of side condi-

Size	ms
3	90
6	1074
7	1104
10	2098
100	32098
1000	438824
10000	17980081

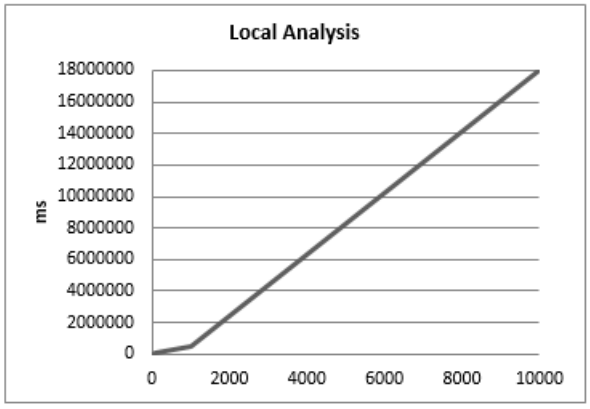


Fig. 11. Results of Experiments Using Local Analysis

tions involving set operations were achieved using MiniSat 2.2. 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, $\mathcal{BRICK}$, which needed exponential time for 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 10000 philosophers and 10000 forks in five hours. As a matter of fact, using our optimisation of $\mathcal{BRICK}$, we were able to analyse a network with 20000 processes. Hence, we were able to go further than one of the main strategies used for deadlock verification in the CSP world, the Deadlock Checker [Mar96], a tool designed for the specific purpose of deadlock verification, which is bounded to verify systems with up to 10000 processes.

In [BBNS10], the authors present a compositional method for the verification of component-based systems described in a subset of the behaviour-interaction-priority (BIP) language encompassing multi-party interaction without data transfer. Their method has been implemented in the D-Finder tool and has been applied for checking deadlock-freedom. Their experimental results on the dining philosophers, presented a deadlock analysis of a network of 3001 philosophers in 54m34s.

8. Industrial Case Study

In this section, we discuss an industrial case study which is more oriented towards a practical component based system. The case study was suggested by one of our industrial partners in the COMPASS project[2], Bang & Olufsen (B&O). It is the formalisation of the protocol used for dynamic integration of their devices, where one must play the role of a leader. Here, we present this case study without describing all the technical details, which can be found in [OSA+13, AOS+14]. Whereas in [AOS+14] this case study is explored in the context of pure CSP, here we consider its systematic development and analysis in $\mathcal{BRICK}$. For conciseness, we give an overview of its component architecture, the communication pattern adopted for local analysis, and the results of the analysis that support our claims; the details are in [OSA+13].

A home Audio/Video (AV) network consists of several devices (such as audio, video, gateway and legacy audio devices) which may be produced by competing manufacturers and distributed across a user's home. In particular, the network can be considered as a System of Systems (SoS) because 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.

[2] `http://www.compass-research.eu/`

- 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 devices join or leave the SoS during streaming or browsing operations; devices can be turned off by users or enter power-saving mode.
- While devices have no interdependence, CSs rely on each other in order to deliver the emergent behavior that fulfills the SoS goal.

Constituent systems (devices) 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. The network, therefore, must always be able to identify a leader (the publisher). Conceptually, the device network contains two global states:

- The publisher-subscriber state: a single publisher (the leader) is present and the device network can guarantee availability and consistency of user experience. All other connected devices are subscribers (followers), and newly joined devices are undecided, until they learn the identity of the leader.
- The election state: no publisher is present and the user experiences are unavailable. In this state all connected devices are undecided.

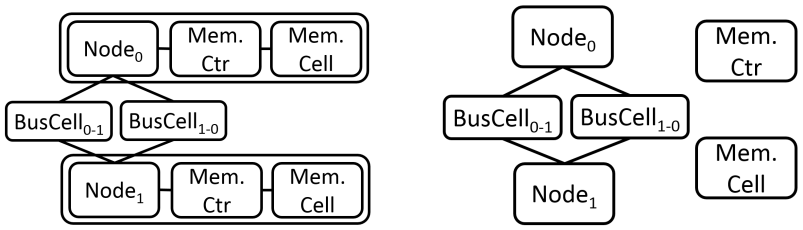
In the election state the devices in the network execute a leadership protocol, in which each device reacts to a set of local transition rules that will guarantee the desired emergent property. Therefore, the network is inherently asynchronous and the algorithm must take the following cases into consideration:

- The algorithm must handle the disappearance of leaders and the appearance of new contenders for leadership. This is because, in practice, devices may enter a power-saving state, restart because of defects, or be turned on or off by their users, at any time (during or after an election).
- There is no coordination of when an election is started, and so any device can initiate an election independently. This is due to the fact that communication is asynchronous, with some latency in the network, which increases the likelihood of simultaneously initiated elections.

Such a dynamic environment considerably increases the risk of the protocol reaching a deadlock state. In order to eliminate this risk, we developed a formal model of the B&O leadership election protocol using $\mathcal{BRICK}$ (see [OSA+13]) and showed that it is deadlock-free. B&O invests in a formal analysis of this kind because of its desire to develop and analyse models in the early design stages, before expensive implementation commitments are made.

In our model, illustrated in Figure 12a for a 2-node configuration, devices 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 devices. 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 of 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.

Each node `i` is initially turned off, where it behaves as the process `OffNode(i,LOWER_LIMIT_PET)`. In



(a) Communication graph of a 2-node configuration.

(b) Essential components after decomposition.

Fig. 12. Views of the Leadership Election Constituent Systems.

this state, it can only turn on. Before turning on, however, it signals to all the bus cells that have this node as a sender that it has turned on (behaviour given by the process `BroadCastControl(id,isOn)`). After this, it starts behaving as the `OnNode` process. 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 process `Node` composed sequentially with the `Fail` process. This means that the main behaviour of a turned on node is given by `Node` and, at some point, this process may successfully terminate, 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 this messages to the bus cells having it as a sender. This broadcast behaviour is given by `BroadCastControl(id,isOff)`. Finally, the node behaves as the `OffNode` process.

```
OffNode(id, prior) = BroadCastControl(id,isOn); OnNode(id, max(LOWER_LIMIT_PET, prior-1))
OnNode(id, prior) = Node(id, <>, undecided, prior); Fail(id,prior)
Fail(id,prior) = BroadCastControl(id,isOff); OffNode(id, prior)
```

The broadcast behaviour `BroadCastControl(id,status)` simply signals the `status` of the node `id` to all the bus cells that have this node as a sender.

```
BroadCastControl(id, status) =
    let BroadCast(<>) = SKIP
        BroadCast(<a>^list) =
            if a == id then BroadCast(list)
            else sender.id.a.out!status -> BroadCast(list)
    within BroadCast(SEQ_NEIGHBOURS)
```

The behaviour of the node when it is ready to communicate is given by the process `Node`. At this point its behaviour is cyclic, and controlled by the sequence of nodes in the second argument. When this sequence is empty, the node is ready to broadcast data. The node, however, might behave as `SKIP`, instead of sending data to another peer. As described before, when the node behaves as `SKIP`, a failure has occurred and the node starts behaving as the process `Fail`. During its data broadcasting, the node sends its `claim` and `petition` to each of the neighbouring nodes (via process `BroadCastData`). In our model, we assume that this broadcast is uninterruptible, meaning that the node cannot fail during its data broadcasting. Moreover, this sending process is made in a two-step interaction, where the node sends the data first and then wait for an acknowledgement. The sequence `SEQ_NEIGHBOURS = <0..N>` is a sequence containing the ids of all nodes, which guides the interaction between nodes.

```
Node(id, <>, myclaim, mypetition) = BroadCastData(id, myclaim, mypetition) |~| SKIP
BroadCastData(id, claim, priority) =
    let BroadCast_Aux(<>) = Node(id, SEQ_NEIGHBOURS, claim, priority)
        BroadCast_Aux(<a>^list) =
            if a == id then BroadCast_Aux(list)
            else sender.id.a.out.pack!claim.priority -> sender.id.a.in.ack ->
                 BroadCast_Aux(list)
```

```
    within BroadCast_Aux(SEQ_NEIGHBOURS)
```

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-step interaction. First the node request some data from a bus cell, then it receives some actual data. Observe, again, that the node, when receiving data from each of the other nodes, may fail and behave as `SKIP`. Furthermore, this two step interaction for receiving is uninterruptible, meaning that the node cannot fail between a request and receiving data.

```
Node(id, <a>^list, myclaim, mypetition) =
    if a != id then (receiver.a.id.out.req -> receiver.a.id.in?x:TPACKS ->
                        node_mem.id.out.mem_pack.a!getValue(x) ->
                        Choice(id, list, myclaim, mypetition))
                   |~| (receiver.a.id.out.isOff -> SKIP)
    else Choice(id, list, myclaim, mypetition)
```

After receiving data from a peer, the election process is triggered, via the process `Choice`, which is responsible for choosing the subsequent behaviour according to the claim of the node.

```
Choice(id, list, myclaim , mypetition) =
    if myclaim == undecided then Undecided(id, list, mypetition)
    else if myclaim == leader then Leader(id, list, mypetition)
    else if myclaim == follower then Follower(id, list, mypetition)
    else sender.myclaim -> STOP
```

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.

```
Follower(id, list, mypetition) =
    node_mem.id.out.reqLeaders -> node_mem.id.in.leaders?valLeaders ->
        if valLeaders == 0 then Node(id, list, undecided, mypetition)
        else Node(id, list, follower, mypetition)
```

A leader also looks at the number of leaders. It becomes `Undecided` if this is greater 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.

```
Leader(id, list, mypetition) =
    node_mem.id.out.reqLeaders -> node_mem.id.in.leaders?valLeaders ->
        if valLeaders > 0 then Node(id, list, undecided, mypetition)
        else if list == <> then Node(id, list, leader, min(UPPER_LIMIT_PET,mypetition+1))
             else Node(id, list, leader, mypetition)
```

The `Undecided` node 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 parameters. 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.

```
Undecided(id, list, mypetition) =
    node_mem.id.out.reqLeaders -> node_mem.id.in.leaders?valLeaders ->
    node_mem.id.out.reqHpetition -> node_mem.id.in.hpetition?highest ->
    node_mem.id.out.reqHpetitionid -> node_mem.id.in.hpetitionid?highestid ->
    ( let myclaim = if valLeaders > 0 then follower
                    else if list == <> then
                            if (highest == mypetition and highestid < id)
                                or highest < mypetition
                            then leader
```

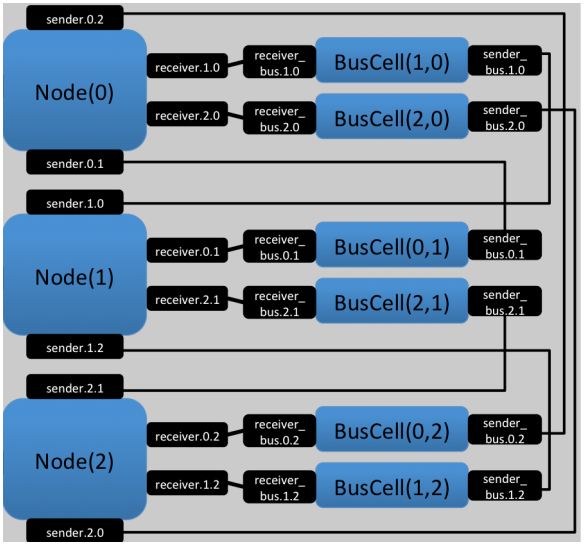


Fig. 13. Leadership election core of a 2-node configuration

```
                else follower
            else undecided
within Node(id, list, myclaim, mypetition) )
```

Communication between nodes takes place over a `Bus` that provides bidirectional communication between every pair of nodes. The `Bus` is composed of various `BusCells`, each of which provides an unidirectional channel between a source and a target node. The `BusCell` process is relatively simple and omitted here.

Based on these processes, we define the contracts of our constituent systems used to build our $\mathcal{BRICK}$ model. For the leadership election algorithm, we use another behavioural pattern to build the core of the system, i.e. the sub-system composed of nodes and bus cells illustrated in Figure 13. The *async dynamic pattern* [OSA+13] 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. 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.

After the core of the system 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.

For the experiments, we considered our pattern-based version of $\mathcal{BRICK}$, the original $\mathcal{BRICK}$ 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, although rather insignificant, were verified using MiniSat 2.2. As we have a parametrised model[3], 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 14.

The original $\mathcal{BRICK}$ 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

[3] The CSP scripts of all experiments can be downloaded from `http://www.dimap.ufrn.br/~marcel/research/compass/`

		Time (in seconds)		
N	# of Comps	AD	Metadata	Global
2	4	6,68	3,77	0,12
3	12	39,88	2723,02	47,01
5	40	424,55	*	*
10	180	8938,63	*	*
20	760	165668,38	*	*
32	1984	1157500,97	*	*

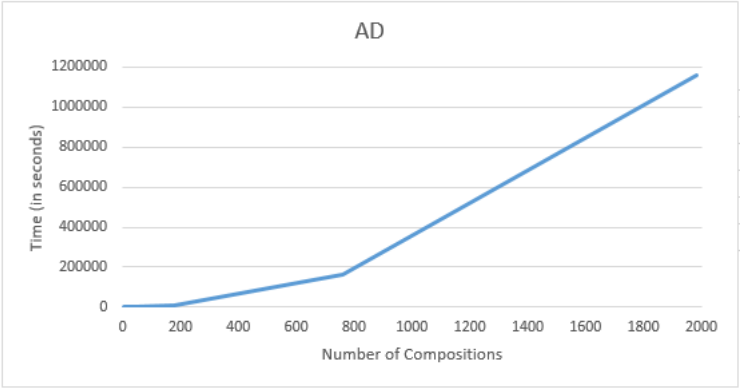


Fig. 14. Results of experiments for the leadership election

strategy goes from a total time of about 4 seconds to verify a 2-node configuration to more than 45 minutes to verify a 3-node configuration. For the global case, note again how it is affected by the explosion in the state space; this strategy is able to verify a 2 and 3-node configuration, but fails to verify whether a 5-node 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 14), 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 number of nodes 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 (a fully connected graph). 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-node configuration in about 14 days whereas the other strategies were not able to cope with more than 3 nodes.

9. Conclusions

Although component-based approaches provide mechanisms and tools for constructing systems by plugging components together, the safe construction of these systems is still a research challenge. Trustworthiness is required during several development activities, such as safe composition of third-party components or the correct adaptation of library components.

In this work, we propose an integrated constructive approach for building deadlock-free component-based systems using metadata and behaviour patterns. The approach focuses on performing analyses that are intended to address engineering concerns on compositional development. In special, we focus on component integration. The entire approach is underpinned by the CSP process algebra, which offers rich semantic models that support a wide range of process verifications, and comparisons. We also propose a fourth basic composition rule, called *reflexive composition*, to represent the general case of component architectures.

The $\mathcal{BRIC}$ component model is aligned to other models with behaviour descriptions. It focuses on (re)active components that are *input deterministic* and *output decisive*. Reuse and compositions are allowed not only to components, but also to connectors. Furthermore, it considers not only compositions between two distinct components, but also the assembly of ports of the same component. This brings more flexibility to design decisions at development. An operation for hiding information to pack components into black-boxes is also presented.

We presented 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 component 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 the overall structure of the system. The composition rules are strongly based on the use of protocols, which play an important role in the approach, and, in conjunction with other properties, help to alleviate verifications by supporting local analyses. When the side conditions based on such local analysis are not satisfied, the developer can perform a broader verification using the *reflexive composition rule*. This composition rule is more costly regarding verification, since it does not solely use protocols, but the entire component behaviour. The soundness proof of all the four rules is a contribution of this work.

We improve the verification using enriched components with metadata. Metadata is 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 order of complexity of the verifications is reduced for most of the composition rules. This is demonstrated in the experiments presented in Section 6, in which we apply our approach in a case study, *the dining philosophers*.

Nevertheless, the results presented in Section 6 demonstrate that verification of the side condition on buffering self-injection compatibility of the *reflexive composition* needs to be avoided because it presents an exponential growth in verification time, which indicates scalability issues. As a matter of fact, it is the only non-compositional one, being the bottleneck of the approach.

The exponential growth of the buffering self-injection compatibility check has been dealt with in Sections 7 and 8, where we presented an extension to $\mathcal{BRICK}$ that allows the integration of behavioural patterns into it. Further experiments using enriched composition rules that consider the resource allocation pattern (Section 7) and the async dynamic pattern (Section 8) provided some evidence that our optimisation using local analysis considerably improves the efficiency of our strategy. Using our optimisation, $\mathcal{BRICK}$ required only a linear time, allowing the strategy to be used in the development of industrial systems in a practical manner. Furthermore, scalability issues were also addressed with our optimised approach, which proved more efficient. Using our approach, we were able go further than established tools like the Deadlock Checker [Mar96] and the D-Finder tool [BBNS10], as detailed in Section 7.

Several aspects of our approach contribute to an efficient analysis in general, like the fact that it is inherently compositional and record relevant metadata information. The use of patterns help to categorise families of problems with a customised local analysis approach. We emphasise, however, that each pattern, like the Resource Allocation one, captures communication protocols of a variety of practical applications, and, therefore, is not tailored to a particular example.

In [dSOSO15] we present inheritance relations for $\mathcal{BRICK}$ components, which support a constructive design based on composition rules that preserves desired properties such as deadlock freedom. In that work, we enhance this component model with support for extensibility via inheritance. The proposed relations allow extension of functionality, whilst preserving service conformance, which we define by means of a substitutability test. We also establish an algebraic connection between component extensibility and refinement.

Besides the work on refinement and extensibility mentioned above, we are also working on local *livelock* analysis. In [OSA+13] we developed a strategy for local livelock analysis based on constructive rules similar to those that ensure deadlock freedom presented here, but with additional side conditions. The benefits of our approach were demonstrated with case studies, in which our results presented significant gains in the performance of livelock analysis in comparison with FDR and SLAP [OPRW13].

9.1. Related work

There are several different approaches to component models. As pointed out in [Wal03], each component model is designed to achieve specific goals. Furthermore, each one has its benefits and deficiencies, depending on the context in which it is analysed.

Our work does not relate with other component models that define low-level granularity components, in which contracts/interfaces capture solely syntactic information (like method signatures). Low-level granularity component models are associated to component technologies found in industry that are usually designed to support quick development or to permit the use of different programming languages in development. These are not designed for reasoning. In order to get around this limitation concerning interface represen-

tation, several authors [FLF01, LD00, LW94] propose the specification of the 'behaviour' part via pre- and postconditions and invariants. According to [Pla05], one of the key obstacles in applying these approaches to components is that they require an explicit capturing of (object) state. This may be both very hard-to-achieve and, potentially, limiting decision at an early stage of a component design.

Our approach is more related to works that support behaviour description of entities. The idea of expressing behaviour of an object as a regular process (via traces as sequences of method calls) has been published in [Nie93]. It even considers the role of client calls (in a simple case) via parallel composition. The importance of capturing behaviour of components as sequences of events for COTS components (commercial off the shelf) is emphasized also in [DR02] where a way of identifying behaviour via monitoring experiments is described.

There have been a huge number of publications on behaviour description of components and connectors [ADG98, BCD02, HLL06b, BHP06, Arb04, Sif10, CZ07]. Our approach integrates aspects from different but closely related domains. The target concrete syntax of our work is CSP, but the elements within $\mathcal{BRIC}$ component contracts (see Definition 3.1) are not directly represented by this notation. CSP is used to give the underlying semantics of our component model, and to help verifications. However, there are more suitable concrete syntaxes to represent our notions at development phase, such as Architectural Description Languages (ADLs) [MT00] or the modelling languages UML-RT [SR98], UML2 [Obj07] and SysML [OMG12]. The concepts in these languages are highly compatible with our component model, and one can benefit from using both approaches, like modelling in one language and performing verifications in another.

Our component model is based on I/O transition systems, has explicit architectural structure, and presents connectors as first class design elements. These characteristics resemble several ADL approaches, such as Wright [All97, ADG98], Darwin [MK96], PADL [BCD02], and ROOM [SGW94]. Our component model focuses on design elements, and does not take into consideration the expressiveness of programming languages as architectural programming models, such as ArchJava [ACN02], SOFA [BHP06], Fractal [BCL$^+$06], rCOS [HLL06a, CHLZ07], MASCOT3 [Geo86] and BIP [Sif10]; the design concepts in these ADLs are, however, compatible with concepts in our component model. Another related ADL is ROOM [SGW94], which later evolved to UML-RT [SR98], which in the meantime has been incorporated into UML2 [Obj07].

Despite their similarities, the representation of components in these works differs in some extent. Some consider the internal behaviour of components, e.g. [BCL$^+$06, HLL06a], other the external behaviour, e.g. [ADG98]. Some component models represent components solely by their port protocols, e.g. [CZ07], other neglects this kind of behaviour, e.g. [HLL06a]. In our work, we discriminate the external behaviour of components and their points of interactions (port protocols). Component contracts have the whole external component behaviour, or are enriched with port protocols (see component contracts and metadata in Section 3). Each kind of behaviour has its benefits in reasoning. Port protocols alleviate verifications, whereas the whole behaviour of components is essential for structural analysis of larger systems. Our component model also has operations to hide information in component contracts. The wrapping operation hides the part of the component behaviour that is not available for composition (the interaction between the sub-components of the composition). This is, however, different from the concept of *publication* presented in rCOS [ZKL10] for creating 'black-box components'. In rCOS, a publication is an abstraction of a contract that removes behavioural information from the contract.

There are several efforts on the verification of Component-based Systems [BCD02, MCM08, HJK10b, MW97, All97]. The scalability issue in compositional verification has been actively addressed in this field; compositional verification is based on the idea that the correctness check of a complex system can be divided into smaller verification tasks for its components. Here, we compare our work, not only with approaches with an explicit component model, but also with others that focus on the verification of "behavioural elements" (which may not be fully aligned with a component development method).

The work reported in [GGMC$^+$07, MCMM07, MCM07] presents an extensive study of quality properties in CBS. It discusses liveness, local progress, deadlock, fairness and robustness. We focus on the deadlock property, which is locally addressed by our compatibility notion. Therefore, deadlock freedom is preserved by our composition rules for $\mathcal{BRIC}$ components. Although not discussed, local progress is also preserved when composition rules are applied for $\mathcal{BRIC}$ components because none of the composition rules introduce livelock. Relating to fairness (of process schedules or of internal event choices), we believe that it must be performed by coordinators, which mediate component interactions. As a consequence, fairness is a property associated to a coordination purpose and that requires a specific verification, which is out of the scope of this work. Robustness is a desirable property which is not addressed by our work.

Even though there are many approaches to formally model component based systems [ADG98, AB03, IM08, HLL06b, PV02], to our knowledge the question of preserving, by construction, behavioural properties has not yet been fully systematised as we have done in this work. Despite the fact that our black-box component contracts are compatible with most component-based approaches, especially those based on CSP or CSP-like notations [Ros98, HLL06b], most approaches to date aim at verifying the entire component-based systems before implementation, but not predicting behavioural properties by construction during design. We can ensure deadlock-freedom in a constructive way, as a result of applying composition rules, as opposed to performing model checking verification after the system has been built. The compositional approach can be applied in heterogeneous systems (synchronous and asynchronous) with different topologies (tree or cyclic).

Approaches to verifying a system tend to use abstraction techniques to reduce the state space. They map a set of states of the actual system to an abstract, and a smaller set of states in a way that preserve the behaviours of the system. [ZM10] adopts counterexample guided abstraction refinement scheme to alleviate the state explosion problem of deadlock detection. It extends the classical labelled transition system models by qualifying transitions as certain and uncertain to make deadlock-freedom conservative. A similar approach is presented in [Kwi07]. It determines their sets of 'conflict-free' actions, called untangled actions. Untangled actions are compositional; synchronisation on untangled actions will not destroy their 'conflict-freedom'. Following the same approach, [CCH$^+$09] proposes a deadlock detection algorithm based on navigating and marking transitions on a dynamic synchronization dependency graph.

In PADL [BCD02] and in [MCM08] compatibility is used to detect architectural mismatches and it is shown that pairwise compatibility is a sufficient criterion to derive deadlock-freedom of an acyclic assembly from the deadlock-freedom of its local components. These approaches consider the whole behaviour of the constituent components in the composition. Differently, our approach is centred on the use of port protocols to alleviate compatibility verifications.

Closer to our approach is the work presented in [LMC10, CZ07] that performs architectural compatibility verifications based on compatibility of port protocols. The restriction in [LMC10] is that only deterministic protocols are considered. [CZ07] proposes a formal model of component interaction, in which component compatibility is verified using labelled Petri nets. In this work, the behaviour of components is represented solely by their port protocols, called *interface languages*, which contains either possible sequences of required or provided services. A request (rich) interface is compatible with a provider (rich) interface if and only if all sequences of services requested by the former can be provided by the latter. This condition reassembles our denotation definition of compatibility. However, as we deal with bidirectional I/O channels, these conditions are verified in each state of the protocol for both directions.

A notion similar to behavioural compatibility is used by [HJK10b] under the name of *neutrality*. The verification of properties for the whole component then follows from the verification step that uses only weakly deterministic port protocols. Behavioural neutrality is defined in terms of observational equivalence between the behaviour of an assembly with two connected components and the behaviour of an assembly with a single component and the binary connector replaced by a unary one. This notion plays an important role in its reduction strategy. A component neutral to another can be removed from the analysis of composition because they do not contribute with any change in the external observable behaviour of the composition. There are two restrictions in the approach: components must be weakly deterministic and in order to be neutral their input and output labels must mutually coincide. As verified in [CZ07], it is possible that one component does not use all services of another, and, therefore, that one component might output fewer events than the other one may possibly input.

Another notion related to behavioural compatibility is used in [CK96] under the name of *transparency*. In [CK96] automatically derived context constraints (restrictions imposed by the environment on subsystem behaviour) are used to construct the LTS behaviour of composed systems more efficiently. Context constraints take the form of *interface processes*, which capture the interplay of the environment of a single fixed component as part of the composition with other components. If the composition of the interface process and the fixed process results in a smaller transition system, it is substituted in the overall analysis. The correctness of the approach relies on a transparency property which requires a strong semantic equivalence between the fixed process and its composition with its interface process. Compatibility is verified by checking if the interface process is well-formed.

In [All97], the interface process associated to a port is called a *deterministic process* of a process. Compatibility of two processes is checked by verifying the refinement relationship between a process and the synchronisation of another process and the deterministic process of the former. In our work, the interface process and deterministic versions are called *contextual process*, and similarly to [All97] is used solely in

compatibility checks, rather than in a more general analysis as in [CK96]. Similarly to [All97], we check compatibility of two protocols as the refinement of a protocol by its context process synchronised with the dual protocol of the other. A dual protocol represents the most nondeterministic process that is compatible with a protocol. We use this notion as we deal with I/O processes in this work.

Similarly to [LMC10, HJK10b, Zub11, All97], compatibility verification in our composition rules are based on the compatibility of protocols. Furthermore, they have the benefit of reducing state space, and relatively low cost verification. We differ by having I/O processes that are not entirely deterministic, and by accepting bidirectional communications on a same channel. The possible existence of non-determinism in I/O processes and of bidirectional communication brings more complexity to our verifications than the works related to the notion of compatibility mentioned above [CK96, All97, BCD02, MCM08, HJK10b, LMC10]. For instance, in [LMC10, BCD02] components must be deterministic. This prevents designer from considering situations where the components take internal decision. We also include the conditions on determinism and communication directions in composition rules, which focus on different interaction patterns and allow the construction of architecture with cyclic topologies. Component metadata is used to alleviate side condition verification, such as avoiding the generation of protocols from components resulting from other compositions. As far as we are aware, no other work has used component metadata in such way. Such solution can also be used in other works [LMC10, HJK10b] to reduce the cost of their verification.

Bidirectional communication may implicitly introduce small cycles (with two components) and is not addressed by the works above, since they use compatibility in component-based systems with tree-topology structures of unidirectional channels. However, bidirectional communication is implicit in our component model, and is furthermore directly supported by our compatibility notion. Except for the work on PADL [BCD02, AB03], none of the works cited above deal with cyclic topologies. Even this approach does not present a solution to alleviate the verification of applications in such topologies. In [BCD02, AB03] deadlock-freedom is locally considered in the relationship of each component with the others in the whole cycle. Similarly to the seminal work on deadlock-freedom [Ros98], the approach needs to know the internal structure of the entire system (which is also a component) *a priori*, which is in the opposite direction of a compositional method. In our work, cyclic topologies are verified in compositional correct-by-construction approach, as soon as the cycle appears. Means to alleviate the verification are presented by the notion of *decoupled channels*.

A further important difference between checking compatibility of port protocols and checking the compatibility of entire component behaviours is that the use of explicit port behaviours makes the check for compatibility more efficient. Furthermore, as mentioned in [LMC10], this supports a gray box view of the components that is desired in CBD similar to the principle of information hiding.

Despite the benefits of protocol representation, representing the whole component is also necessary. For instance, the approach in [CZ07] abstracts the internal behaviour of components, and concentrates solely upon the behaviour exhibited by port protocols. Concentrating solely upon the behaviour exhibited by port protocols, these works indirectly restrict the structure of their systems to tree-topologies, without cycles. For the same reason, it is forbidden to assembly multiple points of interaction between components, which implicitly introduce minor cycles. Similarly, the approach forbids the verification of other emerging properties of the system, such as livelock, which emerges from the interaction of the components.

Some approaches [IM08, MW97] predict some system properties based on the properties of its constituting components. This is performed by categorising components and their communication patterns in order to prevent scenarios in which the interaction among components would introduce improper states. The work reported in [IM08] does not focus on behavioural properties; rather, it presents some results on performance. On the other hand, [MW97] proposes rules to guarantee the absence of deadlocks by construction. However, it presents rules for specific protocol patterns, such as resource sharing and client-server, using simple data communication. For instance, a component must always accept any input data value.

In our work, we use component metadata to store relevant properties that can be used to reduce the computational effort in verification. At each composition, we generate a new component (representing the whole) with metadata fulfilled with information of inner components and related to the kind of composition applied (see composition laws). In such a way, this approach is related to works that reuse previous verification in incremental verification and to those that use component metadata in component evolution. As far the authors are aware, these two concepts have not been used together in a seamless way as we did here; however, other researchers have accomplished important results in their respective fields.

Metadata has been used to represent additional concerns of a component for different purposes. In the wide range of existing works, we relate those with verification and validation purposes. They usually use metadata to link changes to necessary verification in the component evolution. [Bra02] use metadata to store

the causes of component changes and, furthermore, to identify relevant verification [Bra11] in its evolution. In [ODR+07] metadata is used to store additional specification information, and, through the analysis of this information, relevant test cases are selected. In both cases, metadata is used to reduce computation efforts in verification and validation.

Verification throughout a system evolution has a long relation with incremental verification, since changes are often local to restricted parts of a system and, for this reason, an incremental verification approach shows to be beneficial [Ghe12] by using effective resources and improving scalability. In this context, reuse of information [BW13] has been a key factor to reduce the computational effort, or to increase the quality of the verification result. [BW13] has identified three categories in which information from a previous verification should be used in order to save computational effort that would otherwise be necessary: (1) the use of partial results of verification that were not able to completely verify the system [BGL+11]; (2) the reuse of auxiliary information that was computed during previous verification in order to speed up later verifications [CSHL12]; (3) the use of witnesses for verifying the correctness of previous results [SSCS12]. We classify our approach in the second category above, since metadata is computed in previous verifications. Similarly to us, they have shown that scalability has been increased by reusing previous results in incremental verification.

Adherence to communication patterns, as a way to improve the efficiency of the verifications, is another distinguishing feature of our approach. We are not aware of any other constructive approach to design deadlock free systems that uses both metadata and patterns, and whose side conditions can be fully mechanically checked.

9.2. Future work

Despite the new contributions, our approach has some limitations. First, we require that component contracts have an associated behaviour, which is not always the case in component models in industry; this is, however, essential to support behavioural verifications. Next, the strategy with metadata indicates some compatible communications between components as incompatible (false-negatives). This is an intrinsic problem in local analysis methods, which is acceptable considering their advantages. In this case, the developer has to use traditional verification to complement our strategy. The strategy with metadata must be adopted as a technique that guides the attention to the crucial compositions, and not as a 'silver bullet' for the composition problem in general.

The use of patterns to avoid the only non-compositional side condition of reflexive composition has proved very promising. However, we have explored the use of a two patterns: the resource allocation and the async dynamic patterns. As future work we plan to integrate further patterns into $\mathcal{BRICK}$ and develop further case studies. This will help to further emphasise the practical scalability of our compositional verification approach.

We developed a prototype tool that automatically creates the script files for a given number of philosophers (for the first case study) or nodes (second case study), interacts with FDR and returns a performance analysis report. This allowed the execution of more experiments with different number of components. We are currently developing a complete framework for a component-based development based on our approach.

Acknowledgments

Philip Armstrong provided important information on FDR. The EU Framework 7 Integrated Project COMPASS (Grant Agreement 287829) financed most of the work presented here. INES and CNPq supports the work of Marcel Oliveira: grants 573964/2008-4, 560014/2010-4 and 483329/2012-6. We thank Ana Cavalcanti and Jim Woodcock for their suggestions on our work.

A. Further Formal Definitions

A.1. Buffers

We make a slightly change to generalise the notion of buffers, and ease the mapping of events in the two sides of the buffer. Instead of *left* and *right* channels, we assume they are distinct sets of events; more precisely, the domain and the range of a bijection called LR. For instance, the simplest example of a buffer process, the process $COPY$, could be written as follows:

$$COPY(LR) = ?x : \mathrm{dom}(LR) \to LR(x) \to COPY(LR)$$

Another example is a buffer process which, unlike $COPY$, does not insist upon outputting one thing before inputting the next. It represents an infinity buffer B^∞.

$$\begin{aligned} B^\infty(LR) = B^\infty_{\langle\rangle}(LR) &= ?x : \mathrm{dom}\, LR \to B^\infty_{\langle x \rangle} \\ B^\infty_{s^\frown\langle y\rangle}(LR) &= ?x : \mathrm{dom}\, LR \to B^\infty_{\langle x\rangle^\frown{}_s{}^\frown\langle y\rangle}(LR) \\ &\quad \Box\ LR(y) \to B^\infty_s(LR) \end{aligned}$$

We can also imagine a buffer of arbitrary $B^n(LR)$ size as the pipeline of several one-place buffers $COPY(LR)$.

$$B^n(LR) = COPY(LR) \gg_{LR} COPY(LR) \gg_{LR} \ldots \gg_{LR} COPY(LR)$$

The operator $\gg_{LR}$ is a new chain operator that instead of using *left* and *right*, takes LR into consideration:

$$P \gg_{LR} Q = (P \,[\![\, RM \,]\!]\, [\![mid]\!]\, Q \,[\![\, LM]\!]) \setminus mid$$

where LR, RM and LM are bijections and mid is a set of events, such that $(\alpha P \cup \alpha Q) \subseteq (\mathrm{dom}\, LR \cup \mathrm{ran}\, LR)$, $(\mathrm{dom}\, LR \cap \mathrm{ran}\, LR) = \emptyset$, $(\mathrm{dom}\, LR \cup \mathrm{ran}\, LR) \cap mid = \emptyset$, $\mathrm{ran}\, RM = mid$, $\mathrm{ran}\, ML = mid$, $\mathrm{dom}\, RM = \mathrm{ran}\, LR$, and $\mathrm{dom}\, LM = \mathrm{dom}\, LR$.

The new chain operation is as simple as the original one [Ros98]; it renames events of both processes to an intermediary set of events (mid) and synchronise them on this set. For the sake of brevity, we consider that mid is an arbitrary set of events unused by the processes. LR is used as a reference to all other arbitrary bijections, used for the renames.

Instead of the traditional definition of buffer with left and right, we specify a bidirectional buffer parametrised by an ordinary buffer and two bijections.

$$BUFF_{IO}(BF, LR_1, LR_2) = (BF(LR_1) \,|||\, BF(LR_2))$$

For the sake of brevity, we consider the following processes:

$$\begin{aligned} BUFF^1_{IO}(c, z) &= BUFF_{IO}(COPY, R^{c\to z}_{IO}, R^{z\to c}_{IO}) \\ BUFF^n_{IO}(c, z) &= BUFF_{IO}(B^n, R^{c\to z}_{IO}, R^{z\to c}_{IO}) \\ BUFF^\infty_{IO}(c, z) &= BUFF_{IO}(B^\infty_{\langle\rangle}, R^{c\to z}_{IO}, R^{z\to c}_{IO}) \end{aligned}$$

where:

$$COPY(LR) = ?x : \mathrm{dom}(LR) \to LR(x) \to COPY(LR)$$

B. Auxiliary Theorems

Theorem B.1 (Input determinism compositionality). Let P and Q be input deterministic processes, and C a set of channels, such that $\alpha P \cap \alpha Q \subseteq \{\!|\ C\ |\!\}$ and $P \,[\![\, \{\!|\ C\ |\!\} \,]\!]\, Q$ is deadlock-free. Then $P \,[\![\, \{\!|\ C\ |\!\} \,]\!]\, Q$ is input deterministic with respect to all inputs of P and Q, excepting those within C.

Proof. The proof of this theorem is performed by contradicting the statement that $P \,[\![\, \{\!|\ C\ |\!\} \,]\!]\, Q$ is input deterministic. From the definition of Input Determinism (Page 10), we have:

$$\begin{aligned} &\exists\, u : \mathrm{seq}\, \Sigma;\ a : \Sigma \mid\ u^\frown\langle a\rangle \in traces(P \,[\![\, \{\!|\ C\ |\!\} \,]\!]\, Q) \wedge\ \ a \in inputs_P(P \,[\![\, \{\!|\ C\ |\!\} \,]\!]\, Q) \wedge a \notin \{\!|\ C\ |\!\} \\ &\qquad\qquad \bullet\ (u, \{a\}) \in failures(P \,[\![\, \{\!|\ C\ |\!\} \,]\!]\, Q) \\ &\Rightarrow \qquad\qquad [traces \text{ of } P \,[\![\, \{\!|\ C\ |\!\} \,]\!]\, Q] \\ &\exists\, s : traces(P);\ t : traces(Q);\ a : (inputs_P(P) \cup inputs_P(Q)) \setminus \{\!|\ C\ |\!\} \end{aligned}$$

$\bullet\ (s \frown \langle a \rangle \in traces(P) \lor t \frown \langle a \rangle \in traces(Q)) \land (s \,[\![\{\!| C |\!\}]\!]\, t, \{a\}) \in failures(P \,[\![\{\!| C |\!\}]\!]\, Q)$
$\Rightarrow$ [$a \notin \alpha(P) \cap \alpha(Q)$, P and Q are input deterministic]
$\exists\, s : traces(P);\ t : traces(Q);\ a : (inputs_P(P) \cup inputs_P(Q)) \setminus \{\!| C |\!\}$
$\bullet\ ((s, \{a\}) \notin failures(P) \lor (t, \{a\}) \notin failures(Q)) \land (s \,[\![\{\!| C |\!\}]\!]\, t, \{a\}) \in failures(P \,[\![\{\!| C |\!\}]\!]\, Q)$
$\Rightarrow$ [Failures of $P \,[\![\{\!| C |\!\}]\!]\, Q$]
$\exists\, s, t : \mathrm{seq}\,\Sigma;\ Y, Z : \mathbb{P}\,\Sigma;\ a : (inputs_P(P) \cup inputs_P(Q)) \setminus \{\!| C |\!\}$
$\quad \mid (s, Y) \in failures(P) \land (t, Z) \in failures(Q)$
$\quad \bullet\ ((s, \{a\}) \notin failures(P) \lor (t, \{a\}) \notin failures(Q)) \land Y \setminus \{\!| C |\!\} = Z \setminus \{\!| C |\!\} \land \{a\} = Y \cup Z$
$\Rightarrow$ [$\{a\} = Y \cup Z \land (Y \setminus \{\!| C |\!\} = Z \setminus \{\!| C |\!\}) \land a \notin \{\!| C |\!\} \Rightarrow Y = \{\!| C |\!\} \land Z = \{\!| a |\!\}$]
$\exists\, s : traces(P), t : traces(Q), a : (inputs_P(P) \cup inputs_P(Q)) \setminus \{\!| C |\!\}$
$\quad \bullet\ ((s, \{a\}) \notin failures(P) \lor (t, \{a\}) \notin failures(Q)) \land (s, \{a\}) \in failures(P) \land (t, \{a\}) \in failures(Q)$
$\Rightarrow$ [Contradiction]
false

□

Theorem B.2 (Output decisiveness compositionality). Let P and Q be output decisive processes, and C a set of channels, such that $\alpha P \cap \alpha Q \subseteq \{\!| C |\!\}$ and $P \,[\![\{\!| C |\!\}]\!]\, Q$ is deadlock-free. Then $P \,[\![\{\!| C |\!\}]\!]\, Q$ is output decisive with respect to all outputs of P and Q, excepting those within C.

Proof. The proof of this theorem is performed by contradicting the statement that $P \,[\![\{\!| C |\!\}]\!]\, Q$ is output decisive. From the definition of output deciseveness (Page 10), the proof can be divided in two parts:

(i) $\forall\, s : \mathrm{seq}\,\Sigma;\ c.b : \Sigma \mid s \frown \langle c.b \rangle \in traces(P) \land c.b \in outputs(c, P) \bullet (s, outputs(c, P)) \notin failures(P)$
(ii) $\forall\, s : \mathrm{seq}\,\Sigma;\ c.b : \Sigma \mid s \frown \langle c.b \rangle \in traces(P) \land c.b \in outputs(c, P) \bullet (s, outputs(c, P)) \setminus \{c.b\} \notin failures(P)$

The proof of the first part can be performed similarly to the proof of Theorem B.1. Furthermore, we focus on the proof of part ii, which starts by contradicting this statement for the process $P \,[\![\{\!| C |\!\}]\!]\, Q$.

$\exists\, u : \mathrm{seq}\,\Sigma;\ c.b : \Sigma \mid u \frown \langle c.b \rangle \in traces(P \,[\![\{\!| C |\!\}]\!]\, Q) \mid\ c.b \in outputs(c, P \,[\![\{\!| C |\!\}]\!]\, Q) \land c \notin C \bullet$
$\qquad (u, outputs(c, P \,[\![\{\!| C |\!\}]\!]\, Q) \setminus \{c.b\}) \notin failures(P \,[\![\{\!| C |\!\}]\!]\, Q)$
$\Rightarrow$ [*traces* of $P \,[\![\{\!| C |\!\}]\!]\, Q$]
$\exists\, s : traces(P);\ t : traces(Q);\ c.b : (outputs(P) \cup outputs(Q)) \bullet$
$\quad c \notin C \land (s \frown \langle c.b \rangle \in traces(P) \lor t \frown \langle c.b \rangle \in traces(Q))$
$\quad \land (u, outputs(c, P \,[\![\{\!| C |\!\}]\!]\, Q) \setminus \{c.b\}) \notin failures(P \,[\![\{\!| C |\!\}]\!]\, Q)$
$\Rightarrow$ [*failures* of $P \,[\![\{\!| C |\!\}]\!]\, Q$]
$\exists\, s : traces(P);\ t : traces(Q);\ c.b : (outputs(P) \cup outputs(Q))$
$\quad \bullet\ c \notin C \land (s \frown \langle c.b \rangle \in traces(P) \lor t \frown \langle c.b \rangle \in traces(Q)) \land$
$\qquad \neg\ \exists\, s, t : \mathrm{seq}\,\Sigma;\ Y, Z : \mathbb{P}\,\Sigma$
$\qquad\quad \mid (s, Y) \in failures(P) \land (t, Z) \in failures(Q)$
$\qquad\quad \bullet\ Y \setminus \{\!| C |\!\} = Z \setminus \{\!| C |\!\} \land (outputs(c, P) \cup outputs(c, Q) \setminus \{c.b\}) = Y \cup Z$
$\Rightarrow$ [$c.b \notin \alpha P \cap \alpha Q$, P and Q are output decisive]
$\exists\, s : traces(P);\ t : traces(Q);\ c.b : (outputs(P) \cup outputs(Q))$

$$\bullet\ c \notin C \land \left(\begin{array}{l} (\{\!| C |\!\} \subset \alpha P \land (s, outputs(c, P) \setminus \{c.b\}) \in failures(P) \land \\ \neg\ \exists\, s, t : \mathrm{seq}\,\Sigma;\ Y, Z : \mathbb{P}\,\Sigma \mid (s, Y) \in failures(P) \land (t, Z) \in failures(Q) \\ \qquad \bullet\ Y \setminus \{\!| C |\!\} = Z \setminus \{\!| C |\!\} \land (outputs(c, P) \setminus \{c.b\}) = Y \cup Z) \\ \lor \\ (\{\!| C |\!\} \subset \alpha Q \land (t, outputs(c, Q) \setminus \{c.b\}) \in failures(Q) \land \\ \neg\ \exists\, s, t : \mathrm{seq}\,\Sigma;\ Y, Z : \mathbb{P}\,\Sigma \mid (s, Y) \in failures(P) \land (t, Z) \in failures(Q) \\ \qquad \bullet\ Y \setminus \{\!| C |\!\} = Z \setminus \{\!| C |\!\} \land (outputs(c, Q) \setminus \{c.b\}) = Y \cup Z) \end{array} \right)$$

$\Rightarrow$ [$(Y \cup Z) \subset \{\!| C |\!\} \land (Y \setminus \{\!| C |\!\} = Z \setminus \{\!| C |\!\}) \land c \notin C \Rightarrow Y = Z$]
$\exists\, s : traces(P);\ t : traces(Q);\ c.b : (outputs(P) \cup outputs(Q))$

$$\bullet\ c \notin C \wedge \left(\begin{array}{l} (\{\!|\ C\ |\!\} \subset \alpha P \wedge (s, outputs(c, P) \setminus \{c.b\}) \in failures(P) \\ \quad \wedge (s, outputs(c, P) \setminus \{c.b\}) \notin failures(P)) \\ \vee \\ (\{\!|\ C\ |\!\} \subset \alpha Q \wedge (t, outputs(c, Q) \setminus \{c.b\}) \in failures(Q) \\ \quad \wedge (t, outputs(c, Q) \setminus \{c.b\}) \notin failures(Q)) \end{array} \right)$$

$\Rightarrow$ [Contradiction]

false

□

Theorem B.3 (Conjugate protocols and deadlock freedom). Let P and Q be two deadlock-free conjugate protocols. Then $P \parallel Q$ is deadlock-free if, and only if, they are compatible.

Proof. This theorem is a direct consequence of Lemma B.9 and the *Deadlock Rule 1*, presented in [Ros98], which says that any tree topology system free of strong conflict is deadlock-free; strong conflicts are states in which two components have no choice of communicating with rest of the system, and they cannot establish a communication between them.

The system structure $P \parallel Q$ has a tree form; no cycle is presented. Moreover, according to the Lemma B.9, compatible protocols are free of strong conflicts. As a result, according to *Deadlock Rule 1*, the pair of processes is deadlock-free.

□

Theorem B.4 (Protocol buffering). Let P and Q be two deadlock-free I/O confluent communication protocols with distinct alphabets, BF a deterministic buffer, and LR_1 and LR_2 two bijections, such that:

(i) P and Q satisfy the finite output property;
(ii) $LR_1 : outputs(P) \leftrightarrow inputs(Q)$ and $LR_2 : outputs(Q) \leftrightarrow inputs(P)$;
(iii) $P \,[\![LR_1]\!]$ and $Q \,[\![LR_2]\!]$ are strong compatible.

Then the synchronization of P and Q via a buffer BF .

$$P \,[\![\alpha P]\!]\, BUFF_{IO}(BF, LR_1, LR_2) \,[\![\alpha Q]\!]\, Q$$

is deadlock-free

Proof. The proof of this theorem is based on the theory of *confluent* processes, and their ability to be *buffer tolerant* [Ros05]. Besides confluence, another weaker notion, called *channel confluence* has this property. Channel confluent processes are processes whose pattern of communication is independent of which data is sent, and abstracting such data this pattern obeys the confluence property. Buffer tolerant systems have the property of preserving deadlock freedom after buffers are introduced into its internal channels.

Similarly to channel confluent processes, I/O confluent processes are buffer tolerant. The reason is that any communication protocol P has an equivalent protocol P' with two channels, one for communicating inputs and the other for outputs; this is obtained from a simple bijective renaming. If P is I/O confluent, then P' is channel confluent. Similarly, the process Q has an analogue protocol Q' that is channel confluent. A synchronisation of P' and Q' is buffer tolerant, and having an equivalent behaviour, so is the synchronisation of P and Q.

Given these considerations, we start our proof by the following statement, based on the Theorem B.3. Consider, for a given process S, $IN_S = inputs_P(S)$ and $OUT_S = outputs_P(S)$.

$P \,[\![LR_1]\!] \parallel Q \,[\![LR_2]\!]$ is deadlock-free
$\Rightarrow$ [P and Q are equivalent to channel confluent processes]
$(P \,[\![OUT_P]\!]\, BF(LR_1)) \,[\![IN_P \cup IN_Q]\!]\, (Q \,[\![OUT_Q]\!]\, BF(LR_2))$ is deadlock-free
$\Rightarrow$ [$\alpha(BF(LR_1)) \cap \alpha(BF(LR_2)) = \emptyset$]
$P \,[\![IN_P \cup OUT_P]\!]\, (BF(LR_1)) \,|||\, BF(LR_2)) \,[\![IN_Q \cup OUT_Q]\!]\, Q$ is deadlock-free
$\Rightarrow$ [according to definition of $BUFF_I O$, P and Q are I/O processes]
$P \,[\![\alpha P]\!]\, BUFF_{IO}(BF, LR_1, LR_2) \,[\![\alpha Q]\!]\, Q$ is deadlock-free

□

Theorem B.5 (Buffering self-injection compatibility and deadlock-freedom). Let P be a deadlock-free I/O confluent process, c and z communication channels, BF a deterministic and infinite buffer, and LR_1 and LR_2 bijections, such that:

(i) $LR_1 : outputs(P, c) \leftrightarrow inputs(P, z)$ and $LR_2 : outputs(P, z) \leftrightarrow inputs(P, c)$
(ii) $Prot_{IMP}(P, c) \,[\![\, LR_1]\!]$ and $Prot_{IMP}(Q, z) \,[\![\, LR_2]\!]$ are strong compatible
(iii) $P \upharpoonright \{\!|\, c, z \,|\!\}$ is *buffering self-injection compatible* and satisfy the finite output property

Then, $P \upharpoonright \{\!|\, c, z \,|\!\}$ is *buffering self-injection compatible* if, and only if, the following process is deadlock-free:

$$P \upharpoonright \{\!|\, c, z \,|\!\} \,[\![\, \{\!|\, c, z \,|\!\}]\!] \, BUFF_{IO}(BF, LR_1, LR_2)$$

Proof. The proof of this lemma is carried by case analysis. We contradict the theorem above, showing that for each possible case where the process deadlocks there is a possible communication to perform.

The process deadlocks when P and the two buffers within $BUFF^1_{IO}$ are stuck. Therefore, in this proof, we analyse each case where $BUFF^1_{IO}$ may deadlock. These are: when the buffers are empty, and no output of P comes out; and when at least one buffer is full.

1) In the first case, every time both buffers are empty, the process P has communicated the same number of events through c and z. In this case, the last event communicated by P cannot be an output; if so, the buffer would not be empty. Therefore, since P is buffering self-inject compatible, P may output something in this case. These processes do not deadlocks.

Below, we present the formal proof for this case, showing that if it deadlocks, then we have a contradiction. Assume that $O_c = outputs(c, P)$, $O_z = outputs(z, P)$, $I_c = inputs(c, P)$, $I_z = inputs(z, P)$, $P_J = P \upharpoonright \{\!|\, c, z \,|\!\}$ and $B_F = BUFF^1_{IO}(LR_1, LR_2)$.

$$
\begin{array}{ll}
& P \upharpoonright \{\!|\, c, z \,|\!\} \,[\![\, \{\!|\, c, z \,|\!\}]\!] \, B_F \text{ deadlocks} \\
\Leftrightarrow & \qquad\qquad [\text{Lemma B.10}] \\
& \exists\, s : \operatorname{seq} \Sigma;\ X : \mathbb{P}\, \Sigma \mid (s, X) : failures(P_J) \bullet (s, \Sigma \setminus X) \in failures(B_F) \\
\Leftrightarrow & \qquad\qquad [\text{buffers are empty}] \\
& \exists\, s : \operatorname{seq} \Sigma;\ X : \mathbb{P}\, \Sigma \mid (s, X) : failures(P_J) \bullet (s \downarrow O_c = s \downarrow I_z) \wedge (s \downarrow O_z = s \downarrow I_c) \\
& \qquad\qquad \wedge\, (s, \Sigma \setminus X) \in failures(B_F) \wedge (\Sigma \setminus X) \cap (O_c \cup O_z) = \emptyset \\
\Leftrightarrow & \qquad\qquad [P \text{ is buffering self-injection compatible, } X \cap outputs_P(P_J) = \emptyset] \\
& \exists\, s : \operatorname{seq} \Sigma;\ X : \mathbb{P}\, \Sigma \mid (s, X) : failures(P_J) \bullet (s \downarrow O_c = s \downarrow I_z) \wedge (s \downarrow O_z = s \downarrow I_c) \\
& \qquad\qquad \wedge\, ((\Sigma \setminus X) \cap (O_c \cup O_z) = \emptyset) \wedge (X \cap (O_c \cup O_z) = \emptyset) \\
\Leftrightarrow & \qquad\qquad [\text{Contradiction}] \\
& \textbf{false}
\end{array}
$$

2) In the second case, at least one buffer is full. Below, we present the formal proof for this case, showing that if it deadlocks, then we have a contradiction. Here, we also assume that $O_c = outputs(c, P)$, $O_z = outputs(z, P)$, $I_c = inputs(c, P)$, $I_z = inputs(z, P)$, $P_J = P \upharpoonright \{\!|\, c, z \,|\!\}$, $B_F = BUFF^1_{IO}(LR_1, LR_2)$, $P_c = Prot_{IMP}(P, c)$ and $P_z = Prot_{IMP}(P, z)$. We start the proof by a consequence of the compatibility of the protocol implementation, and Lemma B.10.

$$
\begin{array}{ll}
& \forall\, s : \operatorname{seq} \Sigma;\ X : \mathbb{P}\, \Sigma \mid (s, X) : failures(B_F) \bullet (s, \Sigma \setminus X) \notin failures(P_c \,|||\, P_z) \\
\Leftrightarrow & \qquad\qquad [\text{Lemma B.10, } Proj \parallel B_F \text{ deadlocks}] \\
& \exists\, s : \operatorname{seq} \Sigma;\ X : \mathbb{P}\, \Sigma \mid (s, X) : failures(B_F) \bullet (s, \Sigma \setminus X) \notin failures(P_c \,|||\, P_z) \\
& \qquad\qquad \wedge\, (s, \Sigma \setminus X) \in failures(Proj) \\
\Leftrightarrow & \qquad\qquad [\text{The failures semantics of the interleave operator}] \\
& \exists\, s : \operatorname{seq} \Sigma;\ X : \mathbb{P}\, \Sigma \mid (s, X) : failures(B_F) \bullet (s \upharpoonright c, \Sigma \setminus X) \notin failures(P_c) \wedge (s \upharpoonright z, \Sigma \setminus X) \notin failures(P_z) \\
& \qquad\qquad \wedge\, (s, \Sigma \setminus X) \in failures(Proj) \\
\Leftrightarrow & \qquad\qquad [\text{at least one buffer is full, } P \text{ is buffering self-injection compatible}] \\
& \exists\, s : \operatorname{seq} \Sigma;\ X : \mathbb{P}\, \Sigma \mid (s, X) : failures(B_F) \bullet (s \upharpoonright c, \Sigma \setminus X) \notin failures(P_c) \wedge (s \upharpoonright z, \Sigma \setminus X) \notin failures(P_z) \\
& \qquad\qquad \wedge\, ((s \upharpoonright z, (\Sigma \setminus X) \cup \{\!|\, C \,|\!\}) \in failures(Proj \upharpoonright z) \\
& \qquad\qquad\quad \vee\, (s \upharpoonright c, (\Sigma \setminus X) \cup \{\!|\, z \,|\!\}) \in failures(Proj \upharpoonright c)) \\
\Leftrightarrow & \qquad [\text{Definition 3.5, } failures(Proj \upharpoonright c) = failures(P_c) \wedge failures(Proj \upharpoonright z) = failures(P_z)]
\end{array}
$$

$$\exists\, s : \operatorname{seq}\Sigma;\ X : \mathbb{P}\,\Sigma \mid (s, X) : \mathit{failures}(B_F) \bullet (s \upharpoonright c, \Sigma \setminus X) \notin \mathit{failures}(P_c) \land (s \upharpoonright z, \Sigma \setminus X) \notin \mathit{failures}(P_z) \land ((s \upharpoonright z, \Sigma \setminus X) \in \mathit{failures}(P_z) \lor (s \upharpoonright c, \Sigma \setminus X) \in \mathit{failures}(P_c))$$

$\Leftrightarrow$ [Contradiction]

false

□

Theorem B.6 (Unary Composition Monotonicity). Let *Ctr* be a component contract, and $\langle c_1, .., c_n\rangle$ and $\langle z_1, .., z_n\rangle$ sequences of distinct channels within $\mathcal{C}_{Ctr}$, such that the behaviour ($\mathcal{B}$) in $Ctr \nprec^{\langle c_1,..,c_n\rangle}_{\langle z_1,..,z_n\rangle}$ is deadlock-free. Then $P \nprec^{\langle c_1,..,c_n\rangle}_{\langle z_1,..,z_n\rangle}$ is a component contract.

Proof. In order to proof that the resulting tuple is a component contract, we have to show that its structure is compatible with Definition 3.1. This is provided by the use of the unary composition, $P[ic \hookrightarrow oc] = P \asymp|^{\langle ic\rangle}_{\langle oc\rangle}$ (see Definition 3.9). Moreover, the behaviour of the component is also an I/O process, since the composition does not introduce divergences (no hiding operation or undesired renaming is performed), the infinite behaviours of the original components result in a new infinite process, and the resulting process is input deterministic and output decisive with respect to the channels that remain in contract, $\mathcal{C}$ (see Theorems B.1 and B.2). □

Theorem B.7 (Decoupled Channels and Self-Injection Compatibility). Let P be an deadlock-free I/O process, c and z communication channels, and LR_1 and LR_2 bijections, such that:

(i) $LR_1 : \mathit{outputs}(P, c) \leftrightarrow \mathit{inputs}(P, z)$ and $LR_2 : \mathit{outputs}(P, z) \leftrightarrow \mathit{inputs}(P, c)$
(ii) $\mathit{Prot}_{IMP}(P, c) \,[\![\, LR_1]\!]$ and $\mathit{Prot}_{IMP}(Q, z) \,[\![\, LR_2]\!]$ are compatible
(iii) $\{\!|\, c, z \,|\!\}$ *DecoupledIn* P

Then, $P \upharpoonright \{\!|\, c, z \,|\!\}$ is *buffering self-injection compatible.*

Proof. The proof of this theorem is underpinned by the notion of protocol compatibility and by Lemma 5.3. We start by the fact that the synchronisation of compatible protocols is deadlock-free (Theorem B.4).

$(\mathit{Prot}_{IMP}(P, c) \,|||\, \mathit{Prot}_{IMP}(P, z)) \,[\![\, \{\!|\, c, z \,|\!\}\,]\!]\, \mathit{BUFF}^1_{IO}(LR_1, LR_2)$ is deadlock-free
$\Rightarrow$ [Theorem B.3, $\mathit{Prot}_{IMP}(P, z) \sqsubseteq_{\mathrm{F}} Pz'$, $\mathit{Prot}_{IMP}(P, c) \sqsubseteq_{\mathrm{F}} Pc'$]
$(Pc' \,|||\, Pz') \,[\![\, \{\!|\, c, z \,|\!\}\,]\!]\, \mathit{BUFF}^1_{IO}(LR_1, LR_2)$ is deadlock-free
$\Rightarrow$ [$\{c, z\}$ *DecoupledIn* P, $P \upharpoonright \{c, z\} \equiv_{\mathrm{F}} Pz' \,|||\, Pc'$]
$P \upharpoonright \{\!|\, c, z \,|\!\} \,[\![\, \{\!|\, c, z \,|\!\}\,]\!]\, \mathit{BUFF}^1_{IO}(LR_1, LR_2)$ is deadlock-free
$\Rightarrow$ [Theorem 5.3]
$P \upharpoonright \{\!|\, c, z \,|\!\}$ is self-injection compatible

□

The following theorem is based on the following definition of strong conflict.

Definition B.1 (Strong conflict). Let P and Q be conjugate protocols. Then a strong conflict of $Q \parallel P$ is a state $(s, \langle X_P, X_Q\rangle)$ in which $\Sigma \setminus X_P \subseteq X_Q \land \Sigma \setminus X_Q \subseteq X_P$

Theorem B.8. Let P be a communication protocol. Then

$$\mathit{failures}(P) = \{(s, X) \mid s \in \mathit{traces}(P) \land X \subseteq \Sigma \setminus I^s_P \land O^s_P = \emptyset \lor \neg\, O^s_P \subseteq X\}$$

Proof. Based on Definition 3.4, there are, at most, two channels in a communication protocol c_1 and c_2: one for inputting, and another for outputting. Based on the *inputdeterminism* and the *outputdecisiveness* properties, $(s, X) \in \mathit{failures}(P)$ if, and only if, $s \in \mathit{traces}(P)$ and:

$$\forall\, s : \operatorname{seq}\Sigma;\ c_1.a : \Sigma \mid s \frown \langle c_1.a\rangle \in \mathit{traces}(P) \land c_1.a \in \mathit{inputs}(c_1, P) \bullet \neg\, \{c_1.a\} \subseteq X$$
$$\land\, \forall\, s : \operatorname{seq}\Sigma;\ c_2.b : \Sigma \mid s \frown \langle c_2.b\rangle \in \mathit{traces}(P) \land c_2.b \in \mathit{outputs}(c_2, P) \bullet \neg\, \mathit{outputs}(c_2, P) \subseteq X \land \mathit{outputs}(c_2, P) \setminus \{c_2.b\} \subseteq X$$

$\Leftrightarrow$ [based on the definitions of I_P^s and O_P^s, and considering $outputs(c_2, P) = outputs_P(P)$]
$\forall\, a : I_P^s \bullet (s, \{a\}) \notin X \wedge \forall\, b : O_P^s \bullet \neg\, outputs_P(P) \subseteq X \wedge (outputs_P(P) \setminus \{b\}) \subseteq X$
$\Leftrightarrow$ [no event within I_P^s belongs to X]
$X \subseteq \Sigma \setminus I_P^s \wedge \forall\, b : O_P^s \bullet \neg\, outputs_P(P) \subseteq X \wedge (s, outputs_P(P) \setminus \{b\}) \subseteq X$
$\Leftrightarrow$ [if O_P^s is nonempty, then X does not contains all events within O_P^s]
$X \subseteq \Sigma \setminus I_P^s \wedge (O_P^s \neq \emptyset \Rightarrow \neg\, O_P^s \subseteq X)$
$\Leftrightarrow$ [rewriting]
$X \subseteq \Sigma \setminus I_P^s \wedge (O_P^s = \emptyset \vee \neg\, O_P^s \subseteq X)$

□

Theorem B.9. Let P and Q be two conjugate protocols. Then $P \parallel Q$ is a pair of processes free of strong conflicts if, and only if, P and Q are compatible.

Proof. Based on the Definition B.1, the absence of strong conflicts can defined as:

$$\forall\, s : \operatorname{seq}\Sigma;\ X_P : \mathbb{P}\,\Sigma \mid (s, X_P) \in failures(P) \wedge X_P \supseteq \Sigma \setminus (I_P^s \cup O_P^s) \wedge X_Q \supseteq \Sigma \setminus (I_Q^s \cup O_Q^s) \bullet \neg\, (\Sigma \setminus X_P \subseteq X_Q) \vee \neg\, (\Sigma \setminus X_Q \subseteq X_P)$$

So, we have to prove that if two protocols are compatible than the statement above is satisfied, and vice-versa. The first part of our proof focus on that protocol compatibility implies in the absence of strong conflicts. We start by the central statement of protocol compatibility (Page 16), saying that the following is satisfied for all $s \in traces(P) \cap traces(Q)$. For the sake of brevity, consider, for a given process S, $IN_S = inputs_P(S)$ and $OUT_S = outputs_P(S)$.

$\forall\, s : traces(P) \cap traces(Q) \bullet (O_P^s \neq \emptyset \wedge O_P^s \subseteq I_Q^s) \vee (O_Q^s \neq \emptyset \wedge O_Q^s \subseteq I_P^S)$
$\Rightarrow$ [Theorem B.8]
$\forall\, s : \operatorname{seq}\Sigma;\ X_P, X_Q : \mathbb{P}\,\Sigma$
$\quad \mid (s, X_P) \in failures(P) \wedge (s, X_Q) \in failures(Q)$
$\quad\ \wedge X_P \supseteq \Sigma \setminus (I_P^S \cup O_P^s) \wedge X_Q \supseteq \Sigma \setminus (I_Q^s \cup O_Q^s)$
$\quad \bullet (O_P^s \neq \emptyset \wedge O_P^s \subseteq I_Q^s \wedge I_Q^s \subseteq \Sigma \setminus X_Q \wedge \neg\, O_P^s \subseteq X_P) \vee$
$\quad\ (O_Q^s \neq \emptyset \wedge O_Q^s \subseteq I_P^S \wedge I_P^S \subseteq \Sigma \setminus X_P \wedge \neg\, O_Q^s \subseteq X_Q)$
$\Rightarrow$ [as a direct implication of the statement above]
$\forall\, s : \operatorname{seq}\Sigma;\ X_P, X_Q : \mathbb{P}\,\Sigma$
$\quad \mid (s, X_P) \in failures(P) \wedge (s, X_Q) \in failures(Q)$
$\quad\ \wedge X_P \supseteq \Sigma \setminus (I_P^S \cup O_P^s) \wedge X_Q \supseteq \Sigma \setminus (I_Q^s \cup O_Q^s)$
$\quad \bullet (O_P^s \neq \emptyset \wedge O_P^s \subseteq I_Q^s \wedge I_Q^s \subseteq \Sigma \setminus X_Q \wedge \neg\, (O_P^s \subseteq X_P) \wedge \neg\, (\Sigma \setminus X_Q \subseteq X_P))$
$\quad\ \vee (O_Q^s \neq \emptyset \wedge O_Q^s \subseteq I_P^S \wedge I_P^S \subseteq \Sigma \setminus X_P \wedge \neg\, (O_Q^s \subseteq X_Q) \wedge \neg\, (\Sigma \setminus X_P \subseteq X_Q))$
$\Rightarrow$ [simplifying]
$\forall\, s : \operatorname{seq}\Sigma;\ X_P, X_Q : \mathbb{P}\,\Sigma$
$\quad \mid (s, X_P) \in failures(P) \wedge (s, X_Q) \in failures(Q)$
$\quad\ \wedge X_P \supseteq \Sigma \setminus (I_P^S \cup O_P^s) \wedge X_Q \supseteq \Sigma \setminus (I_Q^s \cup O_Q^s)$
$\quad \bullet \neg\, (\Sigma \setminus X_Q \subseteq X_P) \vee \neg\, (\Sigma \setminus X_P \subseteq X_Q)$

The other part of this proof is concerned with proving that the absence of strong conflicts implies that P and Q are compatible.

$\forall\, s : \operatorname{seq}\Sigma;\ X_P, X_Q : \mathbb{P}\,\Sigma$
$\quad \mid (s, X_P) \in failures(P) \wedge (s, X_Q) \in failures(Q)$
$\quad\ \wedge X_P \supseteq \Sigma \setminus (I_P^S \cup O_P^s) \wedge X_Q \supseteq \Sigma \setminus (I_Q^s \cup O_Q^s)$
$\quad \bullet \neg\, (\Sigma \setminus X_Q \subseteq X_P) \vee \neg\, (\Sigma \setminus X_P \subseteq X_Q)$
$\Rightarrow$ [rewriting]
$\forall\, s : \operatorname{seq}\Sigma;\ X_P, X_Q : \mathbb{P}\,\Sigma$

$$
\begin{array}{l}
\quad \mid (s, X_P) \in \mathit{failures}(P) \land (s, X_Q) \in \mathit{failures}(Q) \\
\quad\ \land X_P \supseteq \Sigma \setminus (I_P^S \cup O_P^s) \land X_Q \supseteq \Sigma \setminus (I_Q^s \cup O_Q^s) \\
\quad \bullet (O_P^s \neq \emptyset \land \neg\, \Sigma \setminus X_Q \subseteq X_P) \lor (O_P^s = \emptyset \land \neg\, \Sigma \setminus X_Q \subseteq X_P) \\
\quad\ \lor \\
\quad\ (O_Q^s \neq \emptyset \land \neg\, \Sigma \setminus X_P \subseteq X_Q) \lor (O_Q^s = \emptyset \land \neg\, \Sigma \setminus X_P \subseteq X_Q) \\
\Rightarrow \qquad [\text{Based on the Theorem B.8 we imply: } \exists\, a : O_P^s \bullet I_P^S \cup \{a\} \subseteq \Sigma \setminus X_P.] \\
\quad [\text{By induction, we have that } \neg\, \Sigma \setminus X_P \subseteq X_Q \land O_P^s \neq \emptyset \Rightarrow \neg\, O_P^s \subseteq X_Q; \text{ the same is valid for } O_Q^s] \\
\forall\, s : \mathrm{seq}\, \Sigma;\ X_P, X_Q : \mathbb{P}\, \Sigma \\
\quad \mid (s, X_P) \in \mathit{failures}(P) \land (s, X_Q) \in \mathit{failures}(Q) \\
\quad\ \land X_P \supseteq \Sigma \setminus (I_P^S \cup O_P^s) \land X_Q \supseteq \Sigma \setminus (I_Q^s \cup O_Q^s) \\
\quad \bullet (O_Q^s \neq \emptyset \land \neg\, \Sigma \setminus X_Q \subseteq X_P \land \neg\, O_Q^s \subseteq X_P \land \neg\, O_Q^s \subseteq X_Q) \lor \\
\quad\ (O_P^s \neq \emptyset \land \neg\, \Sigma \setminus X_P \subseteq X_Q \land \neg\, O_P^s \subseteq X_Q \land \neg\, O_P^s \subseteq X_P) \lor \\
\quad\ (O_Q^s = \emptyset \land \neg\, \Sigma \setminus X_Q \subseteq X_P \land I_Q^s = \Sigma \setminus X_Q) \lor \\
\quad\ (O_P^s = \emptyset \land \neg\, \Sigma \setminus X_P \subseteq X_Q \land I_P^S = \Sigma \setminus X_P) \\
\Rightarrow \qquad [P \text{ and } Q \text{ are conjugate, } OUT_P \subseteq IN_Q \land OUT_Q \subseteq IN_P] \\
\quad [\text{By induction, we have that } \neg\, \Sigma \setminus X_P \subseteq X_Q \land O_P^s \neq \emptyset \Rightarrow \neg\, O_P^s \subseteq X_Q; \text{ the same is valid for } O_Q^s] \\
\forall\, s : \mathrm{seq}\, \Sigma;\ X_P, X_Q : \mathbb{P}\, \Sigma \\
\quad \mid (s, X_P) \in \mathit{failures}(P) \land (s, X_Q) \in \mathit{failures}(Q) \\
\quad\ \land X_P \supseteq \Sigma \setminus (I_P^S \cup O_P^s) \land X_Q \supseteq \Sigma \setminus (I_Q^s \cup O_Q^s) \\
\quad \bullet (O_Q^s \neq \emptyset \land \neg\, \Sigma \setminus X_Q \subseteq X_P \land \neg\, O_Q^s \subseteq X_P \land \neg\, O_Q^s \subseteq X_Q \land O_Q^s \subseteq I_P^S) \lor \\
\quad\ (O_P^s \neq \emptyset \land \neg\, \Sigma \setminus X_P \subseteq X_Q \land \neg\, O_P^s \subseteq X_Q \land \neg\, O_P^s \subseteq X_P \land O_P^s \subseteq I_Q^s) \lor \\
\quad\ ((O_Q^s = \emptyset \land \neg\, \Sigma \setminus X_Q \subseteq X_P \land I_Q^s = \Sigma \setminus X_Q \land O_P^s \neq \emptyset \land \neg\, O_P^s \subseteq X_P) \lor \\
\quad\ (O_P^s = \emptyset \land \neg\, \Sigma \setminus X_P \subseteq X_Q \land I_P^S = \Sigma \setminus X_P \land O_Q^s \neq \emptyset \land \neg\, O_Q^s \subseteq X_Q) \\
\Rightarrow \quad [\text{Based on the Theorem B.8, } X_P \text{ is a maximal refusal, we imply: } \exists\, b : O_P^s \bullet X_P = \Sigma \setminus (I_P^S \cup \{a\});\ ] \\
\quad [\text{By induction, considering that } IN_P \cap IN_Q = \emptyset,\ \neg\, I_Q^s \subseteq X_P \Rightarrow O_P^s \subseteq I_Q^s; \text{ the same is valid for } O_Q^s.] \\
\forall\, s : \mathrm{seq}\, \Sigma;\ X_P, X_Q : \mathbb{P}\, \Sigma \\
\quad \mid (s, X_P) \in \mathit{failures}(P) \land (s, X_Q) \in \mathit{failures}(Q) \\
\quad\ \land X_P \supseteq \Sigma \setminus (I_P^S \cup O_P^s) \land X_Q \supseteq \Sigma \setminus (I_Q^s \cup O_Q^s) \\
\quad \bullet (O_Q^s \neq \emptyset \land \neg\, \Sigma \setminus X_Q \subseteq X_P \land \neg\, O_Q^s \subseteq X_P \land \neg\, O_Q^s \subseteq X_Q \land IN_P \setminus I_P^S \subseteq X_P \land O_Q^s \subseteq I_P^S) \lor \\
\quad\ (O_P^s \neq \emptyset \land \neg\, \Sigma \setminus X_P \subseteq X_Q \land \neg\, O_P^s \subseteq X_Q \land \neg\, O_P^s \subseteq X_P \land IN_Q \setminus I_Q^s \subseteq X_Q \land O_P^s \subseteq I_Q^s) \lor \\
\quad\ ((O_Q^s = \emptyset \land \neg\, \Sigma \setminus X_Q \subseteq X_P \land I_Q^s = \Sigma \setminus X_Q \land O_P^s \neq \emptyset \land O_P^s \subseteq I_Q^s) \lor \\
\quad\ (O_P^s = \emptyset \land \neg\, \Sigma \setminus X_P \subseteq X_Q \land I_P^S = \Sigma \setminus X_P \land O_Q^s \neq \emptyset \land O_Q^s \subseteq I_P^S) \\
\Rightarrow \qquad [\text{simplifying, considering that } I_P^S,\ I_Q^s,\ O_P^s \text{ and } O_Q^s \text{ depends solely of the traces of } P \text{ and } Q] \\
\forall\, s : \mathit{traces}(P) \cap \mathit{traces}(Q) \bullet (O_P^s \neq \emptyset \land O_P^s \subseteq I_Q^s) \lor (O_Q^s \neq \emptyset \land O_Q^s \subseteq I_P^S)
\end{array}
$$

□

Theorem B.10. Let P and Q be divergence-free CSP processes. Then $P \parallel Q$ deadlocks if, and only if: $\exists (t, X) : \mathit{failures}(P) \bullet (t, \Sigma \setminus X) \in \mathit{failures}(Q)$

Proof. The proof of this lemma is mainly based on the semantics of the synchronised parallel operator.

$$
\begin{array}{l}
P \parallel Q \text{ is deadlock-free} \\
\Leftrightarrow \forall\, s : \Sigma^* \bullet (s, \Sigma^{\checkmark}) \notin \mathit{failures}(P \parallel Q) \\
\Leftrightarrow \qquad [\text{Semantics of the synchronised parallel operator}] \\
\forall\, s : \Sigma^* \bullet (s, \Sigma^{\checkmark}) \notin \{(t, X \cup Y) \mid (t, X) \in \mathit{failures}(P) \land (t, Y) \in \mathit{failures}(Q)\} \\
\Leftrightarrow \qquad [\text{set theory, } X \cup Y = \Sigma^{\checkmark} \Rightarrow \Sigma^{\checkmark} \setminus X \subseteq Y] \\
\forall\, s : \mathrm{seq}\, \Sigma;\ X : \mathbb{P}\, \Sigma \mid (s, X) \in \mathit{failures}(P) \bullet (s, \Sigma^{\checkmark} \setminus X) \notin \mathit{failures}(Q)
\end{array}
$$

□

C. Proofs of composition rules with metadata

This appendix provides details about the proofs of the theorems stated in Section 4.

Theorem 4.1 (Enriched Interleaving Composition Compatibility) An enriched interleaving composition is an enriched component contract.

Proof. Based on the definitions 4.1 and 4.3, in order to an enriched interleave composition $P\,[\!|||\!]^e\,Q$ being an enriched component contract, we have to prove that:

(i) $S = \langle \mathcal{B}_P, \mathcal{R}_P, \mathcal{I}_P, \mathcal{C}_P \rangle_{\langle\rangle} \asymp {}_{\langle\rangle}\langle \mathcal{B}_Q, \mathcal{R}_Q, \mathcal{I}_Q, \mathcal{C}_Q \rangle$ is a protocol oriented component.

(ii) $\mathrm{dom}\, Prot^{\mathcal{K}}_{PQ} \subseteq \mathcal{C}_S \wedge \forall\, c : \mathrm{dom}\, Prot^{\mathcal{K}}_{PQ} \bullet Prot^{\mathcal{K}}(c) \sqsubseteq_{\mathrm{F}} Prot_{IMP}(S, c)$

(iii) $\mathrm{dom}\, Dec^{\mathcal{K}} \subseteq \mathcal{C}_S \wedge \mathrm{ran}\, Dec^{\mathcal{K}} \subseteq \mathcal{C}_S \wedge$
$(\forall\, c_1, c_2 : \mathcal{C}_S \bullet c_1\, Dec^{\mathcal{K}}\, c_2 \Rightarrow \{c_1, c_2\}\, DecoupledIn\, S \wedge c_2\, Dec^{\mathcal{K}}_{PQ}\, c_1)$

where

Definition C.1 (Protocol oriented component). We say that a component S is protocol oriented if, and only if, for any process R, channel c, and set of events Z, such that $c \in \mathcal{C}_S$, $\alpha R \cap \{\!|\ c\ |\!\} = \emptyset$, $\forall\, z' : Z \bullet z' \in \mathcal{C}_S \setminus \{\!|\ c\ |\!\}$, and the protocols $Prot_{IMP}(R, z')$ and $Prot_{IMP}(\mathcal{B}_S, z')$ are strong compatible, and $\mathcal{B}_S \,[\![\, \{\!|\ Z\ |\!\} \,]\!]\, R$ is deadlock-free, the following holds:

$$\mathcal{B}_S \upharpoonright \{\!|\ c\ |\!\} \sqsubseteq_{\mathrm{F}} (\mathcal{B}_S \,[\![\, \{\!|\ Z\ |\!\} \,]\!]\, R) \upharpoonright \{\!|\ c\ |\!\}$$

Considering the items above, we split the proof in three parts; one corresponding to each item. We do not show the proofs about preservation of context process and dual protocols. These are similar to the proof for item ii.

Part 1 Prove that the composition is a protocol oriented component is the same as proving that for any process R, channel c, and set of events Z, such that $c \in \mathcal{C}_S$, $Z \subseteq \alpha\mathcal{B}_S \cup \alpha R$, $\mathcal{B}_S \,[\![\, Z \,]\!]\, R$ is deadlock-free, the following holds:

$$\begin{array}{ll}
\mathcal{B}_S \upharpoonright \{\!|\ c\ |\!\} \sqsubseteq_{\mathrm{F}} (\mathcal{B}_S \,[\![\, \{\!|\ Z\ |\!\} \,]\!]\, R) \upharpoonright \{\!|\ c\ |\!\} & [\text{Definition 3.10, } \mathcal{C}_S = \mathcal{C}_P \cup \mathcal{C}_Q,\ \mathcal{B}_S = \mathcal{B}_P \,|||\, \mathcal{B}_Q] \\
\Rightarrow (c \in \mathcal{C}_P \wedge ((\mathcal{B}_P \,|||\, \mathcal{B}_Q) \upharpoonright \{\!|\ c\ |\!\} \sqsubseteq_{\mathrm{F}} ((\mathcal{B}_P \,|||\, \mathcal{B}_Q) \,[\![\, \{\!|\ Z\ |\!\} \,]\!]\, R) \upharpoonright \{\!|\ c\ |\!\}) & [\text{rewriting, } \alpha\mathcal{B}_P \cap \alpha\mathcal{B}_Q = \emptyset] \\
\quad \vee (c \in \mathcal{C}_Q \wedge ((\mathcal{B}_P \,|||\, \mathcal{B}_Q) \upharpoonright \{\!|\ c\ |\!\} \sqsubseteq_{\mathrm{F}} ((\mathcal{B}_P \,|||\, \mathcal{B}_Q) \,[\![\, \{\!|\ Z\ |\!\} \,]\!]\, R) \upharpoonright \{\!|\ c\ |\!\}) & \\
\Rightarrow (c \in \mathcal{C}_P \wedge (\mathcal{B}_P \upharpoonright \{\!|\ c\ |\!\} \sqsubseteq_{\mathrm{F}} \mathcal{B}_P \,[\![\, \{\!|\ Z\ |\!\} \,]\!]\, (\mathcal{B}_Q \,[\![\, \{\!|\ Z\ |\!\} \,]\!]\, R) \upharpoonright \{\!|\ c\ |\!\}) & \\
\quad \vee (c \in \mathcal{C}_Q \wedge (\mathcal{B}_Q \upharpoonright \{\!|\ c\ |\!\} \sqsubseteq_{\mathrm{F}} \mathcal{B}_Q \,[\![\, \{\!|\ Z\ |\!\} \,]\!]\, (\mathcal{B}_P \,[\![\, \{\!|\ Z\ |\!\} \,]\!]\, R) \upharpoonright \{\!|\ c\ |\!\}) & \\
& [P \text{ and } Q \text{ are protocol oriented components}] \\
\Rightarrow true &
\end{array}$$

Part 2 In this part, in order to prove that the composition is an enriched component contract, we prove the following assertion:

$$\begin{array}{ll}
\mathrm{dom}\, Prot^{\mathcal{K}}_{PQ} \subseteq \mathcal{C}_S \wedge \forall\, c : \mathrm{dom}\, Prot^{\mathcal{K}}_{PQ} \bullet Prot^{\mathcal{K}}_{PQ}(c) \sqsubseteq_{\mathrm{F}} Prot_{IMP}(S, c) & \\
& [\text{Definition 4.3, } Prot^{\mathcal{K}}_{PQ} = Prot^{\mathcal{K}}_{P} \cup Prot^{\mathcal{K}}_{Q}] \\
\Rightarrow (\mathrm{dom}\, Prot^{\mathcal{K}}_{P} \cup \mathrm{dom}\, Prot^{\mathcal{K}}_{Q}) \subseteq \mathcal{C}_S \wedge & \\
\quad \forall\, c : (\mathrm{dom}\, Prot^{\mathcal{K}}_{P} \cup \mathrm{dom}\, Prot^{\mathcal{K}}_{Q}) \bullet Prot^{\mathcal{K}}_{PQ}(c) \sqsubseteq_{\mathrm{F}} Prot_{IMP}(S, c) & \\
& [\text{Definition 3.10, } \mathcal{C}_S = \mathcal{C}_P \cup \mathcal{C}_Q,\ \mathcal{B}_S = \mathcal{B}_P \,|||\, \mathcal{B}_Q] \\
\Rightarrow (\mathrm{dom}\, Prot^{\mathcal{K}}_{P} \cup \mathrm{dom}\, Prot^{\mathcal{K}}_{Q}) \subseteq (\mathcal{C}_P \cup \mathcal{C}_Q) \wedge & \\
\quad \forall\, c : (\mathrm{dom}\, Prot^{\mathcal{K}}_{P} \cup \mathrm{dom}\, Prot^{\mathcal{K}}_{Q}) \bullet Prot^{\mathcal{K}}_{PQ}(c) \sqsubseteq_{\mathrm{F}} (\mathcal{B}_P \,|||\, \mathcal{B}_Q) \upharpoonright \{\!|\ c\ |\!\} & \\
& [\text{Definition 4.1, } \mathrm{dom}\, Prot^{\mathcal{K}}_{P} \subseteq \mathcal{C}_P \wedge \mathrm{dom}\, Prot^{\mathcal{K}}_{Q} \subseteq \mathcal{C}_Q] \\
\Rightarrow \forall\, c : (\mathrm{dom}\, Prot^{\mathcal{K}}_{P} \cup \mathrm{dom}\, Prot^{\mathcal{K}}_{Q}) \bullet Prot^{\mathcal{K}}_{PQ}(c) \sqsubseteq_{\mathrm{F}} (\mathcal{B}_P \,|||\, \mathcal{B}_Q) \upharpoonright \{\!|\ c\ |\!\} & \\
& [\alpha\mathcal{B}_P \cap \mathcal{C}_Q = \emptyset \wedge \alpha\mathcal{B}_Q \cap \mathcal{C}_P = \emptyset]
\end{array}$$

$\Rightarrow (\forall\, c : \mathrm{dom}\, Prot^{\mathcal{K}}_{P} \bullet Prot^{\mathcal{K}}_{PQ}(c) \sqsubseteq_{\mathrm{F}} \mathcal{B}_P \upharpoonright \{\!|\ c\ |\!\}) \wedge$
$(\forall\, c : \mathrm{dom}\, Prot^{\mathcal{K}}_{Q} \bullet Prot^{\mathcal{K}}_{PQ}(c) \sqsubseteq_{\mathrm{F}} \mathcal{B}_Q \upharpoonright \{\!|\ c\ |\!\})$

[Definitions 4.3 and 3.5]

$\Rightarrow (\forall\, c : \mathrm{dom}\, Prot^{\mathcal{K}}_{P} \bullet Prot^{\mathcal{K}}_{P}(c) \sqsubseteq_{\mathrm{F}} Prot_{IMP}(P, c)) \wedge$
$(\forall\, c : \mathrm{dom}\, Prot^{\mathcal{K}}_{Q} \bullet Prot^{\mathcal{K}}_{Q}(c) \sqsubseteq_{\mathrm{F}} Prot_{IMP}(Q, c))$

[Definition 4.1, P and Q are enriched component contracts]]

$\Rightarrow true$

Part 3 Finally, in this part, we prove the sentence in item 3. This is proved by contradiction, assuming that the statement is false.

$\mathrm{dom}\, Dec^{\mathcal{K}}_{PQ} \subseteq \mathcal{C}_S \wedge \mathrm{ran}\, Dec^{\mathcal{K}}_{PQ} \subseteq \mathcal{C}_S \wedge$
$(\exists\, c_1, c_2 : \mathcal{C}_S \bullet c_1\ Dec^{\mathcal{K}}_{PQ}\ c_2 \Rightarrow \neg\, \{c_1, c_2\}\ DecoupledIn\ S \vee \neg\, c_2\ Dec^{\mathcal{K}}_{PQ}\ c_1)$

[Definition 3.10, $\mathcal{C}_S = \mathcal{C}_P \cup \mathcal{C}_Q$]

$\Rightarrow \mathrm{dom}\, Dec^{\mathcal{K}}_{PQ} \subseteq \mathcal{C}_P \cup \mathcal{C}_Q \wedge \mathrm{ran}\, Dec^{\mathcal{K}}_{PQ} \subseteq \mathcal{C}_P \cup \mathcal{C}_Q \wedge$
$(\exists\, c_1, c_2 : \mathcal{C}_P \cup \mathcal{C}_Q \bullet c_1\ Dec^{\mathcal{K}}_{PQ}\ c_2 \Rightarrow \neg\, \{c_1, c_2\}\ DecoupledIn\ S \vee \neg\, c_2\ Dec^{\mathcal{K}}_{PQ}\ c_1)$

[Definition 4.3, $Dec^{\mathcal{K}}_{PQ}$ is symmetric]

$\Rightarrow Dec^{\mathcal{K}}_{PQ} = Dec^{\mathcal{K}}_{P} \cup Dec^{\mathcal{K}}_{Q} \cup \{(c_1, c_2) \mid (c_1 \in \mathcal{C}_Q \wedge c_2 \in \mathcal{C}_P) \vee (c_1 \in \mathcal{C}_P \wedge c_2 \in \mathcal{C}_Q)\} \wedge$
$(\exists\, c_1, c_2 : \mathcal{C}_P \cup \mathcal{C}_Q \bullet c_1\ Dec^{\mathcal{K}}_{PQ}\ c_2 \Rightarrow \neg\, \{c_1, c_2\}\ DecoupledIn\ S)$

[rewriting the statement, splitting in the four possible subsets of $Dec^{\mathcal{K}}_{PQ}$]

$\Rightarrow (\exists\, c_1, c_2 : \mathcal{C}_P \mid c_1\ Dec^{\mathcal{K}}_{P}\ c_2 \bullet \neg\, \{c_1, c_2\}\ DecoupledIn\ S) \vee$ [Definition x]
$(\exists\, c_1, c_2 : \mathcal{C}_Q \mid c_1\ Dec^{\mathcal{K}}_{Q}\ c_2 \bullet \neg\, \{c_1, c_2\}\ DecoupledIn\ S) \vee$
$(\exists\, c_1 : \mathcal{C}_P, c_2 : \mathcal{C}_Q \bullet \neg\, \{c_1, c_2\}\ DecoupledIn\ S) \vee$
$(\exists\, c_1 : \mathcal{C}_Q, c_2 : \mathcal{C}_P \bullet \neg\, \{c_1, c_2\}\ DecoupledIn\ S)$

$\Rightarrow (\exists\, c_1, c_2 : \mathcal{C}_P \mid c_1\ Dec^{\mathcal{K}}_{P}\ c_2 \bullet \neg\, Prot_{IMP}(S, c_1) \,|||\, Prot_{IMP}(S, c_1) \equiv_{\mathrm{F}} S \upharpoonright \{c_1, c2\}) \vee$
$(\exists\, c_1, c_2 : \mathcal{C}_Q \mid c_1\ Dec^{\mathcal{K}}_{Q}\ c_2 \bullet \neg\, Prot_{IMP}(S, c_1) \,|||\, Prot_{IMP}(S, c_1) \equiv_{\mathrm{F}} S \upharpoonright \{c_1, c2\}) \vee$
$(\exists\, c_1 : \mathcal{C}_P, c_2 : \mathcal{C}_Q \bullet \neg\, Prot_{IMP}(S, c_1) \,|||\, Prot_{IMP}(S, c_2) \equiv_{\mathrm{F}} S \upharpoonright \{c_1, c2\}) \vee$
$(\exists\, c_1 : \mathcal{C}_Q, c_2 : \mathcal{C}_P \bullet \neg\, Prot_{IMP}(S, c_1) \,|||\, Prot_{IMP}(S, c_2) \equiv_{\mathrm{F}} S \upharpoonright \{c_1, c2\})$

[Definitions 3.6 and 3.10, $\mathcal{B}_S = \mathcal{B}_P \,|||\, \mathcal{B}_Q$]

$\Rightarrow (\exists\, c_1, c_2 : \mathcal{C}_P \mid c_1\ Dec^{\mathcal{K}}_{P}\ c_2 \bullet \neg\, Prot_{IMP}(S, c_1) \,|||\, Prot_{IMP}(S, c_2) \equiv_{\mathrm{F}} P \upharpoonright \{c_1, c2\})$ [Definitions 3.5 and 3.10]
$(\exists\, c_1, c_2 : \mathcal{C}_Q \mid c_1\ Dec^{\mathcal{K}}_{Q}\ c_2 \bullet \neg\, Prot_{IMP}(S, c_1) \,|||\, Prot_{IMP}(S, c_2) \equiv_{\mathrm{F}} Q \upharpoonright \{c_1, c2\}) \vee$
$(\exists\, c_1 : \mathcal{C}_P, c_2 : \mathcal{C}_Q \bullet \neg\, Prot_{IMP}(S, c_1) \,|||\, Prot_{IMP}(S, c_2) \equiv_{\mathrm{F}} P \upharpoonright \{c_1\} \,|||\, Q \upharpoonright \{c_2\}) \vee$
$(\exists\, c_1 : \mathcal{C}_Q, c_2 : \mathcal{C}_P \bullet \neg\, Prot_{IMP}(S, c_1) \,|||\, Prot_{IMP}(S, c_2) \equiv_{\mathrm{F}} Q \upharpoonright \{c_1\} \,|||\, P \upharpoonright \{c_2\})$

$\Rightarrow (\exists\, c_1, c_2 : \mathcal{C}_P \mid c_1\ Dec^{\mathcal{K}}_{P}\ c_2 \bullet \neg\, Prot_{IMP}(P, c_1) \,|||\, Prot_{IMP}(P, c_2) \equiv_{\mathrm{F}} P \upharpoonright \{c_1, c2\}) \vee$
$(\exists\, c_1, c_2 : \mathcal{C}_Q \mid c_1\ Dec^{\mathcal{K}}_{Q}\ c_2 \bullet \neg\, Prot_{IMP}(Q, c_1) \,|||\, Prot_{IMP}(Q, c_2) \equiv_{\mathrm{F}} Q \upharpoonright \{c_1, c2\})$

[Contradiction with Definition x]

$\Rightarrow false$

□

Theorem 4.2 (Enriched Communication Composition Compatibility) An enriched communication composition is an enriched component contract.

The following lemmas are stated to help the proof of Theorem 4.2.

Lemma C.1. Let $S = P[ic \leftrightarrow oc]Q$ be a communication composition of two protocol oriented components P and Q, and c_1 and c_2 two channels, such that $\{c_1, ic\}\ DecoupledIn\ P$, $c_1 \in \mathcal{C}_P$, and $c_2 \in \mathcal{C}_Q$. Then $\{c_1, c_2\}\ DecoupledIn\ S$.

Proof.

$S \upharpoonright \{c_1, c_2\}$ [Definition 3.12]
$\equiv_{\mathrm{F}} \mathcal{B}_Q \,[\![\, \{\!|\, oc \,|\!\}\,]\!]\, (\mathcal{B}_P \,[\![\, \{\!|\, ic \,|\!\}\,]\!]\, BUFF^{\infty}_{IO}(ic, oc)) \upharpoonright \{\!|\, c_1, c_2 \,|\!\}$
[rewriting]
$\equiv_{\mathrm{F}} \mathcal{B}_Q \,[\![\, \{\!|\, oc \,|\!\}\,]\!]\, (\mathcal{B}_P \,[\![\, \{\!|\, ic \,|\!\}\,]\!]\, BUFF^{\infty}_{IO}(ic, oc)) \setminus \mathcal{C}_P \setminus \{ic, c_1\} \upharpoonright \{\!|\, c_1, c_2 \,|\!\}$ [applying the hiding operator]
$\equiv_{\mathrm{F}} \mathcal{B}_Q \,[\![\, \{\!|\, oc \,|\!\}\,]\!]\, (P \upharpoonright \{ic, c_1\} \,[\![\, \{\!|\, ic \,|\!\}\,]\!]\, BUFF^{\infty}_{IO}(ic, oc)) \upharpoonright \{\!|\, c_1, c_2 \,|\!\}$
[$\{c_1, ic\}$ *DecoupledIn* P]
$\equiv_{\mathrm{F}} \mathcal{B}_Q \,[\![\, \{\!|\, oc \,|\!\}\,]\!]\, ((P_{IMP}(P, c_1) \,|||\, P_{IMP}(P, ic)) \,[\![\, \{\!|\, ic \,|\!\}\,]\!]\, BUFF^{\infty}_{IO}(ic, oc)) \upharpoonright \{\!|\, c_1, c_2 \,|\!\}$
[rewriting]
$\equiv_{\mathrm{F}} P_{IMP}(P, c_1) \,|||\, (\mathcal{B}_Q \,[\![\, \{\!|\, oc \,|\!\}\,]\!]\, (P_{IMP}(P, ic) \,[\![\, \{\!|\, ic \,|\!\}\,]\!]\, BUFF^{\infty}_{IO}(ic, oc))) \upharpoonright \{\!|\, c_1, c_2 \,|\!\}$
[Definition 3.5, and considering the above]
$\equiv_{\mathrm{F}} P_{IMP}(S, c_1) \,|||\, P_{IMP}(S, c_2)$

Additionally, considering the proof above and that Q is a protocol oriented component, we conclude that: $P_{IMP}(S, c_1) \equiv_{\mathrm{F}} P_{IMP}(P, c_1) \wedge P_{IMP}(P, c_2) \sqsubseteq_{\mathrm{F}} P_{IMP}(S, c_2)$ □

Lemma C.2. Let *Ctr* be a component contract, c_1 a channel, and Z a set of channels, such that $\forall\, z : Z \bullet \{c_1, z\}$*DecoupledIn Ctr*. Then: $Ctr \upharpoonright \{c_1\} \cup Z \equiv_{\mathrm{F}} Ctr \upharpoonright \{c_1\} \,|||\, Ctr \upharpoonright Z$.

Proof. The idea of decoupled channels is close to the idea of non-interference of flows [RS01]. We translate it to our model as below, considering that P is a component contract, and A and B, such that $A \cap B = \emptyset$, then:

$P \upharpoonright A \equiv_{\mathrm{F}} (P \,[\![\, \{\!|\, B \,|\!\}\,]\!]\, SKIP) \upharpoonright A \wedge$
$P \upharpoonright B \equiv_{\mathrm{F}} (P \,[\![\, \{\!|\, A \,|\!\}\,]\!]\, SKIP) \upharpoonright B$
$\Leftrightarrow$
$P \upharpoonright A \cup B \equiv_{\mathrm{F}} P \upharpoonright A \,|||\, P \upharpoonright B$

Composing *SKIP* in parallel with P over the alphabet H has the effect of preventing all traces with events within the synchronizing set (A or B). The notion above presents two ways to represent that the view of the behaviour of P over A is independent of events within B, and vice-versa. Based on the notion presented above, we prove the lemma.

The events within c_1 are independent of events within Z, and vice-versa, $\forall\, z : Z \bullet \{c_1, z\}$ *DecoupledIn Ctr*. So:

$$Ctr \upharpoonright \{c_1\} \equiv_{\mathrm{F}} (Ctr \,[\![\, \{\!|\, Z \,|\!\}\,]\!]\, SKIP) \upharpoonright \{c_1\} \wedge Ctr \upharpoonright Z \equiv_{\mathrm{F}} (Ctr \,[\![\, \{\!|\, c_1 \,|\!\}\,]\!]\, SKIP) \upharpoonright Z$$

as a consequence

$$Ctr \upharpoonright \{c_1\} \cup Z \equiv_{\mathrm{F}} Ctr \upharpoonright \{c_1\} \,|||\, Ctr \upharpoonright Z$$

□

Lemma C.3. Let $S = P[ic \leftrightarrow oc]Q$ be a communication composition of two protocol oriented components P and Q, and c_1 and c_2 two channels, such that $\{c_1, c_2, ic\} \subseteq \mathcal{C}_P$, $\{c_1, ic\}$ *DecoupledIn* P, $\{c_1, c_2\}$ *DecoupledIn* P. Then $\{c_1, c_2\}$ *DecoupledIn* S.

Proof.

$S \upharpoonright \{c_1, c_2\}$ [Definition 3.6]
$\equiv_{\mathrm{F}} S \upharpoonright \{\!|\, c_1, c_2 \,|\!\}$ [rewriting]
$\equiv_{\mathrm{F}} S \upharpoonright \{\!|\, c_1, c_2, ic \,|\!\} \upharpoonright \{\!|\, c_1, c_2 \,|\!\}$ [Lemma C.2]
$\equiv_{\mathrm{F}} (S \upharpoonright \{c_1\} \,|||\, S \upharpoonright \{c_2, ic\}) \upharpoonright \{\!|\, c_1, c_2 \,|\!\}$ [Definition 3.5, and considering the above]]
$\equiv_{\mathrm{F}} (Prot_{IMP}(S, c_1) \,|||\, Prot_{IMP}(S, c_2)) \upharpoonright \{\!|\, c_1, c_2 \,|\!\}$

Additionally, considering the proof above and that Q is a protocol oriented component, we conclude that: $P_{IMP}(S, c_1) \equiv_{\mathrm{F}} P_{IMP}(P, c_1) \wedge P_{IMP}(P, c_2) \sqsubseteq_{\mathrm{F}} P_{IMP}(S, c_2)$ □

We finally move into the proof of Theorem 4.2.

Proof. Despite a communication composition being more complex than an interleave one, this proof follows steps similar to the proof of Theorem 4.1.

Based on the definitions 4.1 and 4.4, in order to an enriched interleave composition $P[ic \leftrightarrow oc]Q$ being an enriched component contract, we have to prove that:

(i) $S = \langle \mathcal{B}_P, \mathcal{R}_P, \mathcal{I}_P, \mathcal{C}_P \rangle_{\langle ic \rangle} \asymp_{\langle oc \rangle} \langle \mathcal{B}_Q, \mathcal{R}_Q, \mathcal{I}_Q, \mathcal{C}_Q \rangle$ is a protocol oriented component.

(ii) $\operatorname{dom} Prot^{\mathcal{K}}_{PQ} \subseteq \mathcal{C}_S \land \forall\, c : \operatorname{dom} Prot^{\mathcal{K}}_{PQ} \bullet Prot^{\mathcal{K}}(c) \sqsubseteq_{\mathrm{F}} Prot_{IMP}(S, c)$

(iii) $\operatorname{dom} Dec^{\mathcal{K}} \subseteq \mathcal{C}_S \land \operatorname{ran} Dec^{\mathcal{K}} \subseteq \mathcal{C}_S \land$
$(\forall\, c_1, c_2 : \mathcal{C}_S \bullet c_1\ Dec^{\mathcal{K}}\ c_2 \Rightarrow \{c_1, c_2\}\ DecoupledIn\ S \land c_2\ Dec^{\mathcal{K}}_{PQ}\ c_1)$

Considering the items above, we split the proof in three parts; one corresponding to each item. We do not show the proofs about preservation of context process and dual protocols. These follow a proof similar to the one associated to item ii.

Part 1 Prove that the composition is a protocol oriented component is the same as proving the following is true for any process R, channel c, and set of events Z, such that $c \in \mathcal{C}_S$, $Z \subseteq \alpha\mathcal{B}_S \cup \alpha R$, $\mathcal{B}_S \,[\![\{\!| Z |\!\}]\!]\, R$ is deadlock-free.

$\mathcal{B}_S \upharpoonright \{\!| c |\!\} \sqsubseteq_{\mathrm{F}} (\mathcal{B}_S \,[\![\{\!| Z |\!\}]\!]\, R) \upharpoonright \{\!| c |\!\}$

[Definition 3.12, $\mathcal{C}_S = (\mathcal{C}_P \cup \mathcal{C}_Q) \setminus \{ic, oc\}$, $\mathcal{B}_S = \mathcal{B}_P \,[\![\{\!| ic |\!\}]\!]\, (\mathcal{B}_Q \,[\![\{\!| oc |\!\}]\!]\, BUFF^{\infty}_{IO}(ic, oc))$]

$\Rightarrow$

$(c \in \mathcal{C}_P \setminus \{ic\} \land (\mathcal{B}_S \upharpoonright \{\!| c |\!\} \sqsubseteq_{\mathrm{F}} (\mathcal{B}_P \,[\![\{\!| ic |\!\}]\!]\, (\mathcal{B}_Q \,[\![\{\!| oc |\!\}]\!]\, BUFF^{\infty}_{IO}(ic, oc)) \,[\![\{\!| Z |\!\}]\!]\, R \upharpoonright \{\!| c |\!\}) \lor$
$(c \in \mathcal{C}_Q \setminus \{oc\} \land (\mathcal{B}_S \upharpoonright \{\!| c |\!\} \sqsubseteq_{\mathrm{F}} (\mathcal{B}_P \,[\![\{\!| ic |\!\}]\!]\, (\mathcal{B}_Q \,[\![\{\!| oc |\!\}]\!]\, BUFF^{\infty}_{IO}(ic, oc)) \,[\![\{\!| Z |\!\}]\!]\, R \upharpoonright \{\!| c |\!\})$

[rewriting, Definition C.1, $\alpha\mathcal{B}_P \cap \alpha\mathcal{B}_Q = \emptyset$]

$\Rightarrow$

$(c \in \mathcal{C}_P \setminus \{ic\} \land (\mathcal{B}_P \upharpoonright \{\!| c |\!\} \sqsubseteq_{\mathrm{F}} \mathcal{B}_P \,[\![\{\!| Z \cup \{ic\} |\!\}]\!]\, ((\mathcal{B}_Q \,[\![\{\!| oc |\!\}]\!]\, BUFF^{\infty}_{IO}(ic, oc)) \,[\![\{\!| Z |\!\}]\!]\, R) \upharpoonright \{\!| c |\!\}) \lor$
$(c \in \mathcal{C}_Q \setminus \{oc\} \land (\mathcal{B}_Q \upharpoonright \{\!| c |\!\} \sqsubseteq_{\mathrm{F}} \mathcal{B}_Q \,[\![\{\!| Z \cup \{oc\} |\!\}]\!]\, ((\mathcal{B}_P \,[\![\{\!| ic |\!\}]\!]\, BUFF^{\infty}_{IO}(ic, oc)) \,[\![\{\!| Z |\!\}]\!]\, R) \upharpoonright \{\!| c |\!\})$

[Definition C.1, P and Q are protocol oriented components]

$\Rightarrow true$

Part 2 In this part, in order to prove that the composition is an enriched component contract, we prove the following assertion:

$\operatorname{dom} Prot^{\mathcal{K}}_{PQ} \subseteq \mathcal{C}_S \land \forall\, c : \operatorname{dom} Prot^{\mathcal{K}}_{PQ} \bullet Prot^{\mathcal{K}}_{PQ}(c) \sqsubseteq_{\mathrm{F}} Prot_{IMP}(S, c)$

[Definition 4.4]

$\Rightarrow (\operatorname{dom} Prot^{\mathcal{K}}_{P} \setminus \{ic\} \cup \operatorname{dom} Prot^{\mathcal{K}}_{Q} \setminus \{oc\}) \subseteq \mathcal{C}_S \land$
$\forall\, c : (\operatorname{dom} Prot^{\mathcal{K}}_{P} \cup \operatorname{dom} Prot^{\mathcal{K}}_{Q}) \setminus \{ic, oc\} \bullet Prot^{\mathcal{K}}_{PQ}(c) \sqsubseteq_{\mathrm{F}} Prot_{IMP}(S, c)$

[Definitions 4.1 and 3.12, $\mathcal{C}_S = (\mathcal{C}_P \cup \mathcal{C}_Q) \setminus \{ic, oc\}$]

$\Rightarrow \forall\, c : (\operatorname{dom} Prot^{\mathcal{K}}_{P} \cup \operatorname{dom} Prot^{\mathcal{K}}_{Q}) \setminus \{ic, oc\} \bullet Prot^{\mathcal{K}}_{PQ}(c) \sqsubseteq_{\mathrm{F}} Prot_{IMP}(S, c)$

[rewriting using Definition 3.12]

$\Rightarrow (\forall\, c : \operatorname{dom} Prot^{\mathcal{K}}_{P} \bullet c \neq ic \land Prot^{\mathcal{K}}_{PQ}(c) \sqsubseteq_{\mathrm{F}} \mathcal{B}_P \,[\![\{\!| ic |\!\}]\!]\, (\mathcal{B}_Q \,[\![\{\!| oc |\!\}]\!]\, BUFF^{\infty}_{IO}(ic, oc)) \upharpoonright \{\!| c |\!\}) \land$
$(\forall\, c : \operatorname{dom} Prot^{\mathcal{K}}_{Q} \bullet c \neq oc \land Prot^{\mathcal{K}}_{PQ}(c) \sqsubseteq_{\mathrm{F}} \mathcal{B}_Q \,[\![\{\!| oc |\!\}]\!]\, (\mathcal{B}_P \,[\![\{\!| ic |\!\}]\!]\, BUFF^{\infty}_{IO}(ic, oc)) \upharpoonright \{\!| c |\!\})$

[P and Q are protocol oriented components with disjoint alphabets]

$\Rightarrow (\forall\, c : \operatorname{dom} Prot^{\mathcal{K}}_{P} \bullet c \neq ic \land Prot^{\mathcal{K}}_{PQ}(c) \sqsubseteq_{\mathrm{F}} \mathcal{B}_P \upharpoonright \{\!| c |\!\}) \land$
$(\forall\, c : \operatorname{dom} Prot^{\mathcal{K}}_{Q} \bullet c \neq oc \land Prot^{\mathcal{K}}_{PQ}(c) \sqsubseteq_{\mathrm{F}} \mathcal{B}_Q \upharpoonright \{\!| c |\!\})$

[Definitions 3.5 and 4.4]

$\Rightarrow (\forall\, c : \operatorname{dom} Prot^{\mathcal{K}}_{P} \bullet c \neq ic \land Prot^{\mathcal{K}}_{P}(c) \sqsubseteq_{\mathrm{F}} Prot_{IMP}(P, c)) \land$
$(\forall\, c : \operatorname{dom} Prot^{\mathcal{K}}_{Q} \bullet c \neq oc \land Prot^{\mathcal{K}}_{Q}(c) \sqsubseteq_{\mathrm{F}} Prot_{IMP}(Q, c))$

[Definition 4.1, P and Q are enriched component contracts]

$\Rightarrow \mathit{true}$

Part 3 Finally, in this part, we prove the sentence in item iii. This is proved by contradiction, assuming that the statement is false.

$\mathrm{dom}\ \mathit{Dec}^{\mathcal{K}}_{PQ} \subseteq \mathcal{C}_S \wedge \mathrm{ran}\ \mathit{Dec}^{\mathcal{K}}_{PQ} \subseteq \mathcal{C}_S \wedge$

$(\exists\, c_1, c_2 : \mathcal{C}_S \bullet c_1\ \mathit{Dec}^{\mathcal{K}}_{PQ}\ c_2 \Rightarrow \neg\, \{c_1, c_2\}\ \mathit{DecoupledIn}\ S \vee \neg\ c_2\ \mathit{Dec}^{\mathcal{K}}_{PQ}\ c_1)$

[Definition 3.12, $\mathcal{C}_S = (\mathcal{C}_P \cup \mathcal{C}_Q) \setminus \{ic, oc\}$]

$\Rightarrow \mathrm{dom}\ \mathit{Dec}^{\mathcal{K}}_{PQ} \subseteq (\mathcal{C}_P \cup \mathcal{C}_Q) \setminus \{ic, oc\} \wedge \mathrm{ran}\ \mathit{Dec}^{\mathcal{K}}_{PQ} \subseteq (\mathcal{C}_P \cup \mathcal{C}_Q) \setminus \{ic, oc\} \wedge$
$(\exists\, c_1, c_2 : (\mathcal{C}_P \cup \mathcal{C}_Q) \setminus \{ic, oc\} \bullet c_1\ \mathit{Dec}^{\mathcal{K}}_{PQ}\ c_2 \Rightarrow \neg\, \{c_1, c_2\}\ \mathit{DecoupledIn}\ S \vee \neg\ c_2\ \mathit{Dec}^{\mathcal{K}}_{PQ}\ c_1)$

[Definition 4.4, Dec_{PQ} is symmetric]

$\Rightarrow \mathit{Dec}^{\mathcal{K}}_{PQ} = \{(c_1, c_2) \mid (\{c_1, c_2\} \cap \{ic, oc\} = \emptyset) \wedge$
$((c_1\ \mathit{Dec}^{\mathcal{K}}_{P}\ ic \vee ic\ \mathit{Dec}^{\mathcal{K}}_{P}\ c_1) \wedge (c_2 \in \mathcal{C}_Q \vee c_1 \mathit{Dec}^{\mathcal{K}}_{P} c_2)) \vee$
$((oc\ \mathit{Dec}^{\mathcal{K}}_{Q}\ c_2 \vee c_2\ \mathit{Dec}^{\mathcal{K}}_{Q}\ oc) \wedge (c_1 \in \mathcal{C}_P \vee c_1 \mathit{Dec}^{\mathcal{K}}_{Q} c_2)))\}$
$\wedge\ (\exists\, c_1, c_2 : (\mathcal{C}_P \cup \mathcal{C}_Q) \setminus \{ic, oc\} \bullet c_1\ \mathit{Dec}^{\mathcal{K}}_{PQ}\ c_2 \Rightarrow \neg\, \{c_1, c_2\}\ \mathit{DecoupledIn}\ S)$

[rewriting the statement, splitting in the four possible subsets of the symmetric relation $\mathit{Dec}^{\mathcal{K}}_{PQ}$]

$\Rightarrow (\exists\, c_1, c_2 : \mathcal{C}_P \setminus \{ic\} \mid c_1\ \mathit{Dec}^{\mathcal{K}}_{P}\ c_2 \wedge c_1\ \mathit{Dec}^{\mathcal{K}}_{P}\ ic \bullet \neg\, \{c_1, c_2\}\ \mathit{DecoupledIn}\ S) \vee$ [Lemma C.1]
$(\exists\, c_1, c_2 : \mathcal{C}_Q \setminus \{oc\} \mid c_1\ \mathit{Dec}^{\mathcal{K}}_{Q}\ c_2 \wedge c_2\ \mathit{Dec}^{\mathcal{K}}_{Q}\ oc \bullet \neg\, \{c_1, c_2\}\ \mathit{DecoupledIn}\ S) \vee$
$(\exists\, c_1 : \mathcal{C}_P \setminus \{ic\}, c_2 : \mathcal{C}_Q \setminus \{oc\} \mid c_1\ \mathit{Dec}^{\mathcal{K}}_{P}\ ic \bullet \neg\, \{c_1, c_2\}\ \mathit{DecoupledIn}\ S) \vee$
$(\exists\, c_1 : \mathcal{C}_P \setminus \{ic\}, c_2 : \mathcal{C}_Q \setminus \{oc\} \mid c_2\ \mathit{Dec}^{\mathcal{K}}_{Q}\ oc \bullet \neg\, \{c_1, c_2\}\ \mathit{DecoupledIn}\ S)$

$\Rightarrow (\exists\, c_1, c_2 : \mathcal{C}_P \setminus \{ic\} \mid c_1\ \mathit{Dec}^{\mathcal{K}}_{P}\ c_2 \wedge c_1\ \mathit{Dec}^{\mathcal{K}}_{P}\ ic \bullet \neg\, \{c_1, c_2\}\ \mathit{DecoupledIn}\ S) \vee$
$(\exists\, c_1, c_2 : \mathcal{C}_Q \setminus \{oc\} \mid c_1\ \mathit{Dec}^{\mathcal{K}}_{Q}\ c_2 \wedge c_2\ \mathit{Dec}^{\mathcal{K}}_{Q}\ oc \bullet \neg\, \{c_1, c_2\}\ \mathit{DecoupledIn}\ S)$

[Contradiction with Lemma C.3]

$\Rightarrow \mathit{false}$

□

Theorem 4.3 (Enriched Feedback Composition Compatibility) An enriched feedback composition is an enriched component contract.

The following lemmas are stated to help in the proof of Theorem 4.3.

Lemma C.4. Let $S = P \asymp|^{\langle ic \rangle}_{\langle oc \rangle}$ be a feedback composition, P a protocol oriented component, and c_1 and c_2 two channels, such that $\{c_1, c_2\}\ \mathit{DecoupledIn}\ P$, $\{c_1, ic\}\ \mathit{DecoupledIn}\ P$, and $\{c_1, oc\}\ \mathit{DecoupledIn}\ P$. Then $\{c_1, c_2\}\ \mathit{DecoupledIn}\ S$.

Proof. Assume $P \upharpoonright \{ic, c_1\} \equiv_{\mathrm{F}} (\mathit{Pic}' \,|||\, \mathit{Pc}1')$, $\mathit{Prot}_{IMP}(P, ic) \sqsubseteq_{\mathrm{F}} \mathit{Pic}'$, and $\mathit{Prot}_{IMP}(P, c_1) \sqsubseteq_{\mathrm{F}} \mathit{Pc}1'$.

$S \upharpoonright \{c_1, c_2\}$ [Definition 3.12]
$\equiv_{\mathrm{F}}\ (\mathcal{B}_P \,[\![\, \{\!|\ ic, oc\ |\!\} \,]\!]\, \mathit{BUFF}^{\infty}_{IO}(ic, oc)) \upharpoonright \{\!|\ c_1, c_2\ |\!\}$

[rewriting]

$\equiv_{\mathrm{F}}\ \mathcal{B}_P \,[\![\, \{\!|\ ic, oc\ |\!\} \,]\!]\, \mathit{BUFF}^{\infty}_{IO}(ic, oc)) \upharpoonright \{\!|\ c_1, c_2, ic, oc\ |\!\} \upharpoonright \{\!|\ c_1, c_2\ |\!\}$

[Lemma C.2]

$\equiv_{\mathrm{F}}\ ((P \upharpoonright \{c_1\} \,|||\, P \upharpoonright \{c_2, ic, oc\}) \,[\![\, \{\!|\ ic, oc\ |\!\} \,]\!]\, \mathit{BUFF}^{\infty}_{IO}(ic, oc)) \upharpoonright \{\!|\ c_1, c_2\ |\!\}$

[Definition 3.5, and considering the above]

$\equiv_{\mathrm{F}}\ \mathit{Prot}_{IMP}(P, c_1) \,|||\, \mathit{Prot}_{IMP}(P, c_2)$

Additionally, considering the proof above and that Q is a protocol oriented component, we conclude that:

$P_{IMP}(S, c_1) \equiv_{\mathrm{F}} P_{IMP}(P, c_1) \wedge P_{IMP}(P, c_2) \sqsubseteq_{\mathrm{F}} P_{IMP}(S, c_2)$

□

Proof. Based on the definitions 4.1 and 4.5, in order to an enriched interleave composition $P\,[|||]\,Q$ be an enriched component contract, we have to prove that:

(i) $S = \langle \mathcal{B}_P, \mathcal{R}_P, \mathcal{I}_P, \mathcal{C}_P \rangle \succcurlyeq \Big|^{\langle ic \rangle}_{\langle oc \rangle}$ is a protocol oriented component.

(ii) $\mathrm{dom}\, Prot^{\mathcal{K}}_{PQ} \subseteq \mathcal{C}_S \wedge \forall\, c : \mathrm{dom}\, Prot^{\mathcal{K}}_{PQ} \bullet Prot^{\mathcal{K}}_{PQ}(c) \sqsubseteq_{\mathrm{F}} Prot_{IMP}(S, c)$

(iii) $\mathrm{dom}\, Dec^{\mathcal{K}} \subseteq \mathcal{C}_S \wedge \mathrm{ran}\, Dec^{\mathcal{K}} \subseteq \mathcal{C}_S \wedge$
$(\forall\, c_1, c_2 : \mathcal{C}_S \bullet c_1\ Dec^{\mathcal{K}}\ c_2 \Rightarrow \{c_1, c_2\}\ DecoupledIn\ S \wedge c_2\ Dec^{\mathcal{K}}_{PQ}\ c_1)$

Considering the items above, we split the proof in three parts; one corresponding to each item. We do not show the proofs about preservation of context process and dual protocols. These follow a proof similar to the one associated to item ii.

Part 1 prove that S is a protocol oriented component is a direct consequence of the fact that P is also a protocol oriented component. So, to prove that we have to prove the following statement for any process R, channel c, and set of events Z, such that $c \in \mathcal{C}_S$, $Z \subseteq \alpha\mathcal{B}_S \cup \alpha R$, $\mathcal{B}_S \,[\![\{\!|\ Z\ |\!\}]\!]\, R$ is deadlock-free (see Definition C.1).

$\mathcal{B}_S \upharpoonright \{\!|\ c\ |\!\} \sqsubseteq_{\mathrm{F}} (\mathcal{B}_S \,[\![\{\!|\ Z\ |\!\}]\!]\, R) \upharpoonright \{\!|\ c\ |\!\}$
[Definition 3.13, $\mathcal{C}_S = \mathcal{C}_P \setminus \{ic, oc\}$, $\mathcal{B}_S = \mathcal{B}_P \,[\![\{\!|\ ic, oc \{\!|\ ]\!] BUFF^{\infty}_{IO}(ic, oc)]$

$\Rightarrow (c \in \mathcal{C}_P \setminus \{ic, oc\} \wedge$ [Definition C.1, P is a protocol oriented component]
$((\mathcal{B}_P \,[\![\{\!|\ ic, oc\ |\!\}]\!]\, BUFF^{\infty}_{IO}(ic, oc)) \upharpoonright \{\!|\ c\ |\!\} \sqsubseteq_{\mathrm{F}} \mathcal{B}_P \,[\![\{\!|\ Z \cup \{ic, oc\}\ |\!\}]\!]\, (BUFF^{\infty}_{IO}(ic, oc)) \,|||\, R) \upharpoonright \{\!|\ c\ |\!\})$

$\Rightarrow \mathcal{B}_P \upharpoonright \{\!|\ c\ |\!\} \sqsubseteq_{\mathrm{F}} \mathcal{B}_P \,[\![\{\!|\ Z \cup \{ic, oc\}\ |\!\}]\!]\, (BUFF^{\infty}_{IO}(ic, oc)) \,|||\, R) \upharpoonright \{\!|\ c\ |\!\})$
[Definition C.1, P is a protocol oriented component]

$\Rightarrow true$

Part 2 In this part, in order to prove that the composition is an enriched component contract, we prove the following assertion:

$\mathrm{dom}\, Prot^{\mathcal{K}}_{PQ} \subseteq \mathcal{C}_S \wedge \forall\, c : \mathrm{dom}\, Prot^{\mathcal{K}}_{PQ} \bullet Prot^{\mathcal{K}}_{PQ}(c) \sqsubseteq_{\mathrm{F}} Prot_{IMP}(S, c)$
[Definitions 4.5 and 3.13]

$\Rightarrow (\mathrm{dom}\, Prot^{\mathcal{K}}_{P} \setminus \{ic, oc\}) \subseteq \mathcal{C}_P \setminus \{ic, oc\} \wedge$
$\forall\, c : \mathrm{dom}\, Prot^{\mathcal{K}}_{P} \setminus \{ic, oc\} \bullet Prot^{\mathcal{K}}_{P}(c) \sqsubseteq_{\mathrm{F}} Prot_{IMP}(S, c)$
[Definitions 4.1]

$\Rightarrow \forall\, c : \mathrm{dom}\, Prot^{\mathcal{K}}_{P} \setminus \{ic, oc\} \bullet Prot^{\mathcal{K}}_{P}(c) \sqsubseteq_{\mathrm{F}} Prot_{IMP}(S, c)$
[Definition 3.13]

$\Rightarrow \forall\, c : \mathrm{dom}\, Prot^{\mathcal{K}}_{P} \setminus \{ic, oc\} \bullet Prot^{\mathcal{K}}_{P}(c) \sqsubseteq_{\mathrm{F}} \mathcal{B}_P \,[\![\{\!|\ ic, oc\ |\!\}]\!]\, BUFF^{\infty}_{IO}(ic, oc) \upharpoonright \{\!|\ c\ |\!\})$
[P is a protocol oriented component]

$\Rightarrow \forall\, c : \mathrm{dom}\, Prot^{\mathcal{K}}_{P} \setminus \{ic, oc\} \bullet Prot^{\mathcal{K}}_{P}(c) \sqsubseteq_{\mathrm{F}} \mathcal{B}_P \upharpoonright \{\!|\ c\ |\!\})$
[Definition 4.1, P is an enriched component contracts]

$\Rightarrow true$

Part 3 Finally, in this part, we prove the sentence in item 3. This is proved by contradiction, assuming that the statement is false.

$\mathrm{dom}\, Dec^{\mathcal{K}}_{PQ} \subseteq \mathcal{C}_S \wedge \mathrm{ran}\, Dec^{\mathcal{K}}_{PQ} \subseteq \mathcal{C}_S \wedge$
$(\exists\, c_1, c_2 : \mathcal{C}_S \bullet c_1\ Dec^{\mathcal{K}}_{PQ}\ c_2 \Rightarrow \neg\, \{c_1, c_2\}\ DecoupledIn\ S \vee \neg\, c_2\ Dec^{\mathcal{K}}_{PQ}\ c_1)$
[Definition 3.12, $\mathcal{C}_S = \mathcal{C}_P \setminus \{ic, oc\}$]

$\Rightarrow \mathrm{dom}\, Dec^{\mathcal{K}}_{PQ} \subseteq \mathcal{C}_P \setminus \{ic, oc\} \wedge \mathrm{ran}\, Dec^{\mathcal{K}}_{PQ} \subseteq \mathcal{C}_P \setminus \{ic, oc\} \wedge$
$(\exists\, c_1, c_2 : (\mathcal{C}_P \cup \mathcal{C}_Q) \setminus \{ic, oc\} \bullet c_1\ Dec^{\mathcal{K}}_{PQ}\ c_2 \Rightarrow \neg\, \{c_1, c_2\}\ DecoupledIn\ S \vee \neg\, c_2\ Dec^{\mathcal{K}}_{PQ}\ c_1)$

[Definition 4.4, $Dec^{\mathcal{K}}_{PQ}$ is symmetric]

$$\Rightarrow Dec^{\mathcal{K}}_{S} = \{(c_1, c_2) \mid (\{c_1, c_2\} \cap \{ic, oc\} = \emptyset) \wedge c_1 Dec^{\mathcal{K}}_{P} c_2 \wedge ((c_1\ Dec^{\mathcal{K}}_{P}\ ic \wedge c_1\ Dec^{\mathcal{K}}_{P}\ oc) \vee (ic\ Dec^{\mathcal{K}}_{P}\ c_2 \wedge oc\ Dec^{\mathcal{K}}_{P}\ c_2))\}$$
$$\wedge (\exists\, c_1, c_2 : (\mathcal{C}_P \cup \mathcal{C}_Q) \setminus \{ic, oc\} \bullet c_1\ Dec^{\mathcal{K}}_{PQ}\ c_2 \Rightarrow \neg\, \{c_1, c_2\}\ DecoupledIn\ S)$$

[rewriting the statement, considering that $Dec^{\mathcal{K}}_{PQ}$ is a symmetric relation]

$$\Rightarrow \exists\, c_1, c_2 : \mathcal{C}_P \setminus \{ic\} \mid c_1\ Dec^{\mathcal{K}}_{P}\ c_2 \wedge c_1\ Dec^{\mathcal{K}}_{P}\ ic \wedge c_1\ Dec^{\mathcal{K}}_{P}\ oc \bullet \neg\, \{c_1, c_2\}\ DecoupledIn\ S$$

[Contradiction with Lemma C.4]

$$\Rightarrow false$$

□

Theorem 4.4 (Enriched Reflexive Composition Compatibility) An enriched reflexive composition is an enriched component contract.

Proof. This rule does not use metadata as side condition. It only uses the metadata to calculate the metadata of the resulting contract. Using Theorem 3.4 we guarantee that the result contract is a valid component contract regarding the $\mathcal{BRIC}$ components. We are left with the proof that the resulting metadata satisfies the conditions of an enriched component contract. This can be achieved using straightforward algebraic calculation on the metadata sets. □

References

[AB03] A. Aldini and M. Bernardo. A general approach to deadlock freedom verification for software architectures. In *International Symposium of Formal Methods Europe*, volume 2805 of *Lecture Notes in Computer Science*, pages 658–677. Springer, 2003.

[ACN02] J. Aldrich, C. Chambers, and D. Notkin. Archjava: connecting software architecture to implementation. In *International Conference on Software Engineering*. ACM Press, 2002.

[ADG98] R. Allen, R. Douence, and D. Garlan. Specifying and analyzing dynamic software architectures. In *Conference on Fundamental Approaches to Software Engineering (FASE)*, Lisbon, Portugal, March 1998.

[All97] R. Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon University, 1997. CMU Technical Report CMUU-CS–97–144.

[AOS+14] P. R. G. Antonino, M. V. M. Oliveira, A. C. A. Sampaio, K. E. Kristensen, and J. W. Bryans. Leadership election: An industrial sos application of compositional deadlock verification. In *NASA Formal Methods - 6th International Symposium, NFM 2014*, volume 8430 of *Lecture Notes in Computer Science*, pages 31–45. Springer, May 2014.

[Arb04] F. Arbab. Reo: a channel-based coordination model for component composition. *Mathematical. Structures in Computer Science*, 14(3):329–366, 2004.

[ASW14] Pedro R. G. Antonino, Augusto Sampaio, and Jim Woodcock. A refinement based strategy for local deadlock analysis of networks of csp processes. In *FM*, pages 62–77, 2014.

[BBNS10] S. Bensalem, M. Bozga, T.-H. Nguyen, and J. Sifakis. Compositional verification for component-based systems and application. *IET Software*, 4:181–193(12), June 2010.

[BCD02] M. Bernardo, P. Ciancarini, and L. Donatiello. Architecting families of software systems with process algebras. *ACM Transactions on Software Engineering and Methodology*, 11(4):386–426, 2002.

[BCL+06] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.B. Stefani. The FRACTAL component model and its support in Java. *Software: Practice and Experience*, 36(11-12):1257–1284, 2006.

[BGL+08] A. Basu, M. Gallien, C. Lesire, T.-H. Nguyen, S. Bensalem, F. Ingrand, and J. Sifakis. Incremental component-based construction and verification of a robotic system. In *18th European Conference on Artificial Intelligence*, volume 178 of *Frontiers in Artificial Intelligence and Applications*, pages 631–635. IOS Press, 2008.

[BGL+11] Saddek Bensalem, Andreas Griesmayer, Axel Legay, Thanh-Hung Nguyen, Joseph Sifakis, and Rongjie Yan. D-finder 2: towards efficient correctness of incremental design. In *NASA Formal Methods*, pages 453–458. Springer, 2011.

[BHP06] T. Bures, P. Hnetynka, and F. Plasil. SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model. In *4th International Conference on Software Engineering Research, Management and Applications*, pages 40–48. IEEE, 2006.

[Bra02] Premysl Brada. Metadata support for safe component upgrades. In *26th International Computer Software and Applications Conference*, pages 1017–1021. IEEE, 2002.

[Bra11] Premek Brada. Enhanced type-based component compatibility using deployment context information. *Electronic Notes in Theoretical Computer Science*, 279(2):17 – 31, 2011.

[BW13] Dirk Beyer and Philipp Wendler. Reuse of verification results. In *Model Checking Software*, pages 1–17. Springer, 2013.

[CCH+09] E. Cheung, X. Chen, H. Hsieh, A. Davare, A. Sangiovanni-Vincentelli, and Y. Watanabe. Runtime deadlock analysis for system level design. *Design Automation for Embedded Systems*, 13(4):287–310, 2009.

[Chi09] Z. Chi. Components Composition Compatibility Checking Based on Behavior Description and Roles Division. In *International Conference on Management of e-Commerce and e-Government*, pages 262–265. IEEE, 2009.

[CHLZ07] X. Chen, J. He, Z. Liu, and N. Zhan. A model of Component-Based programming. In *International Symposium on Fundamentals of Software Engineering*, volume 4767 of *Lecture Notes in Computer Science*, pages 191–206. Springer, 2007.

[CK96] S. Cheung and J. Kramer. Context constraints for compositional reachability analysis. *ACM Transactions on Software Engineering and Methodology*, 5(4):334–377, 1996.

[CSHL12] Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, and Axel Legay. Towards an incremental automata-based approach for software product-line model checking. In *16th International Software Product Line Conference*, pages 74–81. ACM, 2012.

[CZ07] D.C. Craig and WM Zuberek. Compatibility of software components-modeling and verification. In *International Conference on Dependability of Computer Systems*, pages 11–18. IEEE, 2007.

[DK06] L. DeMichiel and M. Keith. Enterprise javabeans specification, version 3.0. Technical Report JSR 220, Sun Microsystems, 2006.

[DR02] MS Dias and DJ Richardson. Identifying cause and effect relations between events in concurrent event-based components. In *17th IEEE International Conference on Automated Software Engineering*, pages 245–248. IEEE, 2002.

[dSOSO15] J. D. da S. Oliveira, A. C. A. Sampaio, and M. V. M. Oliveira. Constructive extensibility of trustworthy component-based systems. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, SAC '15. ACM, 2015.

[FFI+13] John Fitzgerald, Simon Foster, Claire Ingram, Peter Gorm Larsen, and Jim Woodcock. Model-based engineering for systems of systems: the compass manifesto. Technical report, COMPASS, 2013. Available at `http://www.compass-research.eu/Project/Publications/MBESoS.pdf`.

[FG03] A. Farias and Y. Guéhéneuc. On the coherence of component protocols. *Electronic Notes Theoretical Computer Science*, 82(5):42–53, 2003.

[FLF01] R.B. Findler, M. Latendresse, and M. Felleisen. Behavioral contracts and behavioral subtyping. *ACM SIGSOFT Software Engineering Notes*, 26(5):229–236, 2001.

[For98] Formal Systems (Europe) Ltd. *Process Behaviour Explorer - ProBE User Manual*, 1998.

[For12] Formal Systems Ltd. *FDR2: User Manual, version 2.94*, 2012.

[Geo86] Bate George. Mascot 3: an informal introductory tutorial. *Software Engineering Journal*, 1:95–102(7), May 1986.

[GGMC+06] G. Gößler, S. Graf, M. Majster-Cederbaum, M. Martens, and J. Sifakis. Ensuring properties of interaction systems. In *Theory and Practice on Program Analysis and Compilation*, volume 4444 of *Lecture Notes in Computer Science*, pages 201–224. Springer, 2006.

[GGMC+07] G. Gößler, S. Graf, M. Majster-Cederbaum, M. Martens, and J. Sifakis. An approach to modelling and verification of component based systems. In *Current Trends in Theory and Practice of Computer Science*, volume 4362 of *Lecture Notes in Computer Science*, pages 295–308. Springer, 2007.

[Ghe12] Carlo Ghezzi. Evolution, adaptation, and the quest for incrementality. In *Large-Scale Complex IT Systems. Development, Operation and Management*, pages 369–379. Springer, 2012.

[GRABR14] Thomas Gibson-Robinson, Philip Armstrong, Alexandre Boulgakov, and A.W. Roscoe. FDR3 — A Modern Model Checker for CSP. In Erika Ábrahám and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 8413 of *Lecture Notes in Computer Science*, pages 187–201, 2014.

[HGK+06] M. Hepner, R. Gamble, M. Kelkar, L. Davis, and D. Flagg. Patterns of conflict among software components. *The Journal of Systems & Software*, 79(4):537–551, 2006.

[HJK10a] R. Hennicker, S. Janisch, and A. Knapp. On the observable behaviour of composite components. *Electronic Notes in Theoretical Computer Science*, 260:125–153, 2010.

[HJK10b] Rolf Hennicker, Stephan Janisch, and Alexander Knapp. On the observable behaviour of composite components. *ENTCS*, 260:125–153, 2010.

[HLL06a] J. He, X. Li, and Z. Liu. rCOS: a refinement calculus of object systems. *Theoretical Computer Science*, 365(1-2):109–142, 2006.

[HLL06b] J. He, X. Li, and Z. Liu. A theory of reactive components. *Electronic Notes in Theoretical Computer Science*, 160:173–195, 2006.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[IM08] J. Ivers and G. Moreno. PACC starter kit: developing software with predictable behavior. In *ICSE Companion*, pages 949–950. ACM, 2008.

[Kwi07] X.W.M. Kwiatkowska. Compositional state space reduction using untangled actions. In *13th International Workshop on Expressiveness in Concurrency*, volume 175 of *Electronic Notes in Theoretical Computer Science*, pages 27–46, 2007.

[Laz99] R. Lazić. *A semantic study of data-independence with applications to the mechanical verification of concurrent systems*. PhD thesis, Oxford University, 1999.

[LD00] Gary Leavens and Krishna Dhara. Concepts of behavioral subtyping and a sketch of their extension to Component-Based systems. In *Foundations of Component-Based Systems*, pages 113–135. Cambridge University Press, 2000.

[Lev95] N. Leveson. *Safeware: System Safety and Computers*. Addison-Wesley, 1995.

[LMC10] C. Lambertz and M. E. Majster-Cederbaum. Port protocols for deadlock-freedom of component systems. In S. Bliudze, R. Bruni, D. Grohmann, and A. Silva, editors, *ICE*, volume 38 of *EPTCS*, pages 7–11, 2010.

[LU05] K.-K. Lau and V. Ukis. Component metadata in component-based software development: A survey. Preprint CSPP-34, School of Computer Science, The University of Manchester, October 2005.

[LW94] B. H. Liskov and J. M. Wing. A Behavioural Notion of Subtyping. *ACM Transactions on Programming Languages and Systems*, **16**(6), 1994.

[Mah90] M. Mahoney. The roots of software engineering. *CWI Quarterly*, 3(4):325–334, 1990.

[Mar96] Jeremy M. R. Martin. *The Design and Construction of Deadlock-Free Concurrent Systems*. PhD thesis, University of Buckingham, 1996.

[MCM07] M. Majster-Cederbaum and M. Martens. Robustness in interaction systems. In *27th International Conference on Formal Methods for Networked and Distributed Systems*, volume 4574 of *Lecture Notes of Computer Science*, pages 325–340. Springer, 2007.

[MCM08] M Majster-Cederbaum and M. Martens. Compositional analysis of deadlock-freedom for tree-like component architectures. In *8th ACM international conference on Embedded software*, pages 199–206. ACM, 2008.

[MCMM07] M. Majster-Cederbaum, M. Martens, and C. Minnameier. A polynomial-time checkable sufficient condition for deadlock-freedom of component-based systems. *SOFSEM 2007: Theory and Practice of Computer Science*, pages 888–899, 2007.

[MCMM08] M. Majster-Cederbaum, M. Martens, and C. Minnameier. Liveness in Interaction Systems. *Electronic Notes in Theoretical Computer Science*, 215:57–74, 2008.

[MH05] P. Merson and S. Hissam. Predictability by construction. In *20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 134–135. ACM, 2005.

[Mic11] Microsoft Developer Network. Component object model technologies. Technical report, http://www.microsoft.com/com, 2011.

[Min07] C. Minnameier. Local and global deadlock-detection in component-based systems are NP-hard. *Information Processing Letters*, 103(3):105–111, 2007.

[MJG+10] A. Mota, J. Jesus, A. Gomes, F. Ferri, and E. Watanabe. Evolving a Safe System Design Iteratively. In *29th International Conference Computer Safety, Reliability, and Security*, volume 6351 of *Lecture Notes in Computer Science*, pages 361–374. Springer, 2010.

[MK96] J. Magee and J. Kramer. Dynamic structures in software architecture. In *4th Symposium On the Foundations of Software Engineering*, pages 3–14. ACM Press, 1996.

[MT00] N. Medvidovic and R. Taylor. A classification and comparison framework for software architecture description languages. *Transactions on Software Engineering*, 26(1):70–93, 2000.

[MW97] J.M.R. Martin and P.H. Welch. A design strategy for deadlock-free concurrent systems. *Transputer Communications*, 3(4):215–232, 1997.

[Nie93] O. Nierstrasz. Regular types for active objects. *ACM Sigplan Notices*, 28(10):1–15, 1993.

[Obj07] Object Management Group. Unified Modeling Language, Superstructure, V2.1.2. Technical Report formal/2007-11-02, OMG, 2007. OMG Adopted Specification.

[ODR+07] Alessandro Orso, Hyunsook Do, Gregg Rothermel, Mary Jean Harrold, and David S Rosenblum. Using component metadata to regression test component-based software. *Software Testing, Verification and Reliability*, 17(2):61–94, 2007.

[OMG12] OMG. OMG Systems Modeling Language (OMG SysML), Version 1.3. Technical report, Object Management Group, 2012.

[OPRW13] Joel Ouaknine, Hristina Palikareva, A. W. Roscoe, and James Worrell. A static analysis framework for livelock freedom in csp. *Logical Methods in Computer Science*, **9**(3), 2013.

[OSA+13] M. V. M. Oliveira, A. C. A. Sampaio, P. R. G. Antonino, R. T. Ramos, A. L. C. Cavancalti, and J. C. P Woodcock. Compositional Analysis and Design of CML Models. Technical Report D24.1, COMPASS Deliverable, 2013. Available at http://www.compass-research.eu/.

[PA98] G. Papadopoulos and F. Arbab. Coordination models and languages. *Advances in Computers - The Engineering of Large Systems*, 46:330–401, 1998.

[Pla05] F. Plasil. Enhancing component specification by behavior description: the SOFA experience. In *4th international symposium on Information and communication technologies*, page 190. Trinity College Dublin, 2005.

[PV02] F. Plasil and S. Visnovsky. Behavior protocols for software components. *IEEE Transactions on Software Engineering*, 28(11):1056–1076, 2002.

[Ros98] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall Series in Computer Science. Prentice-Hall, 1998.

[Ros05] A. W. Roscoe. The pursuit of buffer tolerance. Technical report, Oxford University, 2005.

[Ros06] A. W. Roscoe. Confluence thanks to extensional determinism. *Electronic Notes in Theoretical Computer Science*, 162:305–309, 2006.

[Ros10] A.W. Roscoe. *Understanding Concurrent Systems*. Springer-Verlag New York, Inc., New York, NY, USA, 1st edition, 2010.

[RS01] P. Ryana and S. Schneider. Process algebra and non-interference. *Journal of Computer Security*, 9(1):75–103, 2001.

[RSM06] R. T. Ramos, A. C. A. Sampaio, and A. C. Mota. Transformation laws for UML-RT. In *8th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems*, volume 4037 of *Lecture Notes in Computer Science*, pages 123–138. Springer, 2006.

[RSM09] R. T. Ramos, A. C. A. Sampaio, and A. C. Mota. Systematic development of trustworthy component systems. In *2nd World Congress on Formal Methods*, volume 5850 of *Lecture Notes in Computer Science*, pages 140–156. Springer, 2009.

[RSM10] R. T. Ramos, A. C. A. Sampaio, and A. C. Mota. Conformance notions for the coordination of interaction components. *Science of Computer Programming*, 75(5):350–373, 2010.

[SGW94] B. Selic, G. Gullekson, and P. Ward. *Real-Time Object-Oriented Modeling*. John Wiley & Sons, Inc., 1994.

[Sif10] J. Sifakis. Component-Based Construction of Heterogeneous Real-Time Systems in Bip. *The Future of Software Engineering*, page 150, 2010.

[SNMI14] Augusto Sampaio, Sidney Nogueira, Alexandre Mota, and Yoshinao Isobe. Sound and mechanised compositional verification of input-output conformance. *Softw. Test., Verif. Reliab.*, 24(4):289–319, 2014.

[Spi04] B. Spitznagel. *Compositional Transformation of Software Connectors*. PhD thesis, Carnegie Mellon University, 2004. Number: CMU-CS-04-128.

[SR98] B. Selic and J. Rumbaugh. Using UML for modeling complex RealTime systems. Technical report, Rational Software Corporation, 1998.

[SSCS12] Nishant Sinha, Nimit Singhania, Satish Chandra, and Manu Sridharan. Alternate and learn: finding witnesses without looking all over. In *Computer Aided Verification*, pages 599–615. Springer, 2012.

[Szy02] C. Szyperski. *Component Software: Beyond Object–Oriented Programming*. Addisonn-Wesley, 2002.

[Wal03] Kurt C. Wallnau. Volume III: a technology for predictable assembly from certifiable components. Technical Report CMU/SEI-2003-TR-009, Software Engineering Institute, Carnegie Mellon University, 2003.

[Weh00] H. Wehrheim. Specification of an automatic manufacturing system: A case study in using integrated formal methods. In *3rd Internationsl Conference Fundamental Approaches to Software Engineering*, volume 1783 of *Lecture Notes in Computer Science*, pages 334–348. Springer, 2000.

[ZKL10] N. Zhan, E. Kang, and Z. Liu. Component publications and compositions. *Unifying Theories of Programming*, pages 238–257, 2010.

[ZM10] H. Zeng and H. Miao. Deadlock Detection for Parallel Composition of Components. *Computer and Information Science*, pages 23–34, 2010.

[Zub11] W. Zuberek. Incremental composition of software components. *Dependable Computer Systems*, pages 301–311, 2011.