The Journal of Systems and Software 167 (2020) 110598

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

A refinement checking based strategy for component-based systems evolution

José Dihego[a,*], Augusto Sampaio[b], Marcel Oliveira[c]

[a] Coordenação de Informática, IFBA, Feira de Santana BA, Brazil
[b] Centro de Informática, Universidade Federal de Pernambuco, Recife-PE, Brazil
[c] Departamento de Informática e Matemática Aplicada, UFRN, Natal-RN, Brazil

ARTICLE INFO

Article history:
Received 7 April 2018
Revised 31 March 2020
Accepted 7 April 2020
Available online 17 May 2020

Keywords:
Component extensibility
Correctness by construction
Behavioural specification
CSP
FDR4

ABSTRACT

We propose inheritance and refinement relations for a CSP-based component model ($\mathcal{BRIC}$), which supports a constructive design based on composition rules that preserve classical concurrency properties such as deadlock freedom. The proposed relations allow extension of functionality, whilst preserving behavioural properties. A notion of extensibility is defined on top of a behavioural relation called *convergence*, which distinguishes inputs from outputs and the context where they are communicated, allowing extensions to reuse existing events with different purposes. We mechanise the strategy for extensibility verification using the FDR4 tool, and illustrate our results with an autonomous healthcare robot case study.

1. Introduction

Designing and reasoning about computational systems that interoperate in heterogeneous environments, and are expected to cope with a variety of application domains, is a challenge of increasing importance. Modelling and analysis techniques become manageable, and possibly scalable, when systems can be understood and developed by smaller (less complex) units. This is the core idea of component-based model driven development (CB-MDD) (Szyperski, 1998), by which a system is defined as a set of components and their connections. This has become an essential infrastructure to the emergence of some other important development paradigms like Service Oriented Computing (Papazoglou and Georgakopoulos, 2003) and Systems of Systems (Ackoff, 1971).

There are several component models for MDD such as, for instance, those presented in Jifeng et al. (2006), Meng and Barbosa (2006), Hennicker et al. (2010), Arbab (2004), and a variety of approaches to analyse and refine component-based systems as, for example, the ones reported in Chen et al. (2009a), Büchi and Sekerinski (1998), Kurki-Suonio (1999). Nevertheless, formal support to component evolution, possibly involving an increment of interface via the addition of new functionality, has not been properly addressed. Component inheritance arises as a natural aspect to be provided by a CB-MDD approach, as a way to support evolution. It has been a successful feature present in object oriented languages from the beginning (Liskov and Wing, 1994; Wegner and Zdonik, 1988; America, 1991); however, differently from object-orientation, our focus is not on defining subtyping, but extension relations that preserve some notions of conformance, a safe way to evolve component systems considering structure and behaviour.

To achieve this goal in a controlled manner, component inheritance must obey the substitutability principle (Liskov, 1987; Wegner and Zdonik, 1988): an instance of the subcomponent should be usable wherever an instance of the supercomponent was expected, without a component, playing the role of a client, being able to observe any difference.

Some works have proposed inheritance relations for behavioural specifications (Liskov and Wing, 1994; America, 1991; Bowman and Derrick, 1999; Puntigam, 1996; Wehrheim, 2003; Dihego et al., 2013). The first four define behaviour in terms of pre- and postconditions, and structure by method signatures (covariance and contravariance), but do not address reactive behaviour. Although the approaches in Wehrheim (2003) and Dihego et al. (2013) consider active objects, they do not focus on structure. Furthermore, none of the mentioned works differentiate the nature of input and output events nor the context in which they are communicated. This differentiation is relevant for a large class of specifications (such as Enterprise JavaBeans (EJB) Rubinger and Burke, 2010, client-server protocols and Model-View-Controller (MVC) components Krasner and Pope, 1988) where components control outputs,

* Corresponding author.
E-mail addresses: jose.dihego@ifba.edu.br, josedihego@gmail.com (J. Dihego), acas@cin.ufpe.br (A. Sampaio), marcel@dimap.ufrn.br (M. Oliveira).

https://doi.org/10.1016/j.jss.2020.110598
0164-1212/

while the environment controls inputs. Finally, these works do not consider the impact of these relations over behavioural properties, such as preservation of deadlock freedom. Therefore, although these approaches deal with substitutability, the proposed inheritance relations do not guarantee deadlock free evolutions, in some cases because it is not an explicit concern (Liskov and Wing, 1994) or just because it admits, implicitly, the introduction of deadlock through weaker inheritance relations (Wehrheim, 2003).

In the current work, we propose component extension relations that address how components behave, how they are structured (channels and interfaces), distinguish the nature of inputs and outputs and, moreover, guarantee safe evolution by means of classical properties preservation.

We develop a fully formal and mechanised approach to component based model driven development. Particularly, our work is in the context of $\mathcal{BRIC}$ (Ramos et al., 2009), which formalises the core CB-MDD concepts (Szyperski, 1998) (components, interfaces, channels and behaviour) and, moreover, supports compositions, where deadlock freedom is ensured by construction. This approach covers not only tree topologies, but also cyclic ones.

In previous work Dihego et al. (2015), we defined $\mathcal{BRIC}$ component refinement and inheritance notions based on the concept of behavioural convergence. Here, we significantly extend our previous work with the following contributions.

- Detailed proofs of lemmas that relate the proposed notions of inheritance, as well as the proofs of theorems that relate inheritance with refinement, and the fact that inheritance preserves deadlock freedom.
- A strategy, based on refinement checking, using the FDR4 tool (Gibson-Robinson et al., 2014), to mechanically verify whether two given component models are related by the proposed inheritance notions.
- An elaborate case study, an autonomous healthcare robot, used to illustrate the overall approach.
- A more detailed account of related work.

It is important to emphasise that the focus of this paper is on an infrastructure we have devised to formally and mechanically check model evolution based on a notion of convergence. Nevertheless, an approach to design model extensions that ensure convergence, in a constructive way, is out of the scope of this paper. This important, complementary contribution, is considered as one of our major topics for future work, and is discussed in more detail in the concluding section.

We structure the paper as follows. Section 2 introduces the $\mathcal{BRIC}$ component model. Section 3 presents a congruent semantics for $\mathcal{BRIC}$, a refinement notion and two inheritance relations, based on a concept called behavioural *convergence*, which allows extensions but preserves conformance. A strategy to mechanically verify conformance with respect to the proposed relations is the subject of Section 4. Our results are illustrated by a case study of an autonomous healthcare robot in Section 5. We conclude with related work in Section 6 and summarise our contributions and future work in Section 7.

2. The $\mathcal{BRIC}$ component model

In the $\mathcal{BRIC}$ component model, one specifies components, connectors and their behaviour in the Communicating Sequential Processes (CSP) language (Roscoe, 1998); $\mathcal{BRIC}$ provides a set of rules to assemble components. The behavioural properties (particularly, deadlock freedom) of compositions using the $\mathcal{BRIC}$ rules are guaranteed by construction, verified by local analyses (Ramos, 2011), which include cyclic networks (Antonino et al., 2014). More recently, some additional rules were proposed to ensure livelock freedom (Filho et al., 2016; 2018) and to avoid nondeterminism (Otoni et al., 2017).

2.1. CSP

A process algebra like CSP can be used to describe systems composed of interacting components, which are independent self-contained processes with interfaces used to interact with the environment. Such formalisms provide mechanisms to specify and reason about interaction between components. Furthermore, phenomena that are exclusive to the concurrent world, that arise from the combination of components rather than from individual components, like deadlock and livelock, can be more easily understood and controlled using such formalisms. Tool support is another reason for the success of CSP in industrial applications. For instance, FDR4 (Gibson-Robinson et al., 2014) provides an automatic analysis of model refinement and of properties like deadlock, livelock and determinism. Each of these classical properties is very complex to verify in a behavioural model. In this context, we contribute to $\mathcal{BRIC}$ by developing a strategy to evolve component specifications without introducing deadlock. More recently, some additional rules were proposed (Filho et al., 2016) to also ensure livelock freedom.

Here we use the machine-readable version of CSP (CSP_M (Roscoe, 1998)). The two basic CSP processes are `STOP` (deadlock) and `SKIP` (successful termination). The prefixing `c -> P` is initially able to perform only the event `c`; afterwards it behaves like process `P`. A boolean guard may be associated with a process: given a predicate `g`, if it holds, the process `g & c?x -> A` inputs a value through channel `c` and assigns it to the variable `x`, and then behaves like `A`, which has the variable `x` in scope; the process deadlocks otherwise. It can also be defined as `if g then c?x -> A else STOP`. Multiple inputs and outputs are also possible. For instance, `c?x?y!z` inputs two values that are assigned to `x` and `y` and outputs the value resulting from the evaluation of expression `z`.

The external choice `P1 [] P2` initially offers events of both processes. The engagement of the process in an event resolves the choice in favor of the process that performs it. On the other hand, the environment has no control over the internal choice `P1 | | P2`. The sequence operator `P1;P2` combines processes `P1` and `P2` in sequence. The synchronised parallel composition `P1 [| cs |] P2` synchronises `P1` and `P2` on the channels in the set `cs`; events that are not in `cs` occur independently. Processes composed in interleaving, as in `P1 ||| P2`, run independently. The event hiding operator `P \ cs` encapsulates (internalises) in `P` the events that are in the channel set `cs`, which become no longer visible to the environment.

In this work we use two denotational models of CSP: traces ($\mathcal{T}$) and stable failures, or just failures ($\mathcal{F}$). A trace of a process `P` is a sequence of events that it can perform and we define $\mathcal{T}(\texttt{P})$ to be the set of all its finite traces. For example $\mathcal{T}(\texttt{e1 -> e2 -> STOP}) = \{\langle\rangle, \langle\texttt{e1}\rangle, \langle\texttt{e1}, \texttt{e2}\rangle\}$. The set $\mathcal{F}(\texttt{P})$ consists of all stable failures (s, X), where s is a trace of `P` ($s \in \mathcal{T}(\texttt{P})$) and X is a set of events `P` can refuse in some stable state after s. A stable state is one in which a process can only engage in a visible event (registered in the process trace). The invisible event is represented in CSP as τ; it can happen, for example, in the internal choice `P | | Q`, which will be implemented as a process that can take an invisible τ event to each of `P` and `Q`. We summarise this discussion in Appendix A.

2.2. $\mathcal{BRIC}$

A component is defined as a contract (Definition 1) that specifies its behaviour, communication points (or channels) and their types.

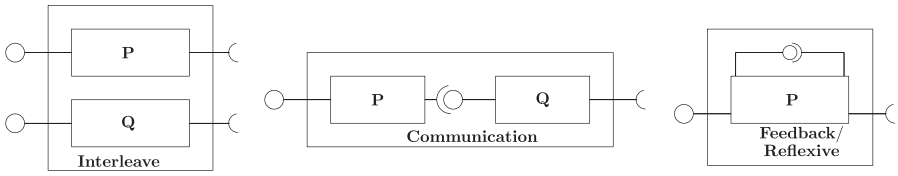


Fig. 1. Composition rules.

Definition 1 (Component contract)**.** A component contract $Ctr{:}\langle \mathcal{B}, \mathcal{R}, \mathcal{I}, \mathcal{C}\rangle$ comprises an observational behaviour $\mathcal{B}$ specified as a CSP process, a set of communication channels $\mathcal{C}$, a set of interfaces $\mathcal{I}$, and a **total** function $\mathcal{R} : \mathcal{C} \mapsto \mathcal{I}$ between channels and interfaces of the contract, such that $\mathcal{B}$ is an I/O process.

We require the CSP process $\mathcal{B}$ to be an I/O process, which is a non-divergent processes with infinite traces. Moreover, it offers to the environment the choice over its inputs (external choice) but reserves the right to choose among its outputs (internal choice). It represents a wide range of specifications, including the server-client protocol, where the client sends requests (inputs) to the server, which decides the outputs to be returned to the client.

Definition 2 (I/O process)**.** We say that a CSP process P is an I/O process if it satisfies the following five conditions, which are formally presented in Ramos (2011) and Ramos et al. (2009):

(1) **I/O channels**: Every channel in P has its events partitioned into inputs and outputs.
(2) **infinite traces**: P has an infinite set of traces (but finite state-space).
(3) **divergence-freedom**: P is divergence-free.
(4) **input determinism**: If a set of input events in P are offered to the environment, none of them are refused.
(5) **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.

Contracts can be composed using any of the four rules available in the model: interleaving, communication, feedback, or reflexive composition. Each of these rules impose associated side conditions, which must be satisfied by the contracts and channels involved in the composition in order to guarantee deadlock freedom by construction.

The rules provide asynchronous pairwise compositions, mediated by buffers, and focus on the preservation of deadlock freedom in the resulting component. Using the rules, developers may connect channels of two components, or even of the same component. The four rules are illustrated in Fig. 1. Two of them, feedback and reflexive, are merged (see the rightmost scheme in the figure) as both entail connection of channels of a same component; more explanation is presented in the sequel for the composition rule and the detailed formalisation in Appendix B for the rest of the rules.

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

The communication composition (Definition 3) states the most common way for linking channels (Ramos et al., 2009) of two different entities. It is given in terms of asynchronous binary composition of channels from P and Q (Definition 4).

Definition 3 (Communication composition)**.** Let P and Q be two component contracts, and ic and oc two communication channels. The communication composition of P and Q (namely $P[ic{\leftrightarrow}oc]Q$) via ic and oc is defined as follows:

$$P[ic \leftrightarrow oc]Q = P_{\langle ic\rangle} \asymp_{\langle oc\rangle} Q$$

This rule assumes the components behaviours on channels ic and oc are I/O confluent, strong compatible and satisfy the finite output property (FOP). These properties are detailed in Roscoe and Dathi (1987), Roscoe (2006), Ramos (2011): I/O confluence means that choosing between inputs (deterministically) or outputs (non-deterministically) does not prevents other inputs/outputs offered alongside from being communicated afterwards; two processes are strong compatible if all outputs produced by one are consumed by the other, and vice versa; finally, FOP guarantees that a process cannot output forever, so eventually it inputs after a finite sequence of outputs. The resulting component $P_{\langle ic\rangle} \approx_{\langle oc\rangle} Q$ is the binary composition of P and Q on channels ic and oc (Definition 4).

The asynchronous binary composition hooks two components, say P and Q, with disjoint communication points, by their respective channels c and z. Instead of communicating directly, their communications are buffered.

Definition 4 (Asynchronous binary composition)**.** Let P and Q be two distinct component contracts, and $c \in \mathcal{C}_P$ and $z \in \mathcal{C}_Q$ two channels, such that $\mathcal{C}_P$ and $\mathcal{C}_Q$ are disjoint. Then, the asynchronous binary composition of P and Q, $P_{\langle c\rangle} \approx_{\langle z\rangle} Q$, is given by:

$$P_{\langle c\rangle} \asymp_{\langle z\rangle} Q = \langle \mathcal{B}_P \parallel [\{\!|c|\!\}] BUFF^n_{IO}(R^{c\to z}_{IO}, R^{z\to c}_{IO}) \parallel [\{\!|z|\!\}] \mathcal{B}_Q, \mathcal{R}', \mathcal{I}', \mathcal{C}'\rangle$$

where $\mathcal{C}' = (\mathcal{C}_P \cup \mathcal{C}_Q)\backslash\{c, z\}$, $\mathcal{R}' = \mathcal{C}' \lhd (\mathcal{R}_P \cup \mathcal{R}_Q)$,
$\mathcal{I}' = \mathrm{ran}\mathcal{R}'$ and $R^{a\to b}_{IO} = \{a.out.x \mapsto b.in.x\}$.

In this composition, the channels c and z are combined such that output events from one channel are consumed by input events of the other, and vice versa. This correspondence is made by two mapping relations, $R^{c\to z}_{IO}$ and $R^{z\to c}_{IO}$, which are used to input/output from the buffer $BUFF^n_{IO}$. The resulting component behaviour is that of P synchronised with the buffer $BUFF^n_{IO}(R^{c\to z}_{IO}, R^{z\to c}_{IO})$ on c and with Q on z. The interface, $\mathcal{C}'$, of the resulting component, $P_{\langle c\rangle} \approx_{\langle z\rangle} Q$, contains channels of both P and Q except for the hooked channels c and z ($(\mathcal{C}_P \cup \mathcal{C}_Q)\backslash\{c, z\}$). Therefore, only channels in $\mathcal{C}'$ appear in the resulting relation $\mathcal{R}'$ ($S \lhd R$ restricts the domain of R to S) and in the resulting interface $\mathcal{I}'$ ($\mathrm{ran}\mathcal{R}'$).

There are more complex systems that present cycles of dependencies in the topology of their structure. The next two compositions, depicted together in Fig. 1, allow the link of two channels of a same entity without introducing deadlock (it is proved by Theorem 1 from Ramos et al., 2009; Ramos, 2011). The feedback composition can only be used to assemble channels that are decoupled: ch_1 and ch_2 are decoupled if the interleaving of the projected component behaviour on ch_1 and on ch_2 is equivalent to the projected behaviour of the component on the set formed of these two channels. This means that there is no interference between these two channels (see Appendix B for a formal account). Reflexive composition is more general than the feedback rule, as it does not require channels to be decoupled; however, it is also more costly regarding verification, since, in general, it requires a global analysis to ensure deadlock freedom. These rules ensure that systems developed in $\mathcal{BRIC}$ are deadlock-free. Theorem 1 is proved in Ramos et al. (2009).

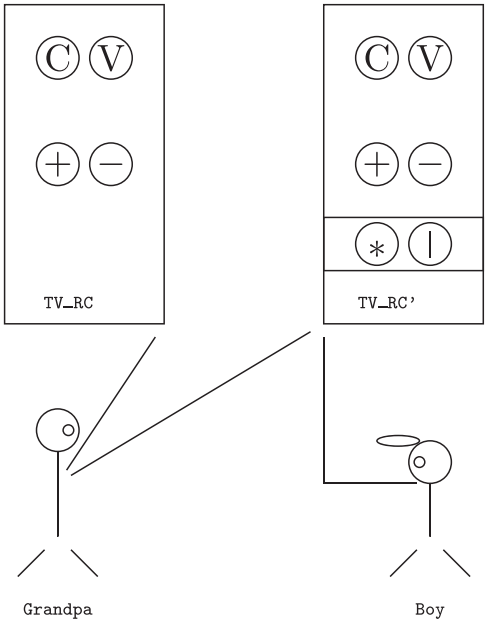


Fig. 2. TV remote control extension.

Theorem 1 (Deadlock-free Component Systems)**.** *Consider P a deadlock-free component and c_1 and c_2 channels. Any system S in normal form, as defined below, built from deadlock-free components, is deadlock-free.*

$$S ::= P \mid S\ [|||]\ S \text{ (interleave)} \mid S[c_1 \leftrightarrow c_2]S \text{ (communication)}$$
$$\mid S[c_1 \hookrightarrow c_2] \text{ (feedback)} \mid S[c_1 \hookrightarrow c_2] \text{ (reflexive)}$$

3. $\mathcal{BRIC}$ extensibility

In this section we present the main contributions of this work: the development of inheritance relations for behavioural specifications that distinguish inputs from outputs, in the context of rigorous trustworthy component development, where structural aspects are also considered. Our relations rely on a behavioural relation called *convergence*. It captures the idea that components can evolve by accepting new inputs or establishing a communication session after these inputs but the components are able to *converge* to the behaviour exhibited by their abstractions. This is a concept that cannot be captured only by the hiding operator as in the case of other inheritance relations Wehrheim (2003). The reason is that, when hiding an event, it is removed from the traces a behaviour exhibits but, in our inheritance relations, an event can have different meanings based on the context it appears and, in some of these contexts we want to hide them, in others we do not. This is exemplified by our motivating example.

Additionally, we propose a denotational semantics and a refinement notion for $\mathcal{BRIC}$ components, which, alongside the proposed inheritance notions, make it a fully formal approach to CB-MDD. In the sequel we present a motivating example that illustrates the purpose and intuition behind this new behavioural relation.

3.1. Motivating example

Consider a TV remote control that offers the options for controlling the TV audio volume and switching between channels. It is represented by the device `TV_RC` in Fig. 2; the user presses the button ©, then he can go forward or backward on the channel list by pressing ⊕ or ⊖, respectively. The same applies when `Grandpa` wants to increase/decrease the TV volume by pressing v◯. The user `Boy` wants to do more by having the option to adjust the TV brightness and contrast by pressing the buttons ⊛ and |◯, respectively. The user `Boy` is aware of a lid on the device `TV_RC'` (Fig. 2), by which he can access these new functionalities, which are unknown by `Grandpa`.

We specify the behaviours of `TV_RC` and `TV_RC'` in the LTSs (Labelled Transition System) depicted in Figs. 3 and 4, respectively. The process `TV_RC(c,v)` represents a state where channel `c` and volume `v` are the last values sent to the TV. The user provides the input event `vol` to increase (`up`) or decrease (`down`) the TV volume or `ch` to navigate forward (`up`) or backward (`down`) on TV channels. Status output events (`st`) acknowledge the user of its commands. Updates in the volume or channel are captured by modular arithmetic; for instance, `(c+1)%Lc` is addition modulo a maximum allowed value Lc for channels. Modular arithmetic avoids results outside the range between zero and the maximum value Lc.

The process `TV_RC'(c,v,b,cn)` in Fig. 4 extends `TV_RC(c,v)` by adding the new input event `sett`, which occurs when the user opens the remote control lid; it gives access to the TV settings menu, in which the user can adjust brightness (`brig`) or contrast (`cont`), whose values are registered by the state variables `b` and `cn`, respectively. After providing `brig` or `cont` as input, it can use the regular input events `up` and `down` to make the image adjustments, which are echoed by output events through the channel `st'`.

If we consider `TV_RC'(c,v,b,cn)` as a valid extension to `TV_RC(c,v)` (in the sense that it preserves convergence) we must decide which relation captures this kind of extension. Such a relation must allow new-in-context input events (those that are not necessarily new in the process alphabet, but that are not

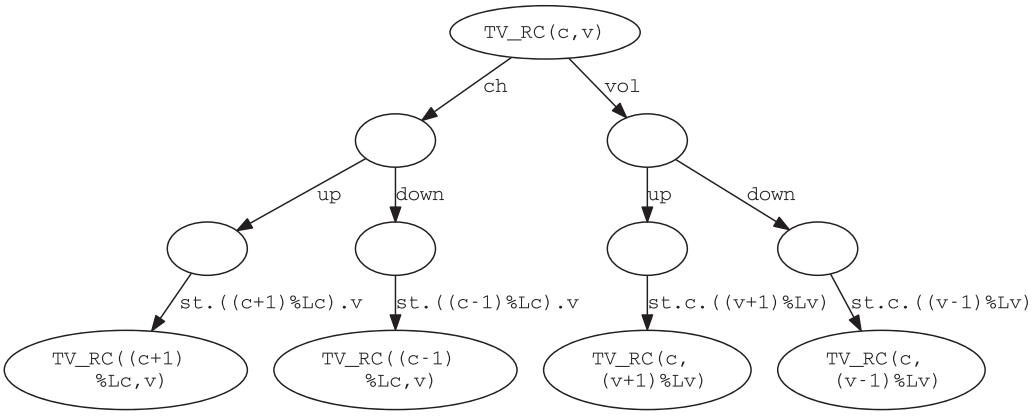


Fig. 3. Labelled transition system of the `TV_RC(c,v)` process.

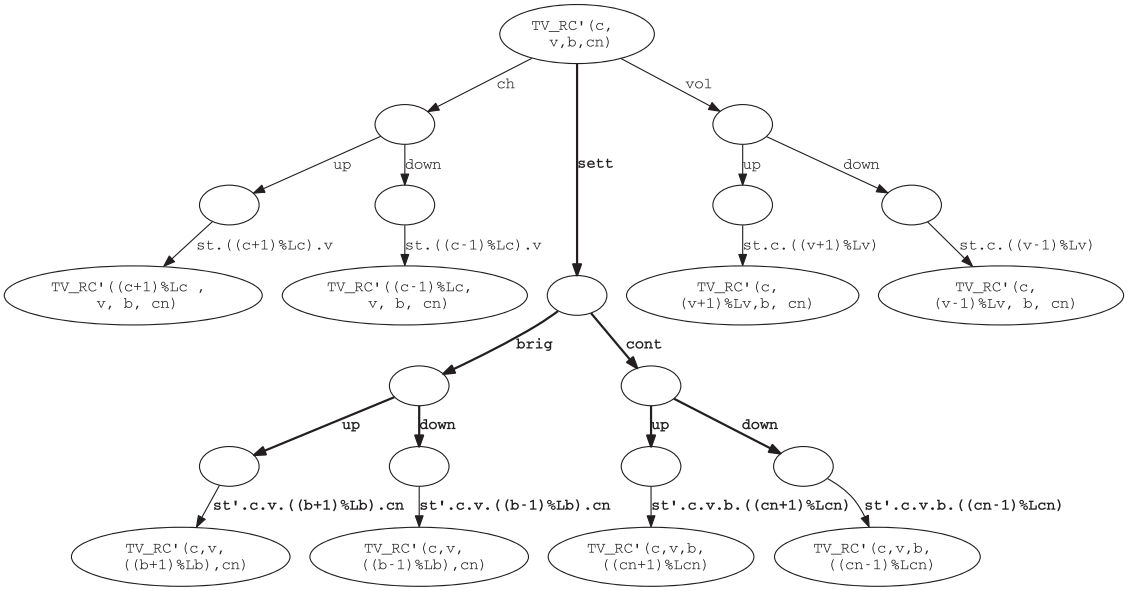


Fig. 4. Labelled transition system of the `TV_RC'(c,v)` process.

among the events offered by the process in a particular context), as `sett`, followed by a finite number of new-in-context input/output events, as `brig`, `up`, and `st'`. Furthermore, the extension must converge (offering what was expected before the new-in-context event happened) to the original behaviour. For instance, this is what `TV_RC'(c,v,((b+1))%Lb),cn)` does after the trace <`sett, brig, up`>, see Figure 4: it offers ch and vol, with the variables c and v unchanged by what happened after sett. In summary, `Grandpa` can share the new TV remote control with Boy, without being stuck with, or even perceiving, the new features.

One can recognise the CSP failures refinement as a candidate to capture the relationship between `TV_RC` and `TV_RC'`, provided the new events are hidden in the extension, but a closer investigation shows this is not the case. This happens because the events `up` and `down` are also used in new contexts in `TV_RC'` to adjust brightness and contrast. This kind of relationship in which we use existing events in a different context cannot be captured by failures refinement.

In Wehrheim (2003) four subtyping relations are defined for behaviour specifications, which allow functionality extension. The first issue with these candidate relations is that they do not differentiate inputs from outputs (as further discussed in the next section) and, moreover, extensions can only be defined in terms of: new events, which can be concealed (they may not be communicated but are not made internal Wehrheim, 2003), hidden (made internal), explained (in terms of existing events) or restricted (completely forbids them). Except for hiding, the other extensions are not directly supported by CSP, and none of them can capture the intended relationship between `TV_RC` and `TV_RC'`.

The ioco (input-output conformance) relation van der Bijl et al. (2004) allows extensions to admit new inputs (more functionalities) and to restrict outputs (more deterministic), but it does not obligate extensions, after a new input, to adhere (converge) to the original behaviour. Taking our example into account, ioco would admit `TV_RC'` to engage in `sett` and then behave as `STOP` (or anything else, including a divergent process). Therefore, the user `Boy` would not be able to navigate on the TV channel list, if he tried to adjust the image settings. Although ioco is a relation adopted in the context of conformance testing, whereas we are concerned with model evolution, we considered it here because it also allows extension of functionality, but in a more restrictive manner than we need.

This discussion highlights the fact that the current behaviour relations cannot cope with functionality extension in a scenario where we need to add new events, to use existing events in different contexts and to distinguish input from output events.

Before formalising convergence we need to say that `TV_RC'` process, and inheritance behaviours in general, can be achieved by a variety of mechanisms, including design patters. Nevertheless, this work does not address the mechanisms to achieve convergent behaviours. In fact, our focus is on the definition of convergence and how it can be mechanically verified to ensure deadlock freedom in the evolution of behaviour component specifications.

3.2. Convergence

Inheritance in the object-oriented paradigm is a well known concept with a comprehensive literature Liskov and Wing (1994). More recently, efforts have been made to extend this concept to process algebras as CSP. Notably, in Wehrheim (2003) the author proposes four types of behavioural inheritance relations for Labelled Transition Systems (LTS). Although very promising, as already discussed, these relations do not consider specifications that distinguish inputs from outputs, as required in $\mathcal{BRIC}$, in which a component behaviour is modelled as a CSP I/O process. Since I/O processes must satisfy behaviour restrictions such as input determinism and strong output decisiveness, we need relations that can capture these restrictions. We base our approach on a concept of *convergence*: a convergent process is allowed to do the same as or more inputs than its parent process, but is restricted to do the same or less outputs in convergent points. A convergent point represents a state reachable by both the original and the convergent process when doing two convergent sequences of events; these sequences differ only because the convergent process is allowed to do extra inputs (inputs not allowed by the original process) in converging points. First, we formalise the concept of convergent traces as follows.

In Definition 5 and others that follow, Σ stands for the alphabet of all possible events, Σ^* is the set of possible sequences of events from Σ, the input events are contained in Σ ($inputs \subseteq \Sigma$) and $in(T, t)$ is a function that yields the set of input events that can be communicated by the I/O process T after some trace $t \in \mathcal{T}(T)$; there-

fore *in* has type $I/OProcess \times \Sigma^* \rightarrow \mathcal{P}(inputs)$, where $\mathcal{P}$ stands for the powerset. Additionally $t_1 \leq t_2$ means that t_1 is a prefix of t_2.

A trace t' (of a process T') is I/O convergent to a trace t (of a process T) if they are equal or if $t = t_1 \hat{} t_3$ and $t' = t_1 \hat{} \langle ne \rangle \hat{} t'_3$ such that $ne \in (T', t_1)$ but $ne \notin in(T, t_1)$ and t is I/O convergent to $t_1 \hat{} t'_3$. This recursive definition means that t' and t might differ because T' can do new-in-context inputs (inputs not allowed by T) where T cannot, but, in spite of that, the trace t' of T' has always a counterpart t of T. In other words, a trace t' is I/O convergent to a trace t if they differ only by certain inputs allowed by T' but not by T. Also, because the CSP hiding operator renames visible events (events that can appear in traces) to the invisible event τ, it is not an alternative to define convergence, where events have different meanings depending on the context they are communicated.

Definition 5 (I/O convergent traces)**.** Consider an I/O process T. Let t and t' be two traces, such that $t \in traces(T)$. We say that t' is an I/O convergent trace of t (t' `cvg` t) if, and only if:

$$(t' = t) \vee \left(\begin{array}{c} (\#t' > \#t) \ \wedge \ \exists t_1, t_3 : \Sigma^*, \exists ne : \Sigma \mid \\ \left(\begin{array}{c} t' = t_1 \hat{} \langle ne \rangle \hat{} t_3 \ \wedge \ t_1 \leq t \wedge \\ ne \in inputs \ \wedge ne \notin in(T, t_1) \wedge \\ t_1 \hat{} t_3 \ \texttt{cvg} \ t \end{array} \right) \end{array} \right)$$

Based on the definition of convergent traces, we are now able to define behavioural convergence.

Definition 6 (I/O convergent behaviour)**.** Consider two I/O process T and T'. T' is an I/O convergent behaviour of T (T' `io_cvg` T) if, and only if:

$$\forall (t', X) \in \mathcal{F}(T'), \exists (t, Y) \in \mathcal{F}(T) \bullet \left(\begin{array}{c} t' \ \texttt{cvg} \ t \wedge \\ Y \cap inputs \supseteq X \cap inputs \wedge \\ Y \cap outputs \subseteq X \cap outputs \end{array} \right)$$

An I/O process T' is convergent to T if, for any trace $t' \in \mathcal{T}(T')$, there exists a trace $t \in \mathcal{T}(T)$, such that t' `cvg` t, and T' after t' can offer more or equal inputs ($Y \cap inputs \supseteq X \cap inputs$) but is restricted to offer less or equal outputs ($Y \cap outputs \subseteq X \cap outputs$) when compared with T after t.

Convergence is a more restrictive relation than those based solely on covariation of inputs and contravariation of outputs (such as ioco van der Bijl et al., 2004), because it requires, after a new-in-context event, the process to converge to its parent, which allows extensions but ensures substitutability as we discuss later. Let us consider the I/O processes T (Listing 1) and T' (Listing 2), whose LTSs are depicted in Figs. 5a and b, respectively. Recall that the suffixes `in` and `out` are used to mark input and output events, respectively.

Based on Definition 6, we have that T' `io_cvg` T. To explain why this is the case, let (t', X) and (t, Y) be failures of T' and T, respectively. Then, by a non-exhaustive analysis, where $\{|c|\}$ stands for all events that can be communicated through the channel c:

- let $(t', X) = (\langle c.in.v.3, c.in.v.1, c.in.v.3 \rangle, \{|c|\} \setminus \{c.out.v.1\})$, which means that after trace t', T' can only communicate $c.out.v.1$, rejecting everything else (i.e., $\{|c|\} \setminus \{c.out.v.1\}$). Considering that $(t, Y) = (\langle c.in.v.1 \rangle, \{|c|\} \setminus \{c.out.v.1\})$ is a failure of T, we have t' `cvg` t as $c.in.v.3$ is a new-in-context input of T in both states 0 and 3 of its LTS (see Fig. 5a); also $Y \cap inputs = X \cap inputs = \{|c.in|\}$ and $Y \cap outputs = X \cap outputs = \{|c.out|\} \setminus \{c.out.v.1\}$;
- let $(t', X) = (\langle c.in.v.1 \rangle, \{|c|\} \setminus \{c.in.v.2, c.in.v.3\})$, which means that after trace t', T' can only communicate $c.in.v.2$ or $c.in.v.3$ rejecting everything else. Considering that $(t, Y) = (\langle c.in.\ v.1 \rangle, \{|c|\} \setminus \{c.out.v.2\})$ is a failure of T, we have t' `cvg` t as equal traces are also convergent by definition; also $X \cap inputs = \{|c.in|\} \setminus \{c.in.v.2, c.in.v.3\}$, $X \cap outputs = \{|c.out|\}$, $Y \cap inputs = \{|c.in|\}$ and $Y \cap outputs = \{|c.out|\} \setminus \{c.out.v.2\}$.

A convergent I/O process can engage in more inputs so that, when converging, it can take more deterministic decisions on what to output. Nevertheless, it can be useful to offer other events after a new input and before converging to its original behaviour according to the relation. This extension to convergence allows convergent processes to add more implementation details. We define this relation in the traces and failures behavioural models in Definitions 7 and 8, respectively.

Definition 7 (I/O extended convergent traces)**.** Consider two I/O processes T and T'. Let t and t' be two of their traces, respectively. We say that t' is an I/O extended convergent trace of t (t' `ecvg` t) if and only if:

$$(t' = t) \vee \left(\begin{array}{c} (\#t' > \#t) \ \wedge \ \exists t_1, t_2, t_3 : \Sigma^*, \exists ne \in \Sigma \mid \\ \left(\begin{array}{c} t' = t_1 \hat{} \langle ne \rangle \hat{} t_2 \hat{} t_3 \ \wedge t_1 \leq t \wedge \\ ne \in inputs \ \wedge ne \notin in(T, t_1) \wedge \\ set(t_2) \cap (in(T, t_1) \cup out(T, t_1)) = \emptyset \wedge \\ t_1 \hat{} t_3 \ \texttt{ecvg} \ t \end{array} \right) \end{array} \right)$$

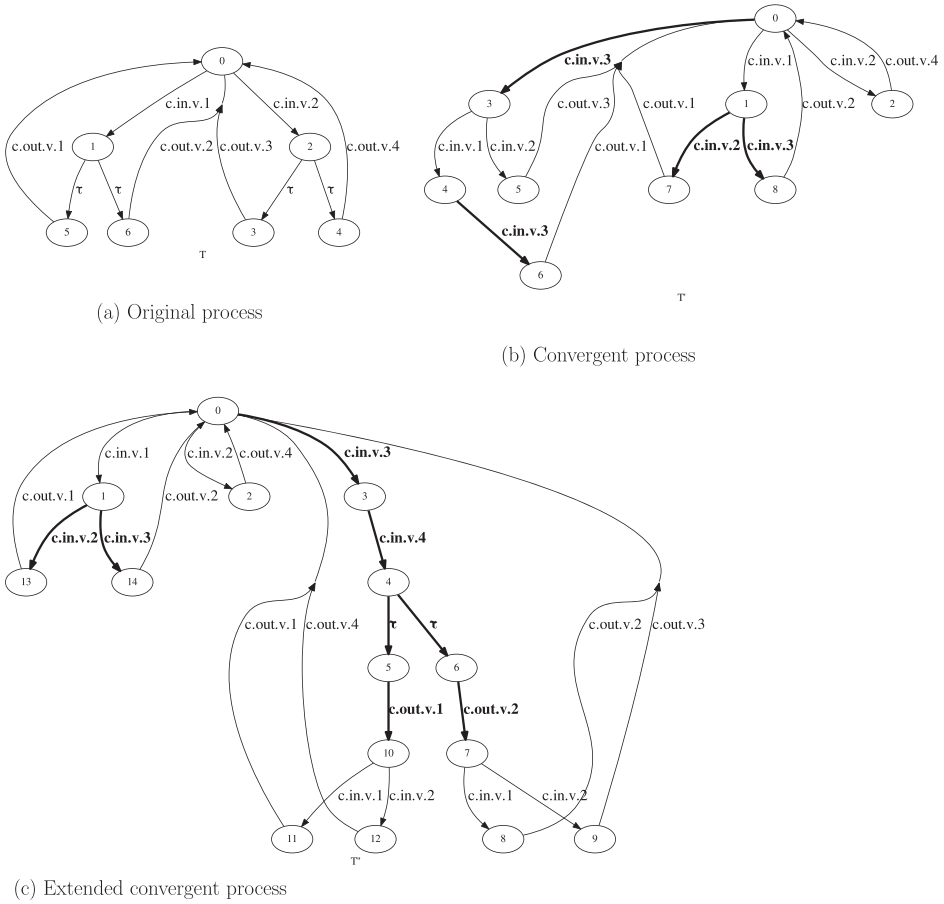
A trace t' is I/O extended convergent to t if they are the same or if it is possible to equate them by concealing each event ne that is offered by T' but not for T after a common subtrace, say t_1, of t and

```
T = c.in.v.1 -> (c.out.v.1 -> T |~| c.out.v.2 -> T)
    []
    c.in.v.2 -> (c.out.v.3 -> T |~| c.out.v.4 -> T)
```

Listing 1. I/O process T.

```
T' = c.in.v.1 -> (c.in.v.2 -> c.out.v.1 -> T'
                  []
                  c.in.v.3 -> c.out.v.2 -> T')
    []
    c.in.v.2 ->  c.out.v.4 -> T'
    []
    c.in.v.3 -> (c.in.v.1 ->  c.in.v.3 -> c.out.v.1 -> T'
                 [] c.in.v.2 -> c.out.v.3  -> T')
```

Listing 2. I/O process T'(version 1).



(a) Original process

(b) Convergent process

(c) Extended convergent process

Fig. 5. I/O convergent behaviours.

t' ($ne \notin in(T, t_1)$, but $ne \in in(T', t_1)$); furthermore, since we allow more events after a new input ne, we also conceal them ($set(t_2) \cap (in(T, t_1) \cup out(T, t_1)) = \emptyset$).

Definition 8 (I/O extended convergent behaviour)**.** Consider two I/O processes T and T'. We say that T' is an I/O extended convergent behaviour of T (T' `io_ecvg` T), if and only if:

$$\forall (t', X) \in \mathcal{F}(T'), \exists (t, Y) \in \mathcal{F}(T)$$
$$\bullet t' \ \texttt{ecvg} \ t \wedge \left(\begin{pmatrix} Y \cap inputs \supseteq X \cap inputs \wedge \\ Y \cap outputs \subseteq X \cap outputs \end{pmatrix} \vee (\Sigma \backslash Y \subseteq X) \right)$$

Definition 8 is very similar to Definition 6, but allows the extended convergent process T' to accept any event not expected by T ($\Sigma \backslash Y \subseteq X$) in an extended convergent point of their execution (t' `ecvg` t), provided a new input ne (see Definition 7) has happened, marking the start of the extended convergent behaviour of T'.

We illustrate this definition with an example. Let us consider the processes T (Listing 1) and T'' (Listing 3), whose LTSs are depicted in Fig. 5a and c, respectively. By Definition 8, we have that T'' `io_ecvg` T. To gather some evidence that this is the case,

```
T' = c.in.v.1 -> ( c.in.v.2 -> c.out.v.1 -> T'
                 [] c.in.v.3 -> c.out.v.2 -> T')
     []
     c.in.v.2 ->  c.out.v.4 -> T'
     []
     c.in.v.3 -> c.in.v.4 -> (c.out.v.1 ->
                                (c.in.v.1 -> c.out.v.1 -> T'
                                [] c.in.v.2 -> c.out.v.4 -> T')
                              |~|
                              c.out.v.2 ->
                                (c.in.v.1 -> c.out.v.2 -> T'
                                [] c.in.v.2 -> c.out.v.3 -> T'))
```

Listing 3. I/O process T'(version 2).

we analyse a couple of failures of these processes. Assume that $(t', X) \in \mathcal{F}(T'')$ and $(t, Y) \in \mathcal{F}(T)$, then:

- considering $(t', X) = (\langle c.in.v.3, c.in.v.4\rangle, \{|c|\} \setminus \{c.out.v.1\})$, we can find $(t, Y) = (\langle\rangle, \{|c|\} \setminus \{c.in.v.1, c.in.v.2\})$, such that t' ecvg t, $\Sigma \backslash Y = \{c.in.v.1, c.in.v.2\}$ and, therefore, $\Sigma \backslash Y \subseteq X$;
- if $(t', X) = (\langle c.in.v.1, c.in.v.3\rangle, \{|c|\} \setminus \{c.out.v.2\})$, we can find the failure $(t, Y) = (\langle c.in.v.1\rangle, \{|c|\} \setminus \{c.out.v.2\})$, such that t' ecvg t, $X = Y$ and, therefore, Definition 8 holds;
- finally, assuming $(t', X) = (\langle c.in.v.2\rangle, \{|c|\} \setminus \{c.out.v.4\})$, we can find the failure $(t, Y) = (\langle c.in.v.2\rangle, \{|c|\} \setminus \{c.out.v.4\})$ and Definition 8 trivially holds.

As one might expect, extended convergence is a generalisation of convergence, which comes from Lemmas 1, 2 and 3 proved in Appendix C. Consider two traces with a common prefix; the first lemma ensures that if one of these traces is convergent to the other, starting from that common prefix trace, it is also extended convergent.

Lemma 1 (*cvg* implies *ecvg* on trace prefixing)**.** *Consider two I/O proc– esses T and T' such that, $t_1 \frown t_3 \in \mathcal{T}(T)$ and $t' \in \mathcal{T}(T')$. If t' cvg $t_1 \frown t_3$, where $t_1 \preceq t'$, then t' ecvg $t_1 \frown t_3$.*

Lemma 2 formalises extended convergence as a generalisation of convergence; therefore, if two traces are convergent they are also extended convergent. Lemma 3 proves the same for I/O processes.

Lemma 2 (cvg $\subseteq$ ecvg)**.** *Consider two I/O processes T and T', and t and t' such that $t \in \mathcal{T}(T)$ and $t' \in \mathcal{T}(T')$. If t' cvg t then t' ecvg t.*

Lemma 3 (io_cvg $\subseteq$ io_ecvg)**.** *Consider two I/O processes T and T'. If we have T' io_cvg T then T' io_ecvg T.*

3.3. Extensibility

Our definition of inheritance deals with component structural and behavioral aspects. Structurally, it guarantees that the inheriting component preserves at least its parent's channels and their types: if T' extends T we have that $\mathcal{R}_T \subseteq \mathcal{R}_{T'}$. Regarding behaviour, they are related by convergence. Additionally, it guarantees, for substitutability purposes, that the inherited component T' refines the protocols exhibited by common channels (default channel congruence, as in Definition 9) or that additional inputs (new in context, see Definitions 6 and 8) over common channels are not exercised by any possible client of its parent T (input channel congruence, Definition 10).

The $\mathcal{BRIC}$ component model restricts how components can be assembled to avoid deadlock. Channel congruence aims at paving a safe way to extend a specification by using convergence without introducing deadlock; it does not reduce possible inputs, but disciplines the way in which existing inputs can be used in convergent extensions. The simplest, but restrictive, form of achieving this is by guaranteeing that the protocol over a channel must be refined, in the failures classical sense (Definition 9).

Definition 9 (Default channel congruence)**.** An I/O process T_{dc} has a default congruent channel, say c, to another I/O process T (T_{dc} **def-cong**(c) T), when there is a failures refinement relation between their projections over c:

$$\mathcal{F}(T_{dc} \backslash (\Sigma \setminus \{c\})) \subseteq \mathcal{F}(T \backslash (\Sigma \setminus \{c\})),$$
which is equivalent to $\mathcal{F}(T_{dc} \upharpoonright \{c\}) \subseteq \mathcal{F}(T \upharpoonright \{c\})$

A more flexible way is given by Definition 10: I/O processes T_{ic} and T are input congruent on an I/O channel c if, after both have done the same trace t, either of the following holds:

i. if T cannot engage in any input, then T_{ic} cannot input on c: it avoids T's clients from deadlocking when interacting with T_{ic}, since these clients do not communicate (after the trace t) outputs on c to T, otherwise they deadlock; therefore T_{ic}, as T, cannot expect inputs on c after the trace t;
ii. if T is not able to input on c, T_{ic} can only do it for events outside T's alphabet: it avoids T's clients to engage in a possible unexpected communication over c, which can (but not necessarily) lead to deadlock. It is worth saying that in places where T can input over c, T_{ic} can also input new-in-context events over c; it is possible because they are offered in external choice, so T's clients will not be able to communicate these new-in-context inputs offered by T_{ic}. This happens because T's clients are not ready to engage in these new-in-context inputs; otherwise, this would mean that their composition with T deadlocks, because T is not able to offer such new-in-context inputs.

An I/O process T_{ic} has an input congruent channel c to an I/O process T (T_{ic} **inp-cong**(c) T), if after a common trace t, such that $(t, X) \in \mathcal{F}(T_{ic})$ and $(t, Y) \in \mathcal{F}(T)$, the following holds: T refuses to input ($inputs \subseteq Y$) and T_{ic} refuses to input over c ($\{|c.in|\} \subset X$) or T refuses to input over c ($\{|c.in|\} \subseteq Y$) and T_{ic} refuses the events on c that can be communicated by T ($\{|c.in|\} \setminus (\alpha T_{ic} \setminus \alpha T) \subset X$), where αT_{ic} and αT stand for the alphabets of T_{ic} and T, respectively. The formal definitions is as follows.

Definition 10 (Input channel congruence)**.** Given two I/O processes T_{ic} and T, and a channel c, we say that T_{ic} **inp-cong**(c) T if, and only if:

$$\forall (t, X) \in \mathcal{F}(T_{ic}) \bullet \exists (t, Y) \in \mathcal{F}(T) \Rightarrow \begin{pmatrix} inputs \subseteq Y \Rightarrow \{|c.in|\} \subset X \\ \vee \\ \{|c.in|\} \subseteq Y \Rightarrow \{|c.in|\} \setminus (\alpha T_{ic} \setminus \alpha T) \subset X \end{pmatrix}$$

provided

$$(\exists (t, X') \in \mathcal{F}(T_{ic}) \Rightarrow X' \subseteq X) \wedge (\exists (t, Y') \in \mathcal{F}(T) \Rightarrow Y' \subseteq Y)$$

In the definition above, the proviso guarantees that (t, X) and (t, Y) are maximal failures.

The following is the most important definition of this work, the component inheritance relation, which allows behaviour extension and, moreover, guarantees substitutability.

Definition 11 ($\mathcal{BRIC}$ inheritance)**.** Consider T and T' two $\mathcal{BRIC}$ components, such that $\mathcal{R}_T \subseteq \mathcal{R}_{T'}$. We say that T' inherits from T:

- by convergence: $T \leftarrow_{cvg} T' \iff \mathcal{B}_{T'}$ io_cvg $\mathcal{B}_T$
- by extended convergence: $T \leftarrow_{ecvg} T' \iff \mathcal{B}_{T'}$ io_ecvg $\mathcal{B}_T$

provided

$$\forall c : \mathcal{C}_T \bullet (\mathcal{B}_{T'} \textbf{ def-cong}(c) \ \mathcal{B}_T) \vee (\mathcal{B}_{T'} \textbf{ inp-cong}(c) \ \mathcal{B}_T)$$

The provided clause guarantees that, when interacting with T', a component originally designed to interact with T will not engage in T' extensions trigged by inputs also used by T, which in T' have a different meaning.

3.4. Semantics and refinement

In this section, we contribute with a denotational semantics and a refinement relation for $\mathcal{BRIC}$. Furthermore, we show that the refinement and inheritance relations form a hierarchy. We also prove that our relations preserve deadlock freedom and moreover, that they respect the substitutability principle. We start by defining a function $\mathcal{S}[\![\cdot]\!]$ from a $\mathcal{BRIC}$ component to an underpinning mathematical model. Consider the component $T : \langle \mathcal{B}, \mathcal{R}, \mathcal{I}, \mathcal{C} \rangle$, the semantics of T is given by:

$$\mathcal{S}[\![T]\!]$$

$= (failures(\mathcal{B}), \{(c, failures(\mathcal{B} \upharpoonright \{c.i\})) \mid c \in \mathcal{C} \ \wedge i \in \mathcal{I} \ \wedge (c, i) \in \mathcal{R}\})$

This semantics captures the relevant properties of a $\mathcal{BRIC}$ component: its overall behaviour (given by the failures of the I/O process $\mathcal{B}$ that defines the component behaviour) and those exhibited through its channels (a set of pairs where each channel c maps into the failures of the overall behaviour projected to c); such projected behaviours are crucial in composition rules. Whenever the type of a particular channel c is known we can simplify its semantics to $\mathcal{S}[[T]] = (failures(\mathcal{B}), \{(c, failures(\mathcal{B} \upharpoonright c.i)) \mid c \in \mathcal{C}\})$. It is important to note that we are not presenting $\mathcal{S}[[\cdot]]$ in a compositional manner, by induction on the process structure; rather, the more concise and simpler presentation is sufficient in this work. With a component semantics we can define a refinement notion (monotonic with respect to the $\mathcal{BRIC}$ composition rules Oliveira, M. and Sampaio, A. and Antonino,P. and Dihego, J. and Filho, M. C. and Bryans, J., 2014).

Definition 12 ($\mathcal{BRIC}$ refinement)**.** Consider two components T and T'. If we consider $\mathcal{S}[[T]] = (f, fp)$ and $\mathcal{S}[[T']] = (f', fp')$, with f and f' standing for the overall behaviour of T and T', and fp and and fp' for the projected behaviours on their channels, respectively, then we say that T' refines T ($T \sqsubseteq_{\mathcal{B}} T'$) if, and only if:

$$(f' \subseteq f) \ \wedge (\text{domfp} = \text{domfp}') \ \wedge (\forall c \in \text{domfp} \bullet \text{fp}'(c) \subseteq \text{fp}(c))$$

This definition ensures that T and T' have the same interaction points. Moreover, it guarantees that the component behaviour of T' refines that of T, which is equivalent to:

$$(\mathcal{B}_T \sqsubseteq_{\text{F}} \mathcal{B}_{T'}) \ \wedge (\mathcal{C}_T = \mathcal{C}_{T'}) \ \wedge (\forall c : \mathcal{C}_T \bullet \mathcal{R}_T(c) \subseteq \mathcal{R}_{T'}(c))$$

Theorem 2 (proved in Appendix C) states that the refinement and inheritance relations form a hierarchy. Component refinement is the strongest relation between $\mathcal{BRIC}$ components; it implies inheritance. We can see refinement as the strongest form of inheritance. As expected, inheritance by convergence is a more strict form of inheritance than extended convergence, as their names suggest.

Theorem 2 (Hierarchy)**.** *The relations $\sqsubseteq_{\mathcal{B}}$, $\twoheadleftarrow_{cvg}$ and $\twoheadleftarrow_{ecvg}$ form a hierarchy: $\sqsubseteq_{\mathcal{B}} \ \subseteq \ \twoheadleftarrow_{cvg} \ \subseteq \ \twoheadleftarrow_{ecvg}$.*

This concludes an important result that relates refinement and the notions of inheritance based on (extended) convergence.

3.4.1. Substitutability

We prove, in Lemma 4, that $\mathcal{BRIC}$ inheritance preserves deadlock freedom. A more interesting result, proved by Theorem 3, guarantees that a component T' can replace T, in any context produced by the $\mathcal{BRIC}$ composition rules, without introducing deadlock, provided T' inherits by convergence from T. These results are proved in Appendix C.

Lemma 4 (Inheritance preserves deadlock freedom)**.** *Consider T and T' two $\mathcal{BRIC}$ components, such that T is deadlock free. If $T \twoheadleftarrow_{ecvg} T'$ then T' is deadlock free.*

Theorem 3 (Substitutability)**.** *Let T and T' be two components such that $T \twoheadleftarrow_{ecvg} T'$. Consider $S[T]$ a deadlock free component contract that includes, as part of its behaviour, a deadlock free component contract T; then $S[T']$, which stands for S with T replaced with T', is deadlock free.*

Note that this result also holds for the other two relations ($\sqsubseteq_{\mathcal{B}}$ and $\twoheadleftarrow_{cvg}$), as a consequence of the hierarchy established by Theorem 2.

4. Checking convergence

Behaviour convergence and the relations built on top of it are the backbone of this work; therefore, we must have an automated strategy to check whether two I/O processes are related by convergence. We start addressing this issue by choosing FDR4 (Failures-Divergence Refinement) Gibson-Robinson et al. (2014) as the model-checker to carry out the analysis; it seems a natural choice given the widespread use of FDR4 both in academy and industry, which makes it a *de facto* standard tool for analysing CSP specifications. Its method of establishing whether a property holds is to check for the refinement between CSP specifications, internally represented by labelled transition systems.

To check conformance of I/O Processes, say P and P', our strategy is to construct, for each relation, a verification strategy of the form:

$$P' \ \texttt{io\_cvg} \ P \iff GLB\_CVG(P) \sqsubseteq_{\text{F}} P'$$
$$P' \ \texttt{io\_ecvg} \ P \iff GLB\_ECVG(P) \sqsubseteq_{\text{F}} P'$$

To explain how these parametrised *GLB_* processes can be constructed, we need to present some additional background on failures refinement, convergence and I/O process alternative representations. Consider P an I/O process, then $cvg^{+}P$ stands for a set of I/O processes such that: $\forall P' \in cvg^{+}P \bullet P' \ \texttt{io\_cvg} \ P$; it contains every I/O process convergent to P, including P itself. The set $cvg^{+}P$ is infinite, which makes its use prohibitive for any implementation that aims to traverse it. A finite subset of $cvg^{+}P$ is given by $cvg^{+n}P$, which stands for the P convergent processes whose depth differ from that of P by at most n. An I/O process depth is given by the longest trace after which the process returns (for the first time) to its initial state or, by considering its LTS, the maximum number of transitions (labelled with visible events) from the initial state to itself. An I/O process depth can be equivalently expressed in two ways: (a) in terms of traces and failures-semantics or (b) based on its LTS representation, where $P \stackrel{\Rightarrow}{t} P$ means that the I/O process P returns to its initial state after the trace t:

(a) $depth(P) = max\{\#t \mid t \in \mathcal{T}(P) \ \wedge P \equiv_F P/t \ \wedge (\nexists s < t \mid P \equiv_F P/s)\}$

(b) $depth(P) = max\{\#t \mid P \stackrel{t}{\Rightarrow} P \ \wedge (\nexists s < t \mid P \stackrel{s}{\Rightarrow} P)\}$

For example, the depths of the processes in Fig. 5a, b and c are, respectively, 2, 4 and 5. The core of our strategy is to build a CSP process *GLB*, such that it belongs to $cvg^{+n}P$ and every member Q of $cvg^{+n}P$ refines it, $GLB \sqsubseteq_{\text{F}} Q$. Furthermore, if there is any other process, say R, which satisfies this property, then $R \sqsubseteq_{\text{F}} GLB$. It means that the process *GLB* is the *Greatest Lower Bound* (Enderton, 1977; Roscoe, 1998) of the set $cvg^{+n}P$ under the CSP failures refinement relation ($\sqsubseteq_{\text{F}}$). Our intention is to construct $GLB\_CVG(P)$ to be failures equivalent to *GLB*, $GLB\_CVG(P) \equiv_F GLB$. Therefore, to verify if a process P' is convergent to P, i.e., if P' belongs to $cvg^{+n}P$, one needs only to verify if $GLB\_CVG(P) \sqsubseteq_{\text{F}} P'$. The same reasoning applies to the extended convergence relation. The next section details how we construct $GLB\_CVG(P)$ and $GLB\_ECVG(P)$ for an I/O process P.

4.1. Building GLB_CVG

Let P and P' be I/O processes that differ in depth by n. To test whether $P' \ \texttt{io\_cvg} \ P$, we must build from P a new process $GLB\_CVG(P)$, which must be able to do *at most n* new-in-context inputs in every state of P. Such a process is, by Definition 6, convergent to P. As we stated before, any process convergent to P (which differs in depth by at most n) must refine in failures $GLB\_CVG(P)$. An important practical question to build $GLB\_CVG(P)$ is to compute the new-in-context inputs for any state of P. Given a state of

```
1  P_serial = <(start, <c.in.v.1, c.in.v.2>, 0),
2              (c.in.v.1, <c.out.v.1, c.out.v.2>, 1),
3              (c.out.v.1, <end>, 2),
4              (end, <>, 3),
5              (c.out.v.2, <end>, 2),
6              (end,<>, 3),
7              (c.in.v.2, <c.out.v.3, c.out.v.4>, 1),
8              (c.out.v.3, <end>, 2),
9              (end, <>, 3),
10             (c.out.v.4, <end>,2),
11             (end, <>, 3)>
```

Listing 4. Serialisation.

P, the easiest way of computing the new-in-context inputs available is to know which inputs can be accepted in this state; the result will be its complement. The problem is that CSP does not have a native mechanism for *backtracking* a process execution: we cannot synchronise on an event and then go back to the state before this communication. A complicating factor is that we are dealing with new-in-context events, not new-in-alphabet events; if this were the case, alphabetised parallelism and hiding could be sufficient to compare the behaviours of the two components modulo the new events, as already demonstrated in Wehrheim (2002).

To circumvent this problem we define an alternative representation for I/O processes, with a finite LTS. We serialise an I/O process as a sequence of tuples of the form (ev, a_ev, l), for a particular state, where the event *ev* is possible from this state and if it happens the next state can accept the events in the sequence *a_ev*, which is at the level $l + 1$. The initial state's level is zero; each subsequent state has the level of its immediate predecessor increased by one.

Let us consider a practical example on how an I/O process can be serialised. Consider the process *P* in Fig. 5a. We define two special events `start` and `end`; these are used only as marking events, and will not be part of the *GLB_CVG* behaviour, but will play a role in its construction: `start` indicates that a process is ready to engage by offering its initial events; `end` marks the state where it is ready to come back to its initial state. Line 1 of `P_serial` (see Listing 4) indicates that *P* can, initially (state level zero) accept `c.in.v.1` or `c.in.v.2`; if `c.in.v.1` happens (line 2, state level 1) then it can output (non-deterministically) `c.out.v.1` or `c.out.v.2`; if it outputs `c.out.v.1` (line 3, state level 2) then it can only go back to its initial state (`end`). Note that we follow a nested-structural pattern, which allows us to backtrack an I/O process by traversing its serial representation in a recursive manner.

A question that arises is how to deal with parallelism in such a representation. We take advantage of the fact that any parallel process has a unique sequential representation in terms of the operators $\rightarrow$, $\sqcap$ and $\Box$ Roscoe (1998). This is the background we need to present our strategy (Fig. 6). Given an I/O process *P* we serialise it as *P_serial*, which is passed to the process *CVG_BUILDER*; *CVG_BUILDER*(*P_serial*) is our strategy to build precisely *GLB_CVG*(*P*), by coordinating the processes *EXEC* and *EXEC_Q*. *CVG_BUILDER* traverses the *P_serial* tuples testing

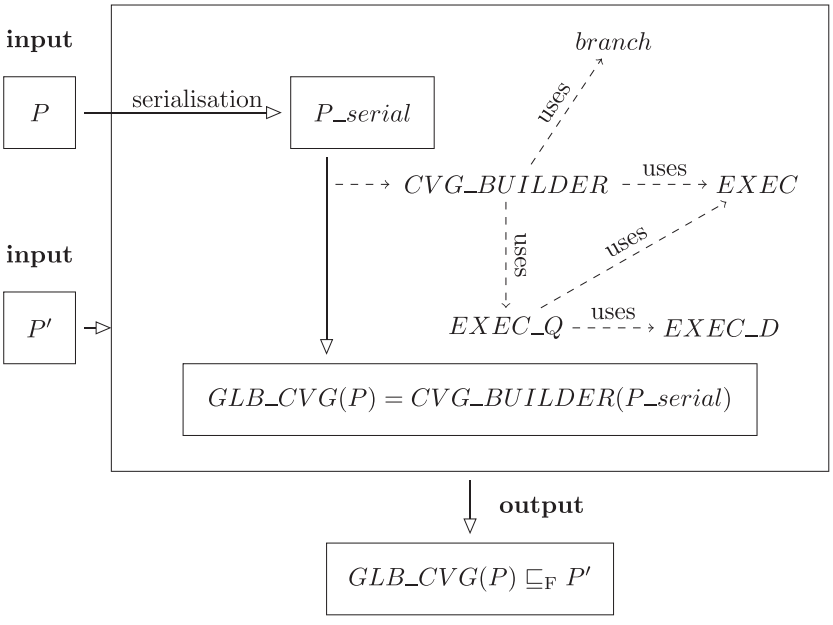


Fig. 6. Conformance checking strategy.

```
1 EXEC(evs,n) =
2 if n > 0 then
3 |  ⊓ x : evs @ ((x → EXEC(evs, n-1)) ⊓
4 |                EXEC(evs, n-1))
5 else
6 |  SKIP
7 end
```

Listing 5. EXEC.

```
1 EXEC_D(evs,n) =
2 if n > 0 then
3 |  □ x : evs @ ((x → EXEC_D(evs, n-1)) □
4 |                EXEC_D(evs, n-1))
5 else
6 |  SKIP
7 end
```

Listing 6. EXEC_D.

```
EXEC_Q(evs1, evs2, n) =
  EXEC(diff(evs1, evs2),n)
          ||
      diff(evs1,evs2)
  EXEC_D(evs1,n)
```

Listing 7. process EXEC_Q.

```
channel start, end
e1((e,_,_)) = e
e2((_,e,_)) = e
e3((_,_,e)) = e
subset(s1,s2) = empty(diff(s1,s2))
```

Listing 8. Helper functions.

```
1  <(c.in.v.1, <c.out.v.1, c.out.v.2>, 1),
2   (c.out.v.1, <end>, 2),
3   (end, <>, 3),
4   (c.out.v.2, <end>, 2),
5   (end, <>, 3)>
```

Listing 9. Branch at level one.

```
1  <(c.out.v.2, <end>, 2),
2   (end, <>, 3)>
```

Listing 10. Branch at level two.

```
1 branch(key,<>, l, b) = <>
2 branch(key,s, l, b) =
3 if (e1(head(s)) == key and e3(head(s)) == l) then
4 |  <head(s)> ^ branch(key,tail(s),l, true)
5 else
6 |  if b then
7 |  |  if (e1(head(s)) != key and e3(head(s)) == l)
  |  |  then
8 |  |  |  <>
9 |  |  else
10|  |  |  <head(s)> ^ branch(key,tail(s),l, b)
11|  |  end
12|  else
13|  |  branch(key,tail(s),l, b)
14|  end
15 end
```

Listing 11. branch function.

whether it has: (a) found an output event, which makes it offer, prior to this output, all inputs in internal choice iteratively, for n times (behaving as *EXEC*), then recursing on the next branch of *P_serial*; (b) found an input event, in which case it offers in internal choice iteratively, for n times (by using *EXEC_Q*) the complementary inputs, while ensuring this complement do not take precedence over expected inputs; (c) reached *P_serial* end, then recursing to its start. Processes *EXEC* and *EXEC_Q* rely on the *branch* function to traverse all behavioural paths of P (subsequences of *P_serial*). The construction of $GLB\_CVG(P)$ ensures it offers, at any point, at least n new-in-context inputs, therefore, a process P' convergent to P (differing in depth by at most n) must refine $GLB\_CVG(P)$.

We detail these processes in the sequel, using the CSP syntax (Appendix A). The process $EXEC(evs, n)$, in Listing 5, offers the internal choice between the events *evs* iteratively, up to n times. The process *EXEC_D* (Listing 6) is quite similar to *EXEC* but differs as it offers *evs* in external choice.

The process *EXEC_Q* (Listing 7) combines *EXEC* and *EXEC_D* in parallel. Let $evs1$ and $evs2$ be sets of events such that $evs2 \subseteq evs1$, then *EXEC* offers $evs1 \setminus evs2$ in internal choice and *EXEC_D* offers $evs1$ in external choice, iteratively, up to n times, synchronising, in each step, on the set $evs1 \setminus evs2$.

Given a 3-tuple of an I/O process serialised representation, the functions $e1$, $e2$ and $e3$ (Listing 8) yield the first, the second and the third element of the tuple, respectively. We declare the aforementioned channels *start* and *end* and define the function $subset(s1, s2)$ to check whether $s1$ is a subset of $s2$.

The *branch* function (Listing 11) yields a local view of a serialized I/O process: giving some event *key* at a particular level l, it provides the tuples (a branch) reached from *key* (at level l) until the end mark (the event *end*). For example, considering `P_serial` (Listing 4), the *branch* of the event `c.in.v.1`, at the level one, and of the event `c.out.v.2`, at the level two, gives, respectively, the local serialised views at Listings 9 e 10.

The *branch* function (Listing 11) works by finding the tuples, in a sequence s, whose events are offered together, by looking for one of them, say *key*, at a specific level l. If s is empty (line 2) the search is finished and the result remains unchanged (line 3). Otherwise, we check if the current tuple has the event at the level we are looking for (line 5); if it is the case, we append this tuple to the result and call *branch* recursively, passing to it the remainder of the sequence and setting the parameter b (marking the first occurrence of a tuple with *key*) to *true* (line 6).

If we have found the key event (b is true, line 8), we must check if we have not reached another branch of the process LTS at the same level (line 9), which marks the end of our search (line 10), otherwise we append such a tuple to the result and call *branch* to the rest of the sequence (line 12). If we have not yet found the *key*, we maintain the result unchanged and recursively call *branch* to the rest of the sequence (line 15).

```
1 CVG_BUILDER(src,crt) =
2 if e1(head(crt))==end then
3 |  CVG_BUILDER(src,src)
4 else
5 |  if subset(set(e2(head(crt))), inputs) then
6 |  |  EXEC_Q(inputs, set(e2(head(crt))), GAP);
  |  |  CVG_BUILDER(src,crt))
7 |  |  ⊓
8 |  |  □ x:set(e2(head(crt))) @ x →
9 |  |     CVG_BUILDER(src, branch(x,tail(crt),
  |  |     e3(head(crt))+1, false)
10 |  else
11 |  |  EXEC(inputs,GAP);
12 |  |     ⊓ x:set(e2(head(crt))) @ x →
13 |  |           CVG_BUILDER(src,
  |  |           branch(x,tail(crt), e3(head(crt))+1,
  |  |           false))
14 |  end
15 end

16 GLB_CVG(P) =
   CVG_BUILDER(P_serial,P_serial) \ {end}
```

Listing 12. CVG_BUILDER and GLB_cvg.

The process *CVG_BUILDER* (Listing 12) coordinates the auxiliary processes we have seen in the procedure of building *GLB_CVG*(*P*) for an I/O process *P*. Let *src* be the serialised representation of *P* and *crt* the serialisation of a *P*'s current execution branch. Both are parameters of the *CVG_BUILDER* process and are equal initially. If we have found the end of a branch (line 2), *CVG_BUILDER* returns to *P*'s initial state by making *crt* equal to *src* (line 3) again. Otherwise it must offer an external choice between inputs (line 5) or an internal choice between outputs (line 10). In the first case, it has the chance to execute up to n ($n = GAP \mid GAP \in \mathbf{N}$) new-in-context inputs (line 6) before converging to *P* (lines 8 and 9). Note that only new in-context-inputs ($inputs \setminus set(e2(head(crt)))$) are allowed to be executed non-deterministically, as we cannot violate the external choice between expected inputs $set(e2(head(crt)))$; therefore we use the helper process *EXEC_Q* to do such a task. In the second case (line 10), before an internal choice between outputs is offered, *CVG_BUILDER* can do up to n inputs (line 11) before converging to *P* (lines 12 and 13).

Therefore, if we consider the CSP processes T and T', whose labelled transition systems are depicted in Fig. 5a and b, respectively, the following FDR4 assertions hold:

i. `assert GLB_CVG(T_serial) [F= T'`
ii. `assert GLB_CVG(T_serial) [F= T`

According to our strategy, the first assertion implies that T' `io_cvg` T. The second assertion is particularly not surprising: it comes from Lemma 2, by which T `io_cvg` T because $T \sqsubseteq_F T$.

4.2. Building GLB_ECVG

The construction of *GLB_ECVG* follows the same principles used to build *GLB_CVG*. The differences relate to some additional auxiliary processes, mainly because *GLB_ECVG* must be able to do, after a new-in-context input, a sequence of any new-in-context events, before converging (Definition 8). Processes *EXEC_D* (Listing 6) and *EXEC_Q* (Listing 7) remain as presented before.

```
1 EXEC(evs, n) =
2 if (n > 0) then
3 |  ⊓ x : evs @ (x → in_ack → EXEC(evs, n -1)
4 |              ⊓
5 |              EXEC(evs, n -1))
6 else
7 |  SKIP
8 end
```

Listing 13. processes EXEC.

```
AFT_IN = in_ack → in_rdt → SKIP □ SKIP

EXEC_AFTER_IN(evs,n) =
      in_rdt → EXEC(evs, n) □ SKIP
```

Listing 14. AFT_IN and EXEC_AFTER_IN.

```
1 EXEC_Q_AFT(evs1, evs2, evs3, n) =
2   (EXEC_Q(evs1, evs2, 1)  ||{|in_ack|}  AFT_IN )
3     ||{|in_rdt|}
4   EXEC_AFTER_IN(diff(evs3, evs2),n)

5 EXEC_AFT(evs1, evs2, evs3, n) =
6   (EXEC(evs1,1)  ||{|in_ack|}  AFT_IN)
7     ||{|in_rdt|}
8   EXEC_AFTER_IN(diff(evs3, evs2),n)
```

Listing 15. EXEC_Q_AFT and EXEC_AFT.

As we need to detect an occurrence of a new-in-context *input* before allowing *any* new-in-context event we performed a subtle change in the *EXEC* process (Listing 13): it acknowledges (by communicating the *in_ack* event) every time some event happens, making it possible to know when a new-in-context *input* was communicated.

The process *AFT_IN* (Listing 14) acts like a watcher for *in_acks* events, reverberating them by communicating the *in_rdt* event. We model the process *EXEC_AFTER_IN* (Listing 14) to catch the *in_rdt* event and turn back to *EXEC*. Note that, playing the role of watchers, both processes cannot force anything to happen, so the successful termination, *SKIP*, is always a possibility.

The process *EXEC_Q_AFT* (Listing 15) acts like a wrapper to *EXEC_Q*. It is parametrised by three sets of events *evs*1, *evs*2 and *evs*3: the first two have the same intent of their counterparts in *EXEC_Q*, where the purpose of the latter is to receive the set containing the events that can happen after a new-in-context input of $evs1 \setminus evs2$ (line 1). After an input, *EXEC_Q* synchronises on *in_ack* with *AFT_IN* (line 2), which in turn communicates *in_rdt*, at this time synchronising with *EXEC_AFTER_IN*, which is responsible for offering, iteratively, up to n times any $evs3 \setminus evs2$ (inputs or outputs) new-in-context events (line 4).

Process *EXEC_AFT* follows the same reasoning used in *EXEC_Q_AFT*, but it uses *EXEC* instead of *EXEC_Q*: it is designed to be used before an internal choice among outputs, therefore there is no need to retain the external choice of expected inputs, as *EXEC_Q* does.

The process *ECVG_BUILDER* (Listing 16) coordinates the auxiliary processes we have seen in the task of building *GLB_ECVG*(*P*) for an I/O process *P*, based on its serialised representation *src*

```
1  ECVG_BUILDER(src,crt) =
2  if e1(head(crt))==end then
3  |  ECVG_BUILDER(src,src)
4  else
5  |  if subset(set(e2(head(crt))), inputs) then
6  |  |  EXEC_Q_AFT(inputs, set(e2(head(crt))), all,
   |  |  GAP-1);
7  |  |    ECVG_BUILDER(src,crt)
8  |  |  ⊓
9  |  |  □ x:set(e2(head(crt))) @ x →
10 |  |    ECVG_BUILDER(src, branch(x,tail(crt),
   |  |    e3(head(crt))+1, false))
11 |  else
12 |  |  EXEC_AFT(inputs, set(e2(head(crt))), all,
   |  |  GAP-1);
13 |  |  ⊓ x:set(e2(head(crt))) @ x →
14 |  |    ECVG_BUILDER(src, branch(x,tail(crt),
   |  |    e3(head(crt))+1, false))
15 |  end
16 end
17 GLB_ECVG(P) =
   ECVG_BUILDER(P_serial,P_serial)
18                          \ {end, in_ack,in_rdt}
```

Listing 16. ECVG_BUILDER and GLB_ECVG.

(line 1). It distinguishes from *CVG_BUILDER* by the use of the auxiliary processes *EXEC_Q_AFT* and *EXEC_AFT* instead of *EXEC_Q* and *EXEC*, respectively. The process *EXEC_Q_AFT* (line 6) can execute up to n (where $n = GAP$) new-in-context events ($all = inputs \cup outputs$), but the first must be an input; as mentioned earlier, it differs from *EXEC_AFT* (line 12) as it preserves the external choice among expected inputs.

Now if we consider the CSP processes T and T'' in Figs. 5a and c, respectively, the following FDR4 assertions hold:

i. `assert GLB_ecvg(T_serial) [F= T''`
ii. `assert GLB_ecvg(T_serial) [F= T`

As before, the first assertion holds, which implies, according to our strategy, that T'' `io_ecvg` T and the second one, as a consequence of Lemma 2, from where we know T `io_ecvg` T because $T \sqsubseteq_F T$.

5. Case study

We model an autonomous healthcare robot that monitors and medicates patients, being able to contact the relevant individuals or systems in case of emergency. It receives data from a number of sensors and actuates by injecting intravenous drugs and/or by calling the emergency medical services and the patient's relatives or neighbours. We use the following data types (Listing 17): `BI` (breath intensity), `BT` (body temperature), `DD` (drug dose), `BGL` (blood glucose level), `CL` (call list, the relevant individuals to be called in the case of emergency), `DRUG` (the drugs in the robot's actuators), `QUEST` (the robot's question list, to ask the patient when its voice recognition module is used).

These types are composed into more elaborated ones whose data will be communicated through the channels used to connect sensors, actuators and phones to the robot. The set `EVENTS` encompasses the data sent in and out of:

```
nametype BI = {1..5}
nametype BT  = {34..41}
nametype DD = {0..5}
datatype BGL = low | normal | threshold | high
datatype CL = c911 | cFamily | cNeighbor | ack
datatype DRUG = insulin | painkiller | antipyretic
datatype QUEST = chest | head | vision | lst

datatype EVENTS =
 breath.BI | bodyTemp.BT | bloodGlucose.BGL |numbnessFace.Bool |
 fainting.Bool | cough.Bool | troubleSpeaking.Bool |
 visionTrouble.Bool | chestDiscomfort.Bool | headache.Bool |
 ask.QUEST| call.CL | administer.DRUG.DD
datatype IO = out.EVENTS | in.EVENTS

subtype BS = breath.BI | bodyTemp.BT | bloodGlucose.BGL
subtype I_BS = in.BS | out.BS

subtype IS = numbnessFace.Bool |  fainting.Bool
subtype I_IS = in.IS | out.IS
```

Listing 17. Types.

the body attached sensors (`breath.BI`, `bodyTemp.BT` and `bloodGlucose.BGL`), the vision recognition devices (`numbness Face.Bool` and `fainting.Bool`), the noise recognition device (`cough.Bool` and `troubleSpeaking.Bool`), the voice interaction devices (`visionTrouble.Bool`, `chestDiscomfort .Bool`, `headache.Bool` and `ask.QUEST`), the phone interface (`call.CL`) and the intravenous injection actuator (`administer.DRUG.DD`).

An event in `EVENTS` can be communicated as an output to a component and become an input to the other to which it connects by one of the $\mathcal{BRIC}$ composition rules. We define `IO` as the set `EVENTS` where each value is tagged with `in` and `out`, which differentiates inputs from outputs.

Each sensor/device communicates with the robot component via a specific channel, according to this schema (Listing 18): `bodySen`, the body attached sensors; `imageRec`, the vision recognition devices/sensors; `voiceRec`, the noise recognition devices/sensors; `talk`, the voice interaction devices/sensors; `phone`, the phone's interface and `intravenousNeedle`, the intravenous injection actuator. Each channel has its own type (a subset of `IO`) that involves only functionality related events. As an example, consider a channel with the `I_BS` type (`I_BS` $\subset$ `IO` and `BS` $\subset$ `EVENTS`), then it can communicate any event registered by the body attached sensors: breath, body temperature and blood glucose level.

```
subtype VS = cough.Bool | troubleSpeaking.Bool
subtype I_VS = in.VS | out.VS

subtype TK = visionTrouble.Bool  |  chestDiscomfort.Bool |
             headache.Bool | ask.QUEST
subtype I_TK = in.TK | out.TK

subtype PH = call.CL
subtype I_PH = in.PH | out.PH

subtype IVN = administer.DRUG.DD
subtype I_IVN = in.IVN | out.IVN

channel bodySen : I_BS
channel imageRec : I_IS
channel voiceRec : I_VS
channel talk : I_TK
channel phone : I_PH
channel intravenousNeedle : I_IVN
```

Listing 18. Types and channels.

```
HC_BOT = bodySen.in.breath?x ->
if (x < 3) then bodySen.out.breath.x ->  MOD_CALL_P1; HC_BOT
else voiceRec.in.cough?b ->
 if (b) then bodySen.in.bodyTemp?t-> bodySen.in.bloodGlucose?g->
  if(t > 38)
  then |~| d_ap : DD @
   intravenousNeedle.out.administer.antipyretic.d_ap ->
   MOD_CALL_P2 ; HC_BOT
  else
   if (g == high or g ==threshold)
   then |~| d_in : DD @
    intravenousNeedle.out.administer.insulin.d_in ->
    MOD_CALL_P2 ; HC_BOT
   else HC_BOT
 else
  imageRec.in.numbnessFace?nf ->
  imageRec.in.fainting?f ->
   if (nf or f) then
    |~| d_pk : DD @
     intravenousNeedle.out.administer.painkiller.d_pk ->
     MOD_CALL_P1; HC_BOT
   else HC_BOT

MOD_CALL_P1 = phone.out.call.cNeighbor -> phone.out.call.c911 ->
phone.out.call.cFamily -> phone.in.call.ack ->
phone.in.call.ack -> SKIP

MOD_CALL_P2 = phone.out.call.cNeighbor ->
phone.out.call.cFamily -> phone.in.call.ack -> SKIP
```

Listing 19. HC_BOT .

The behaviour of our healthcare robot is defined in terms of the I/O process HC_BOT (Listing 19). It waits for the breath level indicator; if this level is critical (< 3), then it behaves as MOD_CALL_P1 (module phone call priority one), which contacts a patient's neighbour, the registered emergency service and relatives, in this order; then it waits for at least two of them to acknowledge before coming back to its initial state. Otherwise, the patient is breathing normally, and the robot reads the noise sensor to check whether he or she is coughing (voiceRec.in.cough?b). If so, it reads the body temperature (bodySen.in.bodyTemp?t) and blood glucose (bodySen.in.bloodGluco se?g) sensors. If the body temperature exceeds 38°C, then it administers a dose of antipyretic (intravenousNeedle.out. administer.antipyretic.d_ap). If the blood glucose level is in the threshold or high, it administers the hormone insulin (intra venousNeedle.out.administer.insulin.d_in), otherwise it just comes back to its initial state. After administrating any drug, and before coming back to its initial state, the robot must contact the patient's neighbour and relatives by behaving as MOD_CALLP2 (module phone call priority two, Listing 19), in which case at least one of them must acknowledge.

If the patient is breathing normally but in silence, the robot asks the image recognition module to inform about: any unusual sign in his face (imageRec.in. numbnessFace?nf) or if he fainted (imageRec.in.fainting?f). If at least one condition holds, the robot administers a painkiller (intravenousNeedle. out. administer.painkill er.d_pk), calls the relevant individuals by behaving as MOD_CALL_P1 (Listing 19). In any case, it goes to its initial state.

In $\mathcal{BRIC}$, the healthcare robot is defined in terms of the Ctr_{HC_BOT} contract (Fig. 7a). It behaves as HC_BOT and can interact with its environment by one of its visible communication channels: bodySen, imageRec, voiceRec, phone and intravenousNeedle.

The robot Ctr_{HC_BOT} is able to diagnose and select the appropriate drug to be administered. Nevertheless, this process abstracts from establishing an appropriate drug dose given the seriousness of the patient condition; in fact it is a nondeterministic decision. For example, consider the indexed nondeterministic choice | | ds:DD @ intravenous Needle.out. administer.antipyretic.ds, which offers the events intravenousNeedle.out.administer.antipyretic.ds for all values of ds in DD. No matter the seriousness of the fever, one might know what will be the dose ds to be administered to the patient. The I/O process HC_BOT_ACC (Listing 20) addresses the dose issue by using two criteria: each degree

$$Ctr_{HC\_BOT} \mathrel{\widehat{=}} \left\langle \mathtt{HC\_BOT}, \left\{ \begin{array}{l} \mathtt{bodySen} \mapsto \mathtt{I\_BS}, \mathtt{imageRec} \mapsto \mathtt{I\_IS}, \\ \mathtt{voiceRec} \mapsto \mathtt{I\_VS}, \mathtt{phone} \mapsto \mathtt{I\_PH}, \\ \mathtt{intravenousNeedle} \mapsto \mathtt{I\_IVN} \end{array} \right\}, \left\{ \begin{array}{l} \mathtt{I\_BS}, \mathtt{I\_IS}, \\ \mathtt{I\_VS}, \mathtt{I\_PH}, \\ \mathtt{I\_IVN} \end{array} \right\}, \right\rangle$$
$$\{\mathtt{bodySen}, \mathtt{imageRec}, \mathtt{voiceRec}, \mathtt{phone}, \mathtt{intravenousNeedle}\}$$

(a) Ctr_{HC_BOT}

$$Ctr_{HC\_BOT\_ACC} \mathrel{\widehat{=}} \left\langle \mathtt{HC\_BOT\_ACC}, \left\{ \begin{array}{l} \mathtt{bodySen} \mapsto \mathtt{I\_BS}, \mathtt{imageRec} \mapsto \mathtt{I\_IS}, \\ \mathtt{voiceRec} \mapsto \mathtt{I\_VS}, \mathtt{phone} \mapsto \mathtt{I\_PH}, \\ \mathtt{intravenousNeedle} \mapsto \mathtt{I\_IVN} \end{array} \right\}, \left\{ \begin{array}{l} \mathtt{I\_BS}, \mathtt{I\_IS}, \\ \mathtt{I\_VS}, \mathtt{I\_PH}, \\ \mathtt{I\_IVN} \end{array} \right\}, \right\rangle$$
$$\{\mathtt{bodySen}, \mathtt{imageRec}, \mathtt{voiceRec}, \mathtt{phone}, \mathtt{intravenousNeedle}\}$$

(b) $Ctr_{HC\_BOT\_ACC}$

$$Ctr_{HC\_BOT\_TK} \mathrel{\widehat{=}} \left\langle \mathtt{HC\_BOT\_TK}, \left\{ \begin{array}{l} \mathtt{bodySen} \mapsto \mathtt{I\_BS}, \mathtt{imageRec} \mapsto \mathtt{I\_IS}, \\ \mathtt{voiceRec} \mapsto \mathtt{I\_VS}, \mathtt{phone} \mapsto \mathtt{I\_PH}, \\ \mathtt{intravenousNeedle} \mapsto \mathtt{I\_IVN}, \\ \mathtt{talk} \mapsto \mathtt{I\_TK} \end{array} \right\}, \left\{ \begin{array}{l} \mathtt{I\_BS}, \mathtt{I\_IS}, \\ \mathtt{I\_VS}, \mathtt{I\_PH}, \\ \mathtt{I\_IVN}, \mathtt{I\_TK} \end{array} \right\}, \right\rangle$$
$$\{\mathtt{bodySen}, \mathtt{imageRec}, \mathtt{voiceRec}, \mathtt{phone}, \mathtt{intravenousNeedle}, \mathtt{talk}\}$$

(c) $Ctr_{HC\_BOT\_TK}$

Fig. 7. Autonomous healthcare robots components.

```
HC_BOT_ACC = bodySen.in.breath?x ->
if (x < 3) then bodySen.out.breath.x ->  MOD_CALL_P1; HC_BOT_ACC
else voiceRec.in.cough?b ->
 if (b)
 then bodySen.in.bodyTemp?t -> bodySen.in.bloodGlucose?g ->
  if(t > 38)
  then intravenousNeedle.out.administer.antipyretic.t%37 ->
   MOD_CALL_P2 ; HC_BOT_ACC
  else
   if (g == high)
   then |~| d_in_h : {3,4,5} @
    intravenousNeedle.out.administer.insulin.d_in_h ->
    MOD_CALL_P2 ; HC_BOT_ACC
   else
    if (g == threshold)
    then |~| d_in_t : {1,2} @
     intravenousNeedle.out.administer.insulin.d_in_t ->
     MOD_CALL_P2 ; HC_BOT_ACC
    else HC_BOT_ACC
 else imageRec.in.numbnessFace?nf -> imageRec.in.fainting?f ->
  if (nf or f) then |~| d_pk : DD @
   intravenousNeedle.out.administer.painkiller.d_pk ->
   MOD_CALL_P1; HC_BOT_ACC
  else HC_BOT_ACC
```

Listing 20. HC_BOT_ACC .

above 38°C corresponds to a unit of the prescribed antipyretic (`intravenousNeedle.out.administer.antipyretic.t`)

This behaviour extension is defined by the $\mathcal{BRIC}$ contract $Ctr_{HC\_BOT\_ACC}$ (Fig. 7b). This healthcare robot version has a better (more deterministic) decision-making mechanism on the drug dose to be administered to the patient it monitors. By Definition 12, we have that $Ctr_{HC\_BOT} \sqsubseteq_{\mathcal{B}} Ctr_{HC\_BOT\_ACC}$: both components share the same channels with equivalent types (interfaces) and have their behaviours related by failures refinement `HC_BOT` $\sqsubseteq_F$ `HC_BOT_ACC`; it can be verified by the FDR4 assertion `assert HC_BOT [F= HC_BOT_ACC`.

The $Ctr_{HC\_BOT\_ACC}$ brings some improvements to Ctr_{HC_BOT}. Nevertheless, the addition of new functionalities (or the enhancement of the existing ones) cannot be always addressed by refinement, even if we hide the implementation details before trying to establish such a relation, as already discussed. The component we present next, $Ctr_{HC\_BOT\_TK}$ (Fig. 7c), extends (inherits from) $Ctr_{HC\_BOT\_ACC}$ (Fig. 7b) with the addition of a talk module, which allows this robot to ask patients about their symptoms and thus can possibly better help them.

The I/O process `HC_BOT_TK` ($Ctr_{HC\_BOT\_TK}$ behaviour, Listing 21) improves `HC_BOT_ACC` by being able to interact with patients via the voice simulation/recognition device through the new channel `talk`. Together with the events `bodySen.in.breath?x`, it offers, initially, the possibility of behaving as `MOD_TALK`: it receives a chat request (`talk.in.ask.lst`), then collects information about chest discomfort (`talk.in.chestDiscomfort?cd`), headache (`talk.in.headache? hd`) and vision problems (`talk.in.visionTrouble?vt`). If the patient reports chest discomfort associated with headache or vision problems, the robot understands that a serious situation is under way and calls all the relevant individuals by behaving as `MOD_CALL_P1`. In any case, it goes to its initial state.

By Definition 11, we have that $Ctr_{HC\_BOT\_ACC} \leftarrow_{ecvg} Ctr_{HC\_BOT\_TK}$. Note that the attempt to establish a failures relation between `HC_BOT_ACC` and `HC_BOT_TK`, provided the events communicated through `talk` are hidden on the latter, fails: as the FDR4 assertion `HC_BOT_ACC [F= HC_BOT_TK |talk|` proves. This shows that convergence and inheritance, in the behavioural and component level perspectives, provide an entire new approach to evolve component based specifications, whilst preserving deadlock freedom. The resulting component hierarchy is depicted in Fig. 8.

This hierarchy guarantees important results when composing the healthcare robot. Suppose we have a drug storage component Ctr_{DRUG_STR} that dispenses drugs, as requested, and informs, afterwards, stock level status; also, consider a communicator hub component Ctr_{HUB_COM} that handles communications through different mediums: phone calls, messages, audio stream and e-mails (their I/O processes are omitted here for the sake of brevity). Considering the following three compositions:

$$Ctr_{SYS} = Ctr_{HC\_BOT}[\texttt{intravenousNeedle} \leftrightarrow \texttt{drugDispenser}]Ctr_{DRUG\_STR}$$
$$Ctr_{SYS2} = Ctr_{HC\_BOT\_ACC}[\texttt{intravenousNeedle} \leftrightarrow \texttt{drugDispenser}]Ctr_{DRUG\_STR}$$
$$Ctr_{SYS3} = (Ctr_{HC\_BOT\_TK}[\texttt{intravenousNeedle} \leftrightarrow \texttt{drugDispenser}]Ctr_{DRUG\_STR})[\texttt{talk} \leftrightarrow \texttt{audioStream}]Ctr_{HUB\_COM}$$

We have that (a) since $Ctr_{HC\_BOT} \sqsubseteq_{\mathcal{B}} Ctr_{HC\_BOT\_ACC}$, we know, by monotonicity of $\mathcal{BRIC}$ component refinement, that $Ctr_{SYS} \sqsubseteq_{\mathcal{B}} Ctr_{SYS2}$ also holds (both being deadlock free) and, for Ctr_{DRUG_STR}, it is impossible to distinguish between the different healthcare

```
HC_BOT_TK =
bodySen.in.breath?x ->
 if (x < 3)
 then bodySen.out.breath.x ->  MOD_CALL_P1; HC_BOT_TK
 else voiceRec.in.cough?b ->
  if (b) then bodySen.in.bodyTemp?t ->
   bodySen.in.bloodGlucose?g ->
   if(t > 38)
   then  intravenousNeedle.out.administer.antipyretic.t%37 ->
    MOD_CALL_P2 ; HC_BOT_TK
   else
    if (g == high)
    then |~| d_in_h : {3,4,5} @
     intravenousNeedle.out.administer.insulin.d_in_h ->
     MOD_CALL_P2 ; HC_BOT_TK
    else
     if (g == threshold)
     then |~| d_in_t : {1,2} @
      intravenousNeedle.out.administer.insulin.d_in_t ->
      MOD_CALL_P2 ; HC_BOT_TK
     else HC_BOT_TK
  else
   imageRec.in.numbnessFace?nf -> imageRec.in.fainting?f ->
   if (nf or f)
   then |~| d_pk : DD @
    intravenousNeedle.out.administer.painkiller.d_pk ->
    MOD_CALL_P1; HC_BOT_TK
   else HC_BOT_TK
[]
MOD_TALK ; HC_BOT_TK

MOD_TALK = talk.in.ask.lst ->
        talk.out.ask.chest -> talk.in.chestDiscomfort?cd ->
        talk.out.ask.head -> talk.in.headache?hd ->
        talk.out.ask.vision -> talk.in.visionTrouble?vt ->
        if (cd and (hd or vt)) then MOD_CALL_P1 else SKIP
```

Listing 21. HC_BOT_TK.

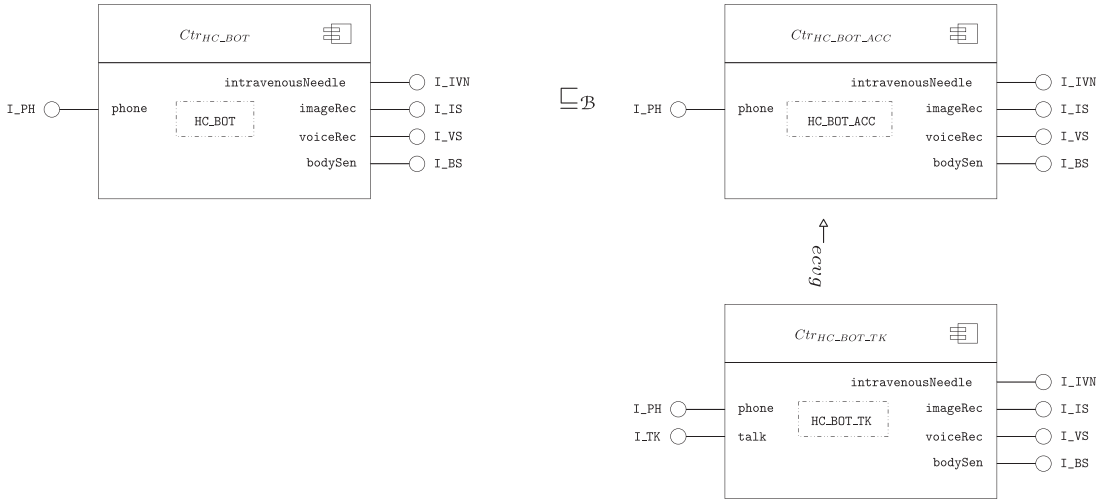


Fig. 8. Autonomous healthcare robot hierarchy.

robots; (b) as $Ctr_{HC\_BOT\_ACC} \xleftarrow{ecvg} Ctr_{HC\_BOT\_TK}$, Ctr_{SYS3} is deadlock free and, again for Ctr_{DRUG_STR}, it is impossible to distinguish between robots, which is true not only for the component Ctr_{DRUG_STR} but for any component able to connect to Ctr_{HC_BOT}, no matter whether new channels on $Ctr_{HC\_BOT\_TK}$ are exercised (as the `talk` channel, which is hooked to `audioStream` from Ctr_{HUB_COM}).

It is important to consider non-convergent component extensions, which can introduce deadlock. Suppose we define a contract $Ctr_{HC\_BOT\_TK\_ECHO}$, which echoes the patient possible responses (output events) asking for confirmation. As these events are new-in-context outputs (not inputs), the contract $Ctr_{HC\_BOT\_TK\_ECHO}$ does not inherit by convergence from $Ctr_{HC\_BOT\_TK}$, as the former can output and, without response lock, where the latter does not. In FDR4 the `assert GLB_CVG(HC_BOT_TK_serial) [F= HC_BOT_TK_ECHO` fails. Although the traces:

- ⟨`bodySen.in.breath.2`, `bodySen.out.breath.2`, `phone.out.call.cNeighbor`⟩, from `HC_BOT_TK`
- ⟨`bodySen.in.breath.2`, `bodySen.out.breath.2`, `phone.out.call.cNeighbor`, `echo.in.response.yes`⟩, from `HC_BOT_TK_ECHO`

are convergent, the following traces:

- ⟨`bodySen.in.breath.2`, `bodySen.out.breath.2`, `phone.out.call.cNeighbor`⟩, from `HC_BOT_TK`)
- ⟨`bodySen.in.breath.2`, `bodySen.out.breath.2`, `phone.out.call.cNeighbor`, **`echo.out.timeout`**, `echo.out.response.yes`⟩, from `HC_BOT_TK_ECHO`

are not convergent, causing the nonconformance. It happens because of the `echo.out.timeout`, which signals a timeout in the patient response, but it is a new-in-context output, not a new-in-context input.

Finally, it is important to notice that this work does not provide a guideline to evolve or correct the models with respect to convergence: it is beyond the scope of this work and is a topic to be addressed as future work.

6. Related work

Apart from the precise definition of component behaviour and interface, a formal approach to CB-MDD must specify how components can be assembled into more complex ones, how they can be refined and, ultimately, how to evolve them into more specialised or functional ones. Several works have proposed a formal foundation for CB-MDD: (Ramos, 2011; Liu et al., 2009; Jifeng et al., 2005; Wang et al., 2009; Chen et al., 2009b; 2007; Meng and Barbosa, 2006; Hennicker et al., 2010; Bauer et al., 2010). Specially, the work reported in Ramos (2011) focuses on the development of deadlock free component-based systems by construction. Nevertheless, it does not propose a refinement or an inheritance relation for components, which is the main purpose of the current work.

In rCOS Liu et al. (2009) a component has a provided and a required interface and code associated to each of their method signatures. An interface is a syntactic notion that encompasses typed variable declarations, called fields, and method signatures with input and output parameters alongside with their types. This approach is distinguished from $\mathcal{BRIC}$ by not treating inputs as an environment decision and outputs as an internal decision, by not analysing behavioural properties in its composition rules and by stating the results on traces instead of the failures model, which compromises substitutability, since, for example, the traces model does not allow one to reason about deadlock freedom.

In rCOS, a contract refines another when there is a corresponding refinement between their required and provided interfaces (given in terms of the Unifying Theories of Programming (UTP) semantics Hoare and He (1998)) and between their traces (traces semantics). The works in Liu et al. (2009), Chen et al. (2009b), Chen et al. (2007) require that both components have the same provided and required interface, as we assume in our refinement relation, although the approaches in Jifeng et al. (2005) and Wang et al. (2009) allow interface extension.

Other works, notably (Hennicker et al., 2010; Bauer et al., 2010), have established their roots in the transition systems theory: in Hennicker et al. (2010), the authors define an I/O labelling as a 3-tuple of outputs, inputs and internal labels used by an I/O-transition system, which encompasses a set of states and transitions. A component is formed of a set of ports and an observable behaviour given in terms of an I/O-transition system. A component refines another if both have the same ports and there is a correspondence between the states of their I/O-transition systems. There is no possibility of functionality extension, since both components must have the same ports and any observable behaviour of one must be possible by the other. The approach in Bauer et al. (2010) uses similar I/O transition systems to define component behaviour. It allows functionality extension by ex-

pressing component refinement in terms of a mapping function between states and transitions of two I/O transition systems. It adopts a similar approach to Meng and Barbosa (2006), whereas ours stands with that of Hennicker et al. (2010), since our understanding is towards distinguishing refinement from inheritance, considering refinement as a way to achieve non-determinism reduction and inheritance as a way to embed on new system functionalities.

Some works have proposed inheritance relations for behavioural specifications (Liskov and Wing, 1994; Wehrheim, 2003; America, 1991; Bowman and Derrick, 1999; Puntigam, 1996; Dihego et al., 2013). In America (1991) and Liskov and Wing (1994) inheritance relations are defined in terms of invariants over state components and by pre and postconditions over defined methods. The remaining works define subtype relations based on models like failures and failures-divergences (denotational models of CSP), relating refinement with inheritance Wehrheim (2003). None of them differentiates inputs from outputs nor considers structural elements besides behavioural specifications, although Wehrheim (2003) analyses substitutability and relates it with behavioural properties, as deadlock freedom.

Aligned to the mentioned works on behavioural inheritance, we also state our relations in the failures model, but we distinguish inputs from outputs, not only by putting them in different sets, but restricting the way they can be communicated. The work in Bertrand et al. (2011) presents a relation named I/O abstraction that has some connection with convergence. It allows an implementation to input more but restricts it to output less than its abstractions; however, it does not consider what happens after the implementation communicates a new input, which clearly weakens substitutability and thus the behavioural properties preservation, as deadlock freedom, in the composition rules. In the same way, the conformance relation used for testing, *ioco* (Tretmans, 1996), allows new inputs to be communicated by an implementation, but as the I/O abstraction relation, it differs from our work by not considering how the implementation behaves after engaging in a new input. We also highlight an important design decision we have taken in this work: we allow functionality extension to be implemented not only in terms of new events but also by existing ones; therefore we allow *new-in-context* (not only *new-in-alphabet*) events to be communicated by an inherited component. It gives more flexibility, but presents a challenge to the inheritance verification as we have demonstrated in the construction of an automated strategy for verifying convergence using the FDR4 model checker.

The treatment of inputs as an environment decision and outputs as being internally resolved by the component itself (I/O process definition) is also considered by the approach proposed in Cavalcanti and Hierons (2013), but it differs from ours by developing a new semantic model named *IOFailures*, which is not compositional, in general. Also, such a relation is proposed for testing, whereas we focus on a design that preserves behavioural properties by construction, supported by the $\mathcal{BRIC}$ composition rules.

The concept of evolution by retrenchment is presented in García-Duque et al. (2009): a retrenchment represents a contradiction of the current specification followed by an evolution of it in a different direction. We understand it is useful in the early stages of a specification, but as it evolves, we generally need to define evolutions that conform to the previous ones.

Following the lines of Liskov and Wing (1994) and Wehrheim (2003), although focusing on data refinement, the work reported in Back and Sere (1996) presents the concept of evolution of reactive action systems by superposition. Extensions must not change the original behaviour and are restricted to be defined in terms of new events. In our work, we allow extensions to reuse existing events, additionally we develop an automated strategy to ensure such extensions are valid.

It is also important to mention that van der Aalst et al. (2002) defines an inheritance relation between components, based on the notion of projection, whose semantics is given in terms of Petri nets and equivalence is checked by bisimulation. Projections define loose relationships between behaviour specifications and in general they do not allow reuse of existing events. We differ by defining inheritance for a component model (based on CSP) where behavioural properties are guaranteed by construction and are locally verified by the FDR4 model checker.

Recent works have addressed how inheritance affects substitutability (Maddox et al., 2018) and how evolution can benefit from it (Bourouz and Zeghib, 2016). In a large-scale study conducted in thousands of Java projects, the work reported in Maddox et al. (2018) has shown that a major portion of inheritance implementations violate the substitutability principle, specially in the contexts where threads were used. It highlights the importance of having theories and tools that guarantee inheritance does not break substitutability, as we propose here. In Bourouz and Zeghib (2016), Petri nets are used to model the behaviour of web services and, as $\mathcal{BRIC}$ does, it also distinguishes inputs from outputs and substitutability is given in terms of structural and behavioural aspects. We differ from the latter aspect bacause our inheritance relations ensure behaviours eventually will converge, whereas, in Bourouz and Zeghib (2016), a candidate extension is only obligated to contain the original behaviour but it is free do communicate anything else, which clearly does not guarantee deadlock-freedom.

More recently, the work presented in Lange and Atkinson (2019) has discussed the rules for inheritance in multi-level modelling, assuming every abstraction has at least one realization. It provides rules for substitutability in each level of modelling, focusing on the structural and data aspects of inheritance relations, whereas we focus on both structural and behavioural aspects of extensions as a means to guarantee behavioural properties.

From what we have seen and, as far as we are aware, this is the first time component inheritance relations are developed for a formal and sound CB-MDD approach, with a formal semantics, a refinement relation, an analysis on the impact of substitutability and an automated strategy to check conformance.

7. Conclusions

We have developed in this work a novel concept called behavioural convergence for specifications that distinguish inputs from outputs. Based on that we have developed refinement and inheritance relations for an approach to CB-MDD, where we consider structural and behavioural aspects. We incorporated these relations in a set of composition rules that guarantee deadlock freedom by construction.

First, we defined a congruent semantics for $\mathcal{BRIC}$ that considers component structure and behaviour. This makes it possible to understand the precise meaning of a component, and is the basis to define component refinement and inheritance. Our refinement notion guarantees the relevant properties required by the $\mathcal{BRIC}$ component compositions, fulfilling the substitutability principle (a refinement should be usable wherever its abstraction is expected, without a client being able to tell the difference)

As a major contribution of this work, we defined two inheritance relations for $\mathcal{BRIC}$, both based on a novel concept called *behavioural convergence*. It captures the idea that components can evolve by accepting new inputs or establishing a communication session after these inputs, but then they must *converge* to the predicted behaviour exhibited by its abstraction. Our defi-

nition of inheritance deals with component structural and behavioural aspects and guarantees substitutability, in the composition rules, preserving deadlock freedom. Therefore a component of a model can evolve by the reduction of non-determinism (refinement) or by the extension of functionality (inheritance) and still preserving, in the entire model, deadlock freedom and protocol compatibility.

We have two forms of convergence and, consequently, two inheritance relations upon them. We have proved that one is a generalisation of the other. Indeed, we have proved our relations form a hierarchy, where component refinement is the strongest relation between $\mathcal{BRIC}$ components.

We have developed an automated strategy for verifying convergence as refinement assertions using FDR4. We have systematised the construction of a *greatest lower bound* process (under the failures refinement relation), of which all convergent processes are stable-failure refinements. Based on this result, we have converted an assertion about convergence (and extended convergence) into a refinement verification, which can be carried out by FDR4. The overall approach was validated using a case study that involved the modelling and verification of an autonomous healthcare robot.

Quality attributes of programming and modelling, like high cohesion and low coupling, as well as some good practices of object-oriented design, are also useful when adopting the approach we propose for component design evolution. However, the main reuse aspect in our context is control flow behaviour, in contrast to data aspects. Therefore, some patterns, such as *decorator, command, template method* and *chain of responsibility*, and concurrent communication patterns Roscoe (1998) as, for instance, *client-server, resource sharing* and *routing*, are likely to potentialise an extensible component design model.

A major topic for future work is to devise a detailed process, tailored to support the approach proposed in this paper. The main benefit of this process is to allow the user to define, as a separate concern, model extensions that ensure convergence by construction. In this context, the developer would work, for instance, with a more appealing notation like UML or SysML. Then, for verification, the graphical models would be translated into CSP and the verification could be carried out in background, completely hidden from the developer, with proper and transparent traceability to the model.

Also as future work, we aim to develop industrial case studies such as traffic aviation control systems and extend a $\mathcal{BRIC}$ modelling tool (BTS–$\mathcal{BRIC}$ Tool Support de A. Pereira et al., 2017) to verify refinement and inheritance. We also plan to mechanise the proofs of our theorems in CSP-Prover ISOBE and ROGGENBACH (2008), an interactive theorem prover for CSP based on Isabelle/HOL Nipkow et al., 2002. The cost of verifying convergence needs to be further investigated. Finally, it is in our agenda to consider the preservation of other classical concurrency properties like (non)determinism and livelock freedom, based on the approaches in Filho et al. (2016), Filho et al. (2018), Otoni et al. (2017), as well as domain specific properties.

Declaration of Competing Interests

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Appendix A. CSP

This appendix presents the relevant definitions, the syntax and the failures semantics of CSP (Roscoe, 1998).

Table A1
CSP processes.

CSP	CSP_M	Description
STOP	`STOP`	termination
SKIP	`SKIP`	successful termination
$c \rightarrow P$	`c-> P`	prefix
$P; Q$	`P;Q`	sequential composition
$P \backslash X$	`P \ X`	hiding
$P \Box Q$	`P[]Q`	external choice
$P \sqcap Q$	`P|  |Q`	internal choice
b & P	`b & P`	boolean guard
if b then P else Q	`if b then P else Q`	if-then-else
$P[[a/b]]$	`P[[a <- b]]`	renaming
$P \|\|\| Q$	`P ||| Q`	interleaving
$P \|\| [a] Q$	`P [|a|] Q`	alphabetized parallel
$\Box\ e : X @ P$	`[] e :X @ P`	replicated external choice
$\sqcap e: X @ P$	`| | e :X @ P`	replicated internal choice
$\|\|\| e: X @ P$	`||| e : X @ P`	replicated interleave
$\|\| e: X @ [X'] P$	`|| e : X @ [X'] P`	replicated alphabetized parallel

Table A2
Events

CSP	CSP_M	Description
Σ	`all`	alphabet of all communications
$\{\|c\|\}$	`{| c |}`	the events communicated through channel c
$X \setminus Y$	`diff(X,Y)`	$\{e \mid e \in X \wedge e \notin Y\}$
$\checkmark$		(tick) termination event
τ		(tau) invisible event
$\Sigma^{\checkmark}$		$\Sigma \cup \{\checkmark\}$
$\Sigma^{\checkmark,\tau}$		$\Sigma \cup \{\checkmark, \tau\}$

Table A3
Traces.

CSP	CSP_M	Description
Σ^*		set of all finite traces over Σ
$\langle\rangle$	`<>`	the empty trace
$t \frown s$	`t^s`	concatenation of traces
$s \leq t$	`s <= t`	$\equiv \exists u.s \frown u = t$ (prefix order)
$\#s$	`#s`	length of s
$t - s$	`t - s`	$t - \langle\rangle = t$ $\langle\rangle - s = \langle\rangle$ $\langle e\rangle \frown t - \langle e\rangle \frown s = t - s$ $(\langle e_1\rangle \frown t) - (\langle e_2\rangle \frown s) = \langle e_1\rangle \frown (t - (\langle e_2\rangle \frown s)) \mid e_1 \neq e_2$
$R^* t$		$R : \Sigma \rightarrow \Sigma$ $R^* \langle\rangle = \langle\rangle$ $R^*(\langle e\rangle \frown s) = \langle R\ e\ \rangle \frown R^* s \mid e \in \Sigma\ \wedge s \in \Sigma^*$

Table A4
CSP failures semantics.

CSP	Failures semantics
STOP	$\mathcal{F}(STOP) = \{(\langle\rangle, X) \mid X \subseteq \Sigma^{\checkmark}\}$
SKIP	$\mathcal{F}(SKIP) = \{(\langle\rangle, X) \mid X \subseteq \Sigma\} \cup \{(\langle\checkmark\rangle, X) \mid X \subseteq \Sigma^{\checkmark}\}$
$ev \rightarrow P$	$\mathcal{F}(ev \rightarrow P) = \{(\langle\rangle, X) \mid ev \notin X\} \cup \{(\langle ev\rangle \frown s, X) \mid (s, X) \in \mathcal{F}(P)\}$
$P; Q$	$\mathcal{F}(P; Q) = \{(s, X) \mid s \in \Sigma^*\ \wedge (s, X \cup \{\checkmark\}) \in \mathcal{F}(P)\} \cup \{(s \frown t, X) \mid s \frown \langle\checkmark\rangle \in \mathcal{T}(P)\ \wedge (t, X) \in \mathcal{F}(Q)\}$
$P \backslash X$	$\mathcal{F}(P \backslash X) = \{(s \backslash X, Y) \mid (s, X \cup Y) \in \mathcal{F}(P)\} \cup \{(\langle\rangle, X) \mid (\langle\rangle, X) \in \mathcal{F}(P) \cap \mathcal{F}(Q)\}$
$P \Box Q$	$\mathcal{F}(P \Box Q) = \cup\{(\langle\rangle, X) \mid X \subseteq \Sigma\ \wedge \langle\checkmark\rangle \in \mathcal{T}(P) \cup \mathcal{T}(Q)\} \cup \{(s, X) \mid (s, X) \in \mathcal{F}(P) \cup \mathcal{F}(Q)\ \wedge s \neq \langle\rangle\}$
$P \sqcap Q$	$\mathcal{F}(P \sqcap Q) = \mathcal{F}(P) \cup \mathcal{F}(Q)$
if b then P else Q	$\mathcal{F}(if\ \ c\ \ then\ \ P\ \ else\ \ Q) =$ if c then $\mathcal{F}(P)$, else $\mathcal{F}(Q)$
$P[[R]]$	$\mathcal{F}(P[[R]]) = \{(t, R(X)) \mid \exists s.s\ R^*\ t\ \wedge (s, X) \in \mathcal{F}(P)\}$
$P \|\| [X] Q$	$\mathcal{F}(P \parallel [X] Q) = \left\{ \begin{array}{r} (u, Y \cup Z) \mid \exists s, t.(s, Y) \in \mathcal{F}(P) \wedge \\ (t, Z) \in \mathcal{F}(Q) \wedge \\ Y \backslash X^{\checkmark} = Z \backslash X^{\checkmark}\ \wedge u = s \parallel [X] t, \\ \text{where } X^{\checkmark} = X \cup \{\checkmark\} \end{array} \right\}$

Appendix B. $\mathcal{BRIC}$

This appendix details the $\mathcal{BRIC}$ composition rules (Ramos et al., 2009), which are represented in terms of binary and unary asynchronous composition.

When a pair of channels are connected by a $\mathcal{BRIC}$ composition rule, outputs from one will be inputs for the other and vice versa. There are two directional flows of communication. Therefore, two buffers are required to emulate an asynchronous medium between these channels, one for each flow.

Definition 13 ($\mathcal{BRIC}$ buffer)**.** A $\mathcal{BRIC}$ buffer maps inputs to outputs and vice versa (by the relations L and R), without loss or reordering.

$$BUFF^n_{IO}(L,R) \;=\; B^n(L) \;|||\; B^n(R), \text{where}$$

$$B^n(R) \;= B^n_{\langle\rangle}(R) \;= ?x : \text{domR} \rightarrow \text{B}^{\text{n}}_{\langle \text{x} \rangle}$$

$$B^n_s(R) \;= \#s < n \;\&\; ?x : \text{domR} \rightarrow \text{B}^{\text{n}}_{\text{s}^\frown\langle \text{x} \rangle}(\text{R})$$
$$\square\; R(head(s)) \rightarrow B^n_{tail(s)}(R)$$

The asynchronous binary composition (Definition 14) hooks two components, say P and Q, with disjoint communication points, by their respective channels c and z. Instead of communicating directly, their communications are buffered by $BUFF^n_{IO}(R^{c\rightarrow z}_{IO}, R^{z\rightarrow c}_{IO})$.

Definition 14 (Asynchronous binary composition)**.** Let P and Q be two distinct component contracts, and $c \in \mathcal{C}_P$ and $z \in \mathcal{C}_Q$ two channels, such that $\mathcal{C}_P$ and $\mathcal{C}_Q$ are disjoint. Then, the asynchronous binary composition of P and Q, $P_{\langle c\rangle} \approx_{\langle z\rangle} Q$, is given by:

$$P_{\langle c\rangle} \asymp_{\langle z\rangle} Q = \langle \mathcal{B}_P \parallel [\{\!|c|\!\}] \mathcal{B}UFF^n_{IO}(R^{c\rightarrow z}_{IO}, R^{z\rightarrow c}_{IO}) \parallel [\{\!|z|\!\}] \mathcal{B}_Q, \mathcal{R}', \mathcal{I}', \mathcal{C}' \rangle$$

where $\mathcal{C}' = (\mathcal{C}_P \cup \mathcal{C}_Q) \backslash \{c, z\}$, $\mathcal{R}' = \mathcal{C}' \lhd (\mathcal{R}_P \cup \mathcal{R}_Q)$,
$\mathcal{I}' = \text{ran}\mathcal{R}'$ and $R^{a\rightarrow b}_{IO} = \{a.out.x \mapsto b.in.x\}$.

In this composition, the channels c and z are combined such that output events from one channel are consumed by input events of the other, and vice versa. This correspondence is made by two mapping relations, $R^{c\rightarrow z}_{IO}$ and $R^{z\rightarrow c}_{IO}$, which are used to input/output from the buffer $BUFF^n_{IO}$. The resulting component behaviour is that of P synchronised with the buffer $BUFF^n_{IO}(R^{c\rightarrow z}_{IO}, R^{z\rightarrow c}_{IO})$ on c and with Q on z. The interface, $\mathcal{C}'$, of the resulting component, $P_{\langle c\rangle} \approx_{\langle z\rangle} Q$, contains channels of both P and Q except for the hooked channels c and z ($(\mathcal{C}_P \cup \mathcal{C}_Q) \backslash \{c, z\}$). Therefore, only channels in $\mathcal{C}'$ appear in the resulting relation $\mathcal{R}'$ ($S \lhd R$ restricts the domain of R to S) and in the resulting interface $\mathcal{I}'$ ($\text{ran}\mathcal{R}'$).

The asynchronous unary composition (Definition 15) hooks two channels, say c and z of the same component P. It allows P to send and receive information to/from itself. It can be very useful if P has inner components, that must be connected (for example, the dining of philosophers).

Definition 15 (Asynchronous unary composition)**.** Let P be a component contract, and $\{c, z\} \subseteq \mathcal{C}_P$ two of its channels. Then, the asynchronous unary composition of P by hooking c and z, $P \asymp|^{\langle c\rangle}_{\langle z\rangle}$, is given by:

$$P \asymp|^{\langle c\rangle}_{\langle z\rangle} = \langle \mathcal{B}_P \parallel [\{\!|c, z|\!\}] \mathcal{B}UFF^n_{IO}(R^{c\rightarrow z}_{IO}, R^{z\rightarrow c}_{IO}), \mathcal{R}', \mathcal{I}', \mathcal{C}' \rangle$$

where $\mathcal{C}' = \mathcal{C}_P \backslash \{c, z\}$, $\mathcal{R}' = \mathcal{C}' \lhd \mathcal{R}_P$, $\mathcal{I}' = \text{ran}\mathcal{R}'$ and
$R^{a\rightarrow b}_{IO} = \{a.out.x \mapsto b.in.x\}$.

The unary composition, like the binary one, combines two channels c and z such that output events from one channel are consumed by input events of the other, and vice versa; the difference is that both channels are from the same component. The resulting component behaviour is that of P synchronised with the buffer $BUFF^n_{IO}(R^{c\rightarrow z}_{IO}, R^{z\rightarrow c}_{IO})$ on both channels c and z. The resulting component $P \asymp|^{\langle c\rangle}_{\langle z\rangle}$ interface is the same as P except for the hooked channels c and z ($\mathcal{C}_P \backslash \{c, z\}$). As the binary composition, only channels in $\mathcal{C}'$ appear in the resulting relation $\mathcal{R}'$ and in the resulting interface $\mathcal{I}'$.

Not all components can be connected, and some connections using binary or unary compositions can lead to deadlock. The $\mathcal{BRIC}$ rules are defined in terms of the unary and binary asynchronous compositions and define the conditions where components can be safe (deadlock free) connected. They differ by the preconditions and the number of components involved, which are detailed in Ramos et al. (2009). What follows summarises the $\mathcal{BRIC}$ composition rules.

The interleave composition rule is the simplest form of composition. It aggregates two independent components such that, after composition, these components still do not communicate between themselves. They directly communicate with the environment as before, with no interference from each other.

Definition 16 (Interleave composition)**.** Let P and Q be two component contracts, such that $\mathcal{C}_P \cap \mathcal{C}_Q = \emptyset$. The interleave composition of P and Q (namely $P[||||]Q$) is given by:

$$P\ [||||]\ Q = P_{\langle\rangle} \;\asymp\; {}_{\langle\rangle}Q$$

This composition requires P and Q to have disjoint sets of channels ($\mathcal{C}_P \cap \mathcal{C}_Q = \emptyset$). The resulting component $P[||||]Q$ is given by the binary composition (Definition 14) of P and Q without hooking any of its channels ($_{\langle\rangle} \approx {}_{\langle\rangle}$), which implies they run independently, in interleaving.

The second rule, the communication composition represents the most common way for linking channels of two different components. As interleaving, it is given in terms of asynchronous binary composition, but it connects channels from P and Q components. The assembled channels cannot be used in subsequent compositions, as imposed by Definition 14.

Definition 17 (Communication composition)**.** Let P and Q be two component contracts, and ic and oc two communication channels. The communication composition of P and Q (namely $P[ic \leftrightarrow oc]Q$) via ic and oc is defined as follows:

$$P[ic \leftrightarrow oc]Q = P_{\langle ic\rangle} \;\asymp\; {}_{\langle oc\rangle}Q$$

This rule assumes the components behaviours on channels ic and oc are I/O confluent, strong compatible and satisfy the finite output property (FOP). These properties are detailed in Roscoe and Dathi (1987), Roscoe (2006), Ramos (2011): I/O confluence means that choosing between inputs (deterministically) or outputs (nondeterministically) does not prevents other inputs/outputs offered alongside from being communicated afterwards; two processes are strong compatible if all outputs produced by one are consumed by the other, and vice versa; finally, FOP guarantees that a process cannot output forever, so eventually it inputs after a finite sequence of outputs. The resulting component $P_{\langle ic\rangle} \approx {}_{\langle oc\rangle}Q$ is the binary composition of P and Q on channels ic and oc (Definition 14).

The next two compositions allow the link of two channels of a same component by means of asynchronous unary composition. The feedback composition provides the possibility of creating safe cycles for components with a tree topology. It achieves this by, among others conditions, ensuring that the channels being connected are decoupled: the behaviour projection over them are equivalent to the interleaving of each one's projection.

Definition 18 (Decoupled channels)**.** Let $\mathcal{B}$ be an I/O process and $\{c, z\}$ two I/O channels. Then, c and z are decoupled in $\mathcal{B}$ if, and only, if:

$$\mathcal{B} \backslash \Sigma \backslash \{\!|c, z|\!\} \equiv_F (\mathcal{B} \backslash \Sigma \backslash \{\!|c|\!\}) |||(\mathcal{B} \backslash \Sigma \backslash \{\!|z|\!\})$$

Definition 19 (Feedback composition)**.** Let P be a component contract, and ic and oc two communication channels. The feedback composition P ($P[oc \hookrightarrow ic]$) hooking oc to ic is defined as follows:

$$P[oc \hookrightarrow ic] = P \succ\!\!\mid_{\langle oc \rangle}^{\langle ic \rangle}$$

As the communication rule, the feedback composition assumes the component behaviour on channels ic and oc are I/O confluent, strong compatible and satisfy the finite output property (FOP). Moreover, it requires that P behaves on ic and oc independently (as it were two distinct components), i.e., ic and oc are decoupled channel (Ramos, 2011). The resulting component $P \succ\!\!\mid_{\langle oc \rangle}^{\langle ic \rangle}$ is achieved by synchronous unary composition of P on channels ic and oc.

The last composition rule, reflexive, is more general than the feedback composition; it is also more costly regarding verification, since it is able to assemble dependent channels (feedback assembles only independent channels), and so in general it requires a global analysis to ensure deadlock freedom. On the other hand, reflexive composition allows to connect channels in a cyclic topology, whereas feedback is restricted to tree topologies.

Definition 20 (Reflexive composition)**.** Let P be a component contract, and ic and oc two communication channels. The reflexive composition P (namely $P[oc \hookrightarrow ic]$) hooking oc to ic is defined as follows:

$$P[ic \hookrightarrow oc] = P \succ\!\!\mid_{\langle oc \rangle}^{\langle ic \rangle}$$

This last rule relaxes feedback composition restrictions by allowing the connection of non decoupled channels but, nevertheless, it requires that outputs produced by ic are consumed in the same rate (differing by at least one) by inputs from oc, and vice versa, i.e., the behaviour of P is self-injection compatible on the hooked channels (Ramos, 2011).

Appendix C. Proofs

In this appendix we present proofs of some of our lemmas and theorems.

Lemma 1 (cvg implies $ecvg$ on trace prefixing)**.** *Consider two I/O processes T and T' such that, $t_1 \hat{} t_3 \in \mathcal{T}(T)$ and $t' \in \mathcal{T}(T')$. If t' $\texttt{cvg}$ $t_1 \hat{} t_3$, where $t_1 \leq t'$, then t' $\texttt{ecvg}$ $t_1 \hat{} t_3$.*

Proof.

Proof by induction on t_3

Basis step: $t_3 = \langle\rangle$

$t_1 \leq t' \,.\, t'$ $\texttt{cvg}$ $t_1 \hat{} \langle\rangle$

$\equiv$ [empty trace concatenation]

$t_1 \leq t' \,.\, t'$ $\texttt{cvg}$ t_1

$\equiv$ [Definition 5]

$t_1 = t' \,.\, t_1$ $\texttt{cvg}$ t_1

$\equiv$ [Definition 7]

$t_1 = t' \,.\, t_1$ $\texttt{ecvg}$ t_1

Inductive step:

Inductive hypothesis: $t_1 \leq t' \,.\, t'$ $\texttt{cvg}$ $t_1 \hat{} t_3 \Rightarrow t_1 \leq t' \,.\, t'$ $\texttt{ecvg}$ $t_1 \hat{} t_3$

Prove that $t_1 \leq t' \wedge e \in \Sigma \,.\, t'$ $\texttt{cvg}$ $t_1 \hat{} (t_3 \hat{} \langle e \rangle) \Rightarrow t'$ $\texttt{ecvg}$ $t_1 \hat{} (t_3 \hat{} \langle e \rangle)$

$t_1 \leq t' \wedge e \in \Sigma \,.\, t'$ $\texttt{cvg}$ $t_1 \hat{} (t_3 \hat{} \langle e \rangle)$

Cases: e is new input ($e \notin in(T, t_1 \hat{} t_3) \cup out(T, t_1 \hat{} t_3)$) or

e is a possible event after $t_1 \hat{} t_3$ ($t_1 \hat{} (t_3 \hat{} \langle e \rangle) \in \mathcal{T}(T)$)

Case 1: e is a new input

rewrite LHS as $(t_1 \hat{} t_3) \hat{} \langle ne' \rangle \hat{} t_3'$ where $t_3' = \langle\rangle$ and $ne' = e$

$\Rightarrow$ [Definition 5]

t' $\texttt{cvg}$ $(t_1 \hat{} t_3)$

$\Rightarrow$ [by inductive hypothesis]

t' $\texttt{ecvg}$ $t_1 \hat{} t_3$

Case 2: e is a possible event after $t_1 \hat{} t_3$

rewrite LHS as $(t_1 \hat{} t_3) \hat{} \langle e' \rangle \hat{} t_3'$ where $t_3' = \langle\rangle$ and $e' = e$

$\Rightarrow$ [Definition 5]

t' $\texttt{cvg}$ $(t_1 \hat{} t_3) \hat{} \langle e' \rangle$ and $(t_1 \hat{} t_3) \hat{} \langle e' \rangle = t' \Rightarrow t'$ $\texttt{ecvg}$ $(t_1 \hat{} t_3) \hat{} \langle e' \rangle$

□

Lemma 2 ($\texttt{cvg} \subseteq \texttt{ecvg}$)**.** *Consider two I/O processes T and T', and t and t' two of its traces, respectively ($t \in \mathcal{T}(T)$ and $t' \in \mathcal{T}(T')$). If t' $\texttt{cvg}$ t then t' $\texttt{ecvg}$ t.*

Proof.

t' $\texttt{cvg}$ t

$\equiv$ [Definition 5]

$$(t' = t) \vee \left(\begin{array}{c} (\#t' > \#t) \ \wedge \ \exists t_1, t_3 : \Sigma^*, \exists ne : \Sigma \mid \\ \left(\begin{array}{c} t' = t_1 \hat{} \langle ne \rangle \hat{} t_3 \ \wedge \ t_1 \leq t \wedge \\ ne \in inputs \ \wedge ne \notin in(T, t_1) \wedge \\ t_1 \hat{} t_3 \ \ \texttt{cvg} \ \ t \end{array} \right) \end{array} \right)$$

$\Rightarrow$ [predicate calculus]

$$(t' = t) \vee \left(\begin{array}{c} (\#t' > \#t) \ \wedge \ \exists t_1, t_2, t_3 : \Sigma^*, \exists ne \in \Sigma \mid t_2 = \langle\rangle \\ \wedge \left(\begin{array}{c} t' = t_1 \hat{} \langle ne \rangle \hat{} t_2 \hat{} t_3 \ \wedge t_1 \leq t \wedge \\ ne \in inputs \ \wedge ne \notin in(T, t_1) \wedge \\ set(t_2) \cap (in(T, t_1) \cup out(T, t_1)) = \emptyset \wedge \\ t_1 \hat{} t_3 \ \ \texttt{cvg} \ \ t \end{array} \right) \end{array} \right)$$

$\Rightarrow$ [Lemma 1]

$$(t' = t) \vee \left(\begin{array}{c} (\#t' > \#t) \ \wedge \ \exists t_1, t_2, t_3 : \Sigma^*, \exists ne \in \Sigma \mid t_2 = \langle\rangle \\ \wedge \left(\begin{array}{c} t' = t_1 \hat{} \langle ne \rangle \hat{} t_2 \hat{} t_3 \ \wedge t_1 \leq t \wedge \\ ne \in inputs \ \wedge ne \notin in(T, t_1) \wedge \\ set(t_2) \cap (in(T, t_1) \cup out(T, t_1)) = \emptyset \wedge \\ t_1 \hat{} t_3 \ \ \texttt{ecvg} \ \ t \end{array} \right) \end{array} \right)$$

$\equiv$ [Definition 7] t' $\texttt{ecvg}$ t

□

Lemma 3 ($\texttt{io\_cvg} \subseteq \texttt{io\_ecvg}$)**.** *Consider two I/O processes T and T'. If T' $\texttt{io\_cvg}$ T then T' $\texttt{io\_ecvg}$ T.*

Proof.

T' $\texttt{io\_cvg}$ T

$\equiv$ [Definition 6]

$$\forall (t', X) \in \mathcal{F}(T'), \exists (t, Y) \in \mathcal{F}(T) \bullet \left(\begin{array}{c} t' \ \ \texttt{cvg} \ \ t \wedge \\ Y \cap inputs \supseteq X \cap inputs \wedge \\ Y \cap outputs \subseteq X \cap outputs \end{array} \right)$$

$\equiv$ [Lemma 2]

$$\forall (t', X) \in \mathcal{F}(T'), \exists (t, Y) \in \mathcal{F}(T) \bullet \left(\begin{array}{c} t' \ \ \texttt{ecvg} \ \ t \wedge \\ Y \cap inputs \supseteq X \cap inputs \wedge \\ Y \cap outputs \subseteq X \cap outputs \end{array} \right)$$

$\Rightarrow$ [$a \Rightarrow a \vee b$]

$$\forall (t', X) \in \mathcal{F}(T'), \exists (t, Y) \in \mathcal{F}(T) \bullet \left(\begin{array}{c} \left(\begin{array}{c} t' \ \ \texttt{ecvg} \ \ t \wedge \\ Y \cap inputs \supseteq X \cap inputs \wedge \\ Y \cap outputs \subseteq X \cap outputs \end{array} \right) \\ \vee (\Sigma \setminus Y \subseteq X) \end{array} \right)$$

$\Rightarrow$ [Definition 8] T' $\texttt{io\_ecvg}$ T

□

Theorem 2 (Hierarchy)**.** *The relations $\sqsubseteq_{\mathcal{B}}$, $\leftarrow_{cvg}$ and $\leftarrow_{ecvg}$ form a hierarchy: $\sqsubseteq_{\mathcal{B}} \subseteq \leftarrow_{cvg} \subseteq \leftarrow_{ecvg}$. In this proof, assume T and T' are components.*

Proof.

Hypothesis: $\sqsubseteq_{\mathcal{B}} \subseteq \leftarrow_{cvg}$
$T \sqsubseteq_{\mathcal{B}} T'$
$\equiv$ [Definition 12]
$(\mathcal{B}_T \sqsubseteq_F \mathcal{B}_{T'}) \wedge (\mathcal{C}_T = \mathcal{C}_{T'}) \wedge (\forall c : \mathcal{C}_T \bullet \mathcal{R}_T(c) \subseteq \mathcal{R}_{T'}(c))$
$\Rightarrow$ [failures semantics, hiding semantics]
$(\mathcal{B}_T \sqsubseteq_F \mathcal{B}_{T'}) \wedge (\mathcal{C}_T = \mathcal{C}_{T'}) \wedge (\forall c : \mathcal{C}_T \bullet \mathcal{B}_T \upharpoonright c \sqsubseteq_F \mathcal{B}_{T'} \upharpoonright c)$
$\Rightarrow$ [Definition 9]
$(\mathcal{B}_T \sqsubseteq_F \mathcal{B}_{T'}) \wedge (\mathcal{C}_T = \mathcal{C}_{T'}) \wedge (\forall c : \mathcal{C}_T \bullet \mathcal{B}_{T'}$ **def-cong**(c) $\mathcal{B}_T)$
$\Rightarrow$ [Definition 6]
$(\mathcal{B}_{T'}$ io_cvg $\mathcal{B}_T) \wedge (\mathcal{C}_T = \mathcal{C}_{T'}) \wedge (\forall c : \mathcal{C}_T \bullet \mathcal{B}_{T'}$ **def-cong**(c) $\mathcal{B}_T)$
$\Rightarrow$ [Definitions 11] $T \leftarrow_{cvg} T'$
Hypothesis: $\leftarrow_{cvg} \subseteq \leftarrow_{ecvg}$
$T \leftarrow_{cvg} T'$
$\equiv$ [Definition 11]

$$\mathcal{B}_{T'} \text{ io\_cvg } \mathcal{B}_T \wedge \mathcal{R}_T \subseteq \mathcal{R}_{T'} \wedge \forall c : \mathcal{C}_T \bullet \begin{pmatrix} \mathcal{B}_{T'} & \textbf{def-cong}(c) & \mathcal{B}_T & \vee \\ \mathcal{B}_{T'} & \textbf{inp-cong}(c) & \mathcal{B}_T & \end{pmatrix}$$

$\Rightarrow$ [Lemma 3]

$$\mathcal{B}_{T'} \text{ io\_ecvg } \mathcal{B}_T \wedge \mathcal{R}_T \subseteq \mathcal{R}_{T'} \wedge \forall c : \mathcal{C}_T \bullet \begin{pmatrix} \mathcal{B}_{T'} & \textbf{def-cong}(c) & \mathcal{B}_T & \vee \\ \mathcal{B}_{T'} & \textbf{inp-cong}(c) & \mathcal{B}_T & \end{pmatrix}$$

$\Rightarrow$ [Definition 11] $T \leftarrow_{ecvg} T'$

□

Lemma 4 (Inheritance preserves deadlock freedom)**.** *Consider T and T' two $\mathcal{BRIC}$ components, such that T is deadlock free. If $T \leftarrow_{ecvg} T'$ then T' is deadlock free.*

Proof.

$T \leftarrow_{ecvg} T' \wedge T$ is deadlock free
$\equiv$ [Definition 11, deadlock free process]
$\mathcal{B}_{T'}$ io_ecvg $\mathcal{B}_T \wedge \forall s \in \Sigma^* \bullet (s, \Sigma^{\checkmark}) \notin failures(\mathcal{B}_T)$
$\equiv$ [Definition 8]

$$\forall (t', X) \in \mathcal{F}(\mathcal{B}_{T'}), \exists (t, Y) \in \mathcal{F}(\mathcal{B}_T) \bullet \left(\begin{pmatrix} t' \text{ ecvg } t \wedge \\ Y \cap inputs \supseteq X \cap inputs \wedge \\ Y \cap outputs \subseteq X \cap outputs \end{pmatrix} \vee (\Sigma \setminus Y \subseteq X) \right)$$

$\wedge \forall s \in \Sigma^* \bullet (s, \Sigma^{\checkmark}) \notin failures(\mathcal{B}_T)$
$\Rightarrow$ [failures semantics, predicate calculus]

$$\forall (t', X) \in \mathcal{F}(\mathcal{B}_{T'}), \exists (t, Y) \in \mathcal{F}(\mathcal{B}_T) \bullet \left(\begin{pmatrix} (t' \text{ ecvg } t) \wedge (\Sigma^{\checkmark} \notin Y) \wedge \\ Y \cap inputs \supseteq X \cap inputs \wedge \\ Y \cap outputs \subseteq X \cap outputs \end{pmatrix} \vee (\Sigma \setminus Y \subseteq X) \right)$$

$\Rightarrow$ [predicate calculus]
$\forall (t', X) \in \mathcal{F}(\mathcal{B}_{T'}) \bullet (\Sigma^{\checkmark} \notin X)$
$\Rightarrow$ [deadlock freedom]
$\mathcal{B}_{T'}$ is deadlock free process $\Rightarrow T'$ is a deadlock free component

□

Theorem 3 (Substitutability)**.** Let T, T' be two components such that $T \leftarrow_{ecvg} T'$. Consider $S[T]$ a deadlock free component contract, where T is a deadlock free component contract that appears within the context S, then $S[T']$ is deadlock free.

The proof follows by structural induction on the composition operators of $\mathcal{BRIC}$. Assuming it holds for T, we prove that it holds for the following cases: $T[||||]Q$, $T[c \leftrightarrow z]Q$, $T[c \hookrightarrow z]$ and $T[c \overset{\sim}{\hookrightarrow} z]$, where Q is a $\mathcal{BRIC}$ component.

Proof.

Base case
$S[T] = T$
$\Rightarrow$ [$T \leftarrow_{ecvg} T'$, $S[T]$ is deadlock free, Lemma 4]
$S[T'] = T'$ is deadlock free
Interleaving composition
$S[T] = T\ [||||]\ Q$
$\Rightarrow$ [$T \leftarrow_{ecvg} T'$, $S[T]$ is deadlock free, Lemma4]
$\mathcal{C}_{T'} \cap \mathcal{C}_Q = \emptyset \wedge T'$ is deadlock free
$\Rightarrow$ [Q is deadlock free, Definition 16, Theorem 1]
$S[T'] = T'\ [||||]\ Q$ is deadlock free
Communication composition
$T[c \leftrightarrow z]Q$
$\Rightarrow$ [$T \leftarrow_{ecvg} T'$, $S[T]$ is deadlock free, Lemma 4]
$\mathcal{C}_{T'} \cap \mathcal{C}_Q = \emptyset \wedge T'$ is deadlock free
$\Rightarrow$ [Definition 17]
$\mathcal{C}_{T'} \cap \mathcal{C}_Q = \emptyset \wedge T'$ is deadlock free $\wedge$
$\mathcal{B}_{T[c \leftrightarrow z]Q} = \mathcal{B}_T \parallel [\{|c|\}] BUFF^n_{I0}(R^{c \to z}_{I0}, R^{z \to c}_{I0}) \parallel [\{|z|\}] \mathcal{B}_Q$
$\Rightarrow$ [$\mathcal{B}_{T[c \leftrightarrow z]Q}$ is deadlock free, hiding semantics, $R^{c \to z^* t}_{I0}$ Appendix A]
$\mathcal{C}_{T'} \cap \mathcal{C}_Q = \emptyset \wedge T'$ is deadlock free $\wedge$
$\forall t \in \mathcal{T}(\mathcal{B}_T \upharpoonright c) \bullet R^{c \to z^*}_{I0} t \in \mathcal{T}(\mathcal{B}_Q \upharpoonright z) \Rightarrow$

$$\begin{pmatrix} R^{c \to z}_{I0} out(\mathcal{B}_T \upharpoonright c, t) \subseteq in(\mathcal{B}_Q \upharpoonright z, R^{c \to z^*}_{I0} t), \\ R^{z \to c}_{I0} out(\mathcal{B}_Q \upharpoonright z, R^{c \to z^*}_{I0} t) \subseteq in(\mathcal{B}_T \upharpoonright c, t) \end{pmatrix}$$

$\Rightarrow$ [$\mathcal{B}_{T'}$ **def-cong**(c) $\mathcal{B}_T \vee \mathcal{B}_{T'}$ **inp-cong**(c) $\mathcal{B}_T$]
$\mathcal{C}_{T'} \cap \mathcal{C}_Q = \emptyset \wedge T'$ is deadlock free $\wedge$
$\forall t : \Sigma^* \mid t \in \mathcal{T}(T \upharpoonright c) \wedge t \in \mathcal{T}(T' \upharpoonright c) \bullet$
$out(T', t) \subseteq out(T, t) \wedge in(T, t) \subseteq in(T', t) \wedge$
$\mathcal{T}(\mathcal{B}_{T'} \upharpoonright c) \cap \mathcal{T}(\mathcal{B}_Q \upharpoonright z[[R^{z \to c}_{I0}]]) \subseteq \mathcal{T}(\mathcal{B}_T \upharpoonright c) \cap \mathcal{T}(\mathcal{B}_Q \upharpoonright z[[R^{z \to c}_{I0}]])$
$\Rightarrow$ [Definition 17, Theorem 1]
$S[T'] = T'[c \leftrightarrow z]Q$ is deadlock free
Feedback composition
$T[c \hookrightarrow z]$
$\Rightarrow$ [$T \leftarrow_{ecvg} T'$, $S[T]$ is deadlock free, Lemma 4]
$\mathcal{C}_{T'} \cap \mathcal{C}_Q = \emptyset \wedge T'$ is deadlock free
$\Rightarrow$ [Definition 19]
$\mathcal{C}_T \subseteq \mathcal{C}_{T'} \wedge T'$ is deadlock free $\wedge$
$\mathcal{B}_{T[c \hookrightarrow z]} = \mathcal{B}_T \parallel [\{|c, z|\}] BUFF^n_{I0}(R^{c \to z}_{I0}, R^{z \to c}_{I0})$
$\Rightarrow$ [$\mathcal{B}_{T[c \hookrightarrow z]}$ is deadlock free, hiding semantics]
$\mathcal{C}_T \subseteq \mathcal{C}_{T'} \wedge T'$ is deadlock free $\wedge$
$\forall t \in \mathcal{T}(\mathcal{B}_T \upharpoonright c) \bullet R^{c \to z^*}_{I0} t \in \mathcal{T}(\mathcal{B}_T \upharpoonright z) \Rightarrow$

$$\begin{pmatrix} R^{c \to z}_{I0} out(\mathcal{B}_T \upharpoonright c, t) \subseteq in(\mathcal{B}_T \upharpoonright z, R^{c \to z^*}_{I0} t) \\ R^{z \to c}_{I0} out(\mathcal{B}_T \upharpoonright z, R^{c \to z^*}_{I0} t) \subseteq in(\mathcal{B}_T \upharpoonright c, t) \end{pmatrix}$$

$\Rightarrow$ [$\mathcal{B}_{T'}$ **def-cong**(c) $\mathcal{B}_T \vee \mathcal{B}_{T'}$ **inp-cong**(c) $\mathcal{B}_T$]
$\mathcal{C}_T \subseteq \mathcal{C}_{T'} \wedge T'$ is deadlock free $\wedge$
$\forall t : \Sigma^* \mid t \in \mathcal{T}(T \upharpoonright c) \wedge t \in \mathcal{T}(T' \upharpoonright c) \bullet$
$out(T', t) \subseteq out(T, t) \wedge in(T, t) \subseteq in(T', t) \wedge$
$\mathcal{T}(\mathcal{B}_{T'} \upharpoonright c) \cap \mathcal{T}(\mathcal{B}_T \upharpoonright z[[R^{z \to c}_{I0}]]) \subseteq \mathcal{T}(\mathcal{B}_T \upharpoonright c) \cap \mathcal{T}(\mathcal{B}_T \upharpoonright z[[R^{z \to c}_{I0}]])$
$\Rightarrow$ [Definition 19, Theorem 1]
$S[T'] = T'[c \hookrightarrow z]$ is deadlock free
Reflexive composition. It is almost identical to the feedback composition.

□

CRediT authorship contribution statement

José Dihego: Conceptualization, Software, Formal analysis, Investigation, Writing - original draft. **Augusto Sampaio:** Methodology, Validation, Writing - review & editing, Supervision, Project administration. **Marcel Oliveira:** Writing - review & editing, Visualization.

References

de A. Pereira, D.I., Oliveira, M.V.M., Filho, M.S.C., Silva, S.R.D.R., 2017. BTS: A Tool for Formal Component-based Development. In: Polikarpova, N., Schneider, S. (Eds.),

Proceedings of the 13th International Conference on Integrated Formal Methods - IFM 2017. Springer, pp. 211–226.
van der Aalst, W.M.P., van Hee, K.M., van der Toorn, R.A., 2002. Component-based software architectures: a framework based on inheritance of behavior. Sci. Comput. Program. 42 (23), 129171. doi:10.1016/S0167-6423(01)00005-3.
Ackoff, R.L., 1971. Towards a system of systems concepts. Manag. Sci. 17 (11), 661–671. doi:10.1287/mnsc.17.11.661.
America, P., 1991. Designing an object-oriented programming language with behavioural subtyping. In: Proceedings of the REX School/Workshop on Foundations of Object-Oriented Languages. Springer-Verlag, London, UK, pp. 60–90.
Antonino, P.R.G., Sampaio, A., Woodcock, J., 2014. A refinement based strategy for local deadlock analysis of networks of CSP processes. In: FM 2014: Formal Methods, Singapore, May 12–16, 2014. Proceedings, pp. 62–77. doi:10.1007/978-3-319-06410-9_5.
Arbab, F., 2004. Reo: a channel-based coordination model for component composition. Mathematical. Structures in Comp. Sci. 14 (3), 329–366. doi:10.1017/S0960129504004153.
Back, R.J., Sere, K., 1996. Superposition refinement of reactive systems. Formal Aspects Comput. 8 (3), 324–346.
Bauer, S., Mayer, P., Schroeder, A., Hennicker, R., 2010. On Weak Modal Compatibility, Refinement, and the Mio Workbench. In: Esparza, J., Majumdar, R. (Eds.), Tools and Algorithms for the Construction and Analysis of Systems. In: Lecture Notes in Computer Science, 6015. Springer Berlin Heidelberg, pp. 175–189. doi:10.1007/978-3-642-12002-2_15.
Bertrand, N., Jéron, T., Stainer, A., Krichen, M., 2011. Off-line test selection with test purposes for non-deterministic timed automata. In: Proceedings of the 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer-Verlag, Berlin, Heidelberg, pp. 96–111.
van der Bijl, M., Rensink, A., Tretmans, J., 2004. Compositional Testing with Ioco. In: Petrenko, A., Ulrich, A. (Eds.), Formal Approaches to Software Testing. In: Lecture Notes in Computer Science, 2931. Springer Berlin Heidelberg, pp. 86–100. doi:10.1007/978-3-540-24617-6_7.
Bourouz, S., Zeghib, N., 2016. Towards formal checking of web services substitutability. In: Proceedings of the International Conference on Advanced Aspects of Software Engineering (ICAASE), 2016, pp. 1–8.
Bowman, H., Derrick, J., 1999. A Junction between State Based and Behavioural Specification (Invited Talk). Kluwer, B.V., Deventer, The Netherlands, pp. 213–239.
Büchi, M., Sekerinski, E., 1998. Formal Methods for Component Software: The Refinement Calculus Perspective. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 332–337. doi: 10.1007/3-540-69687-3_68
Cavalcanti, A., Hierons, R.M., 2013. Testing with inputs and outputs in CSP. In: Proceedings of the 16th international conference on Fundamental Approaches to Software Engineering. Springer-Verlag, Berlin, Heidelberg, pp. 359–374. doi:10.1007/978-3-642-37057-1_26.
Chen, X., He, J., Liu, Z., Zhan, N., 2007. A model of component-based programming. In: Proceedings of the International Conference on Fundamentals of Software Engineering. Springer-Verlag, Berlin, Heidelberg, pp. 191–206.
Chen, Z., Liu, Z., Ravn, A.P., Stolz, V., Zhan, N., 2009a. Refinement and verification in component-based model-driven design. Sci. Comput. Program. 74 (4), 168–196. doi:10.1016/j.scico.2008.08.003.
Chen, Z., Liu, Z., Ravn, A.P., Stolz, V., Zhan, N., 2009b. Refinement and verification in component-based model-driven design. Sci. Comput. Program. 74 (4), 168–196. doi:10.1016/j.scico.2008.08.003.
Dihego, J., Antonino, P., Sampaio, A., 2013. Algebraic Laws for Process Subtyping. In: Groves, L., Sun, J. (Eds.), Formal Methods and Software Engineering. In: Lecture Notes in Computer Science, 8144. Springer Berlin Heidelberg, pp. 4–19. doi:10.1007/978-3-642-41202-8_2.
Dihego, J., Sampaio, A., Oliveira, M., 2015. Constructive extensibility of trustworthy component-based systems. In: Proceedings of the 30th Annual ACM Symposium on Applied Computing. ACM, New York, NY, USA, pp. 1808–1814. doi:10.1145/2695664.2695916.
Enderton, H.B., 1977. Elements of Set Theory. Academic Press, New York.
Filho, M.C., Oliveira, M., Sampaio, A., Cavalcanti, A., 2018. Compositional and local livelock analysis for CSP. Inf. Process. Lett. 133, 21–25. doi:10.1016/j.ipl.2017.12.011.
Filho, M.S.C., Oliveira, M.V.M., Sampaio, A., Cavalcanti, A., 2016. Local Livelock Analysis of Component-Based Models. In: Ogata, K., Lawford, M., Liu, S. (Eds.), Formal Methods and Software Engineering - 18th International Conference on Formal Engineering Methods, ICFEM 2016. Springer International Publishing, pp. 279–295. doi:10.1007/978-3-319-47846-3_18.
García-Duque, J., Pazos-Arias, J.J., López-Nores, M., Blanco-Fernández, Y., Fernández-Vilas, A., Díaz-Redondo, R.P., Ramos-Cabrer, M., Gil-Solla, A., 2009. Methodologies to evolve formal specifications through refinement and retrenchment in an analysisrevision cycle. Requir. Eng. 14 (3), 129153. doi:10.1007/s00766-009-0074-z.
Gibson-Robinson, T., Armstrong, P.J., Boulgakov, A., Roscoe, A.W., 2014. FDR3 - A modern refinement checker for CSP. In: Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, 2014, Grenoble, France, April 5–13, 2014. Proceedings, pp. 187–201. doi:10.1007/978-3-642-54862-8_13.
Hennicker, R., Janisch, S., Knapp, A., 2010. Refinement of components in connection-safe assemblies with synchronous and asynchronous communication. In: Proceedings of the 15th Monterey Conference on Foundations of Computer Software: Future Trends and Techniques for Development. Springer-Verlag, Berlin, Heidelberg, pp. 154–180. doi:10.1007/978-3-642-12566-9_9.
Hoare, C., He, J., 1998. Unifying Theories of Programming. Prentice-Hall.
ISOBE, Y., ROGGENBACH, M., 2008. Csp-prover: a proof tool for the verification of scalable concurrent systems. Comput. Softw. 25 (4).
Jifeng, H., Li, X., Liu, Z., 2005. Component-based software engineering. In: Proceedings of the Second International Conference on Theoretical Aspects of Computing. Springer-Verlag, Berlin, Heidelberg, pp. 70–95. doi:10.1007/11560647_5.
Jifeng, H., Li, X., Liu, Z., 2006. Rcos: a refinement calculus of object systems. Theor. Comput. Sci. 365, 109–142. doi:10.1016/j.tcs.2006.07.034. Formal Methods for Components and Objects Formal Methods for Components and Objects
Krasner, G.E., Pope, S.T., 1988. A cookbook for using the model-view controller user interface paradigm in smalltalk-80. J. Object Oriented Program. 1 (3), 26–49.
Kurki-Suonio, R., 1999. Component and Interface Refinement in Closed-System Specifications. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 134–154. doi: 10.1007/3-540-48119-2_10
Lange, A., Atkinson, C., 2019. On the rules for inheritance in Iml. In: Proceedings of the ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C), pp. 113–118.
Liskov, B., 1987. Keynote address - data abstraction and hierarchy. SIGPLAN Not. 23 (5), 17–34. doi:10.1145/62139.62141.
Liskov, B.H., Wing, J.M., 1994. A behavioral notion of subtyping. ACM Trans. Program. Lang. Syst. 16 (6), 1811–1841. http://doi.acm.org/10.1145/197320.197383.
Liu, Z., Morisset, C., Stolz, V., 2009. rCOS: theory and tool for component-based model driven development. In: Proceedings of the Third IPM International Conference on Fundamentals of Software Engineering. Springer-Verlag, Berlin, Heidelberg, pp. 62–80. doi:10.1007/978-3-642-11623-0_3.
Maddox, J., Long, Y., Rajan, H., 2018. Large-scale study of substitutability in the presence of effects. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. Association for Computing Machinery, New York, NY, USA, p. 528538. doi:10.1145/3236024.3236075.
Meng, S., Barbosa, L.S., 2006. Components as coalgebras: the refinement dimension. Theor. Comput. Sci. 351 (2), 276–294. doi:10.1016/j.tcs.2005.09.072.
Nipkow, T., Wenzel, M., Paulson, L.C., 2002. Isabelle/HOL: A Proof Assistant for Higher-Order Logic. Springer-Verlag, Berlin, Heidelberg.
OliveiraM. and Sampaio, A. and Antonino,P. and Dihego, J. and Filho, M. C. and Bryans, J., 2014. Compositional Analysis and Design of CML Models. Technical Report. Comprehensive Modelling for Advanced Systems of Systems.
Otoni, R., Cavalcanti, A., Sampaio, A., 2017. Local analysis of determinism for csp. In: Cavalheiro, S., Fiadeiro, J. (Eds.), Formal Methods: Foundations and Applications. Springer International Publishing, Cham, pp. 107–124.
Papazoglou, M.P., Georgakopoulos, D., 2003. Introduction: service-oriented computing. Commun. ACM 46 (10), 24–28. doi:10.1145/944217.944233.
Puntigam, F., 1996. Types for Active Objects Based on Trace Semantics. In: Proceedings FMOODS 96. Chapman and Hall, pp. 4–19.
Ramos, R., Sampaio, A., Mota, A., 2009. Systematic development of trustworthy component systems. In: Proceedings of the 2nd World Congress on Formal Methods. Springer, pp. 140–156.
Ramos, R.T., 2011. Systematic Development of Trustworthy Component-based Systems. Center of Informatics - Federal University of Pernambuco, Brazil Ph.D. thesis.
Roscoe, A.W., 1998. Theory and Practice of Concurrency. Prentice-Hall Series in Computer Science. Prentice-Hall.
Roscoe, A.W., 2006. Confluence thanks to extensional determinism. Electron. Notes Theor. Comput. Sci. 162, 305–309.
Roscoe, B., Dathi, N., 1987. The pursuit of deadlock freedom. Inf. Comput. 75 (3), 289–327.
Rubinger, A.L., Burke, B., 2010. Enterprise javabeans 3.1. O'Reilly Media, Inc..
Szyperski, C., 1998. Component Software: Beyond Object-Oriented Programming. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA.
Tretmans, J., 1996. Test generation with inputs, outputs and repetitive quiescence. Softw. Concepts Tools 17 (3), 103–120.
Wang, Z., Wang, H., Zhan, N., 2009. Towards a theory of refinement of component-based systems. Report, 427. UNU-IIST.
Wegner, P., Zdonik, S.B., 1988. Inheritance as an incremental modification mechanism or what like is and isn't like. In: Proceedings of the European Conference on Object-Oriented Programming. Springer-Verlag, London, UK, pp. 55–77.
Wehrheim, H., 2002. Checking behavioural subtypes via refinement. In: Jacobs, B., Rensink, A. (Eds.), Formal Methods for Open Object-Based Distributed Systems V. Springer US, Boston, MA, pp. 79–93.
Wehrheim, H., 2003. Behavioral subtyping relations for active objects. Form. Methods Syst. Des. 23 (2), 143–170. doi:10.1023/A:1024764232069.

José Dihegoda Silva Oliveira is a Professor of the Bahia Federal Technical Institute (IFBA). He holds a Bachelor's Degree in Computer Science from the Federal University of Pernambuco (2008), a Master's Degree in Computer Science from the Federal University of Pernambuco (2011) and a Ph.D. in Computer Science from the Federal University of Pernambuco (2016). Currently, he is a member of the 's Graduate Program in Information Systems and was Coordinator of the Technical Courses at the IFBA. José Dihego has experience in the area of Computer Science, with emphasis on Formal Methods and Programming Languages. More specifically, his research has focused on computation, refinement and inheritance. Recently, he is researching on using formal methods to improve mobile development and architecture. He has taught the disciplines of Analysis and Specification, Web programming, Object-oriented programming and Mobile Development.

Augusto Sampaio is a graduate of Centro de Informtica (CIn) at Universidade Federal de Pernambuco (UFPE) (B.Sc. 1985 and M.Sc. 1988) and Oxford Ph.D., 1993. From 1995 to 2007 he was an Associate professor at CIn-UFPE, and since 2007 he is a Software Engineering Full Professor at the same institution. In 2010 he received the title of Comendador da Ordem Nacional do MĘrito Cientđfico (Commander of the National Order of Scientific Merit) awarded by the Brazilian President. He holds a scholarship as Productive Researcher from CNPq (the Brazilian National Research Agency), since 1995. As a result of his Ph.D. thesis, he contributed an innovative approach to compilation based on algebraic transformations; the work has been published as a book in 1997. A joint work with C.A.R Hoare and He Jifeng resulted in establishing a formal link between algebraic and operational semantics. Linking theories and tools, model and program analysis and transformation, and model based testing are among his major research topics. He has coordinated several national and international projects, including three projects jointly funded byNSF (USA) and CNPq (Brazil). More recently, he was the Brazilian coordinator (Principal Investigator - PI) of the COMPASS project (http://www.compass- research.eu), funded by the European Commission under an FP7 call. He has also significantly focused on cooperation with industry: particularly, since 2002 he is the PI of a long term collaboration with Motorola, on the development and application of formal testing and analysis techniques to mobile phone applications. This programme is part of what became known as the Brazil Test Center, which has attracted a significant amount in funding over the years, with the participation of more than a thousand collaborators, including students, engineers and researchers. It is responsible for the test of most applications that run on Motorolas mobile phones, worldwide. The current infrastructure includes some 4G testing Labs that are unique in the south hemisphere. The Brazil Test Center has received several prizes for excellence in research and development and for innovation. Concerning academic qualifications in the context of this project, Augusto Sampaio has had a major role in the conception and implementation of a novel educational programme that is now widely known in Brazil as Software Residency; this gives young Software Engineers the kind of experience that medical doctors obtain with a residency in a hospital. The course involves a theoretical background and hands-on training, at an industry site, and has been extensively exercised in the context of the cooperation with Motorola, where the emphasis was on a specialised background on Software Testing. As a result, more than 500 students have graduated, from all over Brazil, and the course model has been awarded a Prize on Software Quality and Productivity by the Brazilian Ministry of Science, Technology and Innovation; this has inspired subsequent calls from CNPq to spread the model throughout the country. Since 2006, Augusto Sampaio has also established a cooperation project with Embraer, both on testing and on qualitative and quantitative analysis of models of avionics applications. He has co-organised two international summer schools that gave rise to two books in Springer LNCS; one was on Testing Techniques in Software Engineering, where he has contributed both as co-editor and as co-author of one of the chapters. Augusto Sampaio has been participating in the PC of the main national and international conferences related to formal methods (FM, SEFM, ICFEM, ICTAC, IFM, FMi, ICTSS, ISOLA, UTP, VSTTE. SBMF, SBES, among others), and has participated in panels and delivered calls as invited speakers; particularly related to the theme of the current project, he was a keynote at ICFEM-2010 and gave invited talks at Motorola Labs at Basingstoke (2006), at the Oxford University Computing Lab (2009), Kent University (2013), Universidade do Minho (2015) and the British Computer Society Provably Correct Systems workshop (2015). He is a member of the Editorial Board of Formal Aspects of Computing (Springer), one of the major journals of the field. He constantly reviews papers for journals, including Theoretical Computer Science, Science of Computer Programming, Acta Informatica, Software Testing Verification and Reliability, Journal of The Brazilian Computer Society, among others. He is a member of the Steering Committee of ICTAC since 2004 and is the PC chair for ICTAC 2016, in Taiwan. Since 2013 he is a member of the Technical Committee 1 (TC1) of IFIP. Since 2014 he is also a member of the Awards Committee for Formal Methods Europe, responsible for two awards: the FME fellowship for distinguished service to the formal methods community; and the FM Symposium's most-influential paper. He has supervised more than 40 graduate students (Ph.D. and M.Sc.) and has published more than 100 journal and conference papers. At CIn-UFPE, Augusto Sampaio has been Vice-Coordinator for Undergraduate Studies (1996–1998), Vice-Coordinator for Graduate Studies (1998–2001), Coordinator for Graduate Studies (2001–2003), Research Coordinator (2005–2013), and since 2006 he is the Head of the Computer Science Department (one of the three departments in the center for informatics at CIn-UFPE); also, since 2013 he is the Coordinator for Industry- Academia Cooperation. Augusto Sampaio has also contributed to shape the policies for education and research in Computer Science as the Planning Director of the Brazilian Computer Society, and as a member of the expert committees of the Brazilian Research Councils (CNPq and CAPES) that evaluate all the higher-education institutions in Brazil and the major applications for funding. Augusto Sampaio is one of the most distinguished Brazilian Computer Scientists in the area of Software Engineering with emphasis on Formal Methods; he was one of the founders of the Brazilian Commission for Formal Methods, and has continuously contributed to consolidate the conference organized by this commission (SBMF) with international standards. SBMF is already a well-established conference in the Formal Methods international community, and since 2010 its proceedings are published by Springer, in the LNCS series. He has also significantly contributed with the Commission for Software Engineering, and the associated conference (SBES).

Marcel Vinicius Medeiros Oliveira is a Reader of the Department of Informatics and Applied Mathematics (DIMAp) of the Federal University of Rio Grande do Norte (UFRN). He holds a Bachelor's Degree in Computer Science from the Federal University of Pernambuco (2000), a Master's Degree in Computer Science from the Federal University of Pernambuco (2002) and a Ph.D. in Computer Science from the University of York, England (2006). Currently, he is a member of the National Institute of Software Engineering, a member of the Special Committee on Formal Methods of the Brazilian Computer Society, a member of the UFRN's Graduate Program in Systems and Computing (PPgSC) and Coordinator of the Technical Courses at the Institute Digital Metropolis of UFRN with about 2000 students. At Institute Digital Metropolis, he is a collaborating member, in addition to being a member of the Academic Development Committee, Deputy Director of Teaching and Vice-Coordinator of SETE, Center for Integration, Research and Innovation in Software Engineering of the Institute. Marcel Oliveira has experience in the area of Computer Science, with emphasis on Formal Methods. More specifically, his research has focused on computation and refinement tactics, competition, semantics of formal languages, integration of formal methods, and synthesis of code from formal specifications. He has taught the disciplines of Database, Applied Logic to Software Engineering and Formal Methods.