Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 312 (2015) 69–88

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Enabling Synchronous and Asynchronous Communications in CSP for SOC

Abeer S. Al-Humaimeedy[†,‡] [1] Maribel Fernández [†2]

[†] *Department of Informatics, King's College London, UK*
[‡] *Department of Information Technology, King Saud University, Saudi Arabia*

Abstract

Service Oriented Computing (SOC) is based on service composition, that is, loosely coupled autonomous heterogeneous services, which are collectively composed to implement a particular task. We develop a new calculus for SOC within the framework of CSP process algebra, aiming to improve the verification techniques and to enhance the expressiveness of SOC calculi. This paper presents the part of the calculus that extends CSP with built-in buffers to facilitate direct asynchronous communications. We provide the operational semantics of the calculus and extend the FDR (a CSP model checker) by implementing functions for asynchronous communications.

Keywords: asynchrony, CSP, asynchronous calculi, buffers, SOC calculi.

1 Introduction

The Service Oriented Computing (SOC) paradigm, which is based on service composition, has been successfully applied to facilitate systems integration in distributed environments. Service composition refers to an aggregate of loosely coupled autonomous heterogeneous services, which are collectively composed to implement a particular task.

In the SOC paradigm, services can communicate using messages solely, where messages can be sent synchronously or asynchronously. In the web services (WS) standards [18,25], messages are sent according to predefined *interaction patterns* [9]. These interaction patterns include the *request-response* pattern and the *solicit-response* pattern, which represent synchronous communication. In synchronous communication, the initiator of the message suspends processing until it receives a response. On the other hand, the *one-way* and *notification* patterns represent

[1] Email: Abeer@ksu.edu.sa
[2] Email: Maribel.Fernandez@kcl.ac.uk

http://dx.doi.org/10.1016/j.entcs.2015.04.005

asynchronous communication, where the initiator of the message continues running the next coded statement, without suspending processing.

Process calculi, such as SCC [5] and CaSPiS [6], provide a formal specification of interaction patterns with synchronous communications, allowing designers to reason about the correctness of SOC systems. Other formal systems, such as COWS [16] and Conversation Calculus [24], consider only asynchronous communications, since synchronous communications in the internet standards are usually implemented by network protocols such as TCP/IP [12], which are by default asynchronous. To the best of our knowledge, none of the formal calculi developed for SOC supports both synchronous and asynchronous communications.

In this paper, we propose a new process calculus, called CSPa, that supports mixed synchronous and asynchronous communications. Our calculus can be described as a buffered version of Hoare's CSP [13]. CSP has been chosen as the foundation for our calculus because its communication model supports mixed synchronous/interleaving communications. In CSP, all processes participating in a parallel composition synchronise on a predefined set of events. For instance, let p, q, r be CSP processes, then in the parallel composition $(p \parallel_{\{|a,b|\}} q \parallel_{\{|a,b|\}} r)$, the processes p, q, and r synchronise on events a, b only, the rest of the events are evaluated independently. This version of the parallel composition is called generalised parallel composition. In CSP, the parallel composition has another version, called alphabetised parallel composition [20] and written $p \,{}_{\alpha p}\|_{\alpha q}\, q$, where participants synchronise in all shared events. However, the alphabetised parallel composition can be encoded in the generalised parallel composition by setting the interface set to be $(\alpha p \cap \alpha q)$. Here, αp denotes p's alphabet which is the set of events that process p can perform. In this paper, we consider the generalised parallel composition only. Therefore, from now on when we write parallel composition we mean the generalised parallel composition.

The novelty in CSPa is the introduction of *implicit* buffers, which are used in the channel semantics to facilitate asynchronous communications in a transparent way. In other words, CSPa includes asynchronous communication primitives, which rely on buffers, but designers do not need to create, maintain or terminate buffers. We provide an operational semantics that explains how buffers work.

Additionally, we study the relationship between our calculus and CSP, and we encode our calculus in the original CSP. Although the new constructs in CSPa do not enhance the expressiveness of CSP in the sense that the new constructs can be encoded in CSP, CSPa simplifies reasoning by replacing the encoding transitions by one transition.

In previous work, we extended CSP with primitives to model dynamic compensations [1]. Here we focus on communication primitives solely: we enhance CSP by allowing asynchronous communications in addition to its standard mixed synchronous/interleaving communications. More precisely, we extend the parallel composition operator in order to permit mixed synchronous, interleaving and asynchronous communications. In future work, we plan to include also primitives to create and maintain sessions and to model mobility, to obtain an expressive calcu-

lus for SOC modelling.

We observe that although SOC calculi are equipped with formal semantics, which enables formal verification of properties, a feature not available in the WS standards, these standards are still the dominant modelling languages. In our view, this is mostly because formal methods are often perceived as hard to apply. To facilitate the practical use of our calculus we have encoded CSPa in the CSP model checker (FDR [11]). FDR implements the mathematical machinery and the theory of refinement that Hoare built for reasoning on the external behaviour of systems. It provides simple proof techniques to assert the conformance between specification and implementation, deadlock-freedom, and divergence-freedom, in addition to determinism and bisimulations (we illustrate some of these features in the example in Section 6). Other model checkers, such as SPIN [14], which accepts models in Promela, or Maude [2], could be used if these checks are implemented. In the model checker PAT [23] the refinement checks are implemented, but we choose FDR because CSP# (the input language of PAT) does not support the generalised parallel composition operator which we extend in this paper.

Paper overview: Section 2 recalls some basic CSP notions. Section 3 introduces CSPa. Section 4 discusses the relationship between CSP and CSPa. Section 5 presents the implementation of CSPa in FDR. Section 6 illustrates the usability of this extension with an example. Section 7 discusses related work. Finally, Section 8 concludes and discusses future directions.

2 Preliminaries

We assume that the reader is familiar with the core theory of CSP. Below we briefly recall its syntax and operational semantics. For more details we refer the reader to [20]; our notation is drawn largely from it.

In the rest of the paper, the following notations are used: $p, q, \ldots$ denote processes; Σ is the universal set that contains all the observable events in a system; a, b are used to range over this set. Σ^{τ} (resp., $\Sigma^{\surd}$) is the universal set Σ and in addition the silent event τ (resp., the termination event $\surd$). $\Sigma^{\tau\surd}$ is the set $\Sigma \cup \{\tau, \surd\}$. Capital letters A, B denote sets of observable events; s denotes lists, and $\langle x \rangle$ denotes a list which has the element x.

CSP processes are defined by the following grammar.

$$p, q ::= a \rightarrow p \mid p \Box q \mid p \sqcap q \mid p \|_A q \mid p; q \mid p \backslash A \mid p[\![R]\!] \mid \mu p.f(p) \mid p|||q \mid SKIP \mid STOP$$

where a can be the name of an atomic action, inputting through channel a (written as $a?x$), outputting through channel a (written as $a!x$), or a combination of them (e.g. $a!x?y$). The interleaving ($|||$) and the parallel composition ($\|_A$) operators have indexed versions written as $|||_{i=1}^{N}$ and $\|_{A_{i=1}}^{N}$ respectively.

The operational semantics is given in Figure 1, where $\xrightarrow{a}$, $\xrightarrow{a.x}$, $\xrightarrow{\tau}$, $\xrightarrow{\surd}$ denote a labelled transition relation. Here, a represents an atomic action, and $a.x$ represents inputting or outputting through channel a where x can be variable or data; both a

and $a.x$ are considered events in Σ. $STOP$ is the process which does nothing. $SKIP$ is the process that immediately terminates successfully. *Prefixing* $(a \rightarrow p)$ is the process which is ready to engage in event a and then behave as p. *External choice* $(p \Box q)$ is resolved by the environment offering the first action of p or q. *Internal choice* $(p \sqcap q)$ is resolved internally; thus, either alternative can be available after an internal action τ. *Recursive functions* $(\mu p.f(p))$ are defined using an explicit fixed point notation, where $f(p)$ is a CSP term involving process p, and $\mu p.f(p)$ defines exactly the same process as $p = f(p)$. The law of recursion here is that the recursively defined process $\mu p.f(p)$ satisfies the equation defining it, i.e., $\mu p.f(p) = f[\mu p.f(p)/p]$ (details of the fixed point recursion in CSP can be found in [20]). In *Hiding* $(p \backslash a)$, action a is hidden in process p that is the external environment can not observe it. If R is a relation which maps events in set A to the events in set B, then *Renaming* $(p[\![R]\!])$ is mapping events from A to B in process p. In *Sequential Composition* $(p; q)$ q is executed if p terminates successfully. In *Parallel Composition* $(p \|_A q)$ p and q are executed in parallel and synchronise on events in A only; events not in A are interleaved. The *Parallel Composition* does not require processes to synchronise on the terminal event $\surd$. Therefore, if p or q are ready to terminate successfully, i.e. evaluating the event $\surd$, then the process can terminate and then the parallel composition will evolve to an intermediate state called Ω until both of the processes terminate then the parallel composition will terminate successfully; this is called *distributed termination*. In *Interleaving* $(p|||q)$ the events from processes p, q can be performed in any order.

Apart from the prefix operator where different rules are provided for a and $a.x$, the a event in Figure 1 rules can be a or $a.x$ where x is a variable or a value. In rule (par3), since the whole $a.x$ is an event, $a.x$ can only be synchronised with $a.y$ if the values of x and y are equal or if one or both of them are input variables.

In this paper, we assume that standard functions on *lists* and *conditional if-then-else* are available and executed as τ transitions.

We recall below the definitions of the weak transition [21], bisimilarity [21], and similarity [17] relations.

Definition 2.1 [Weak transition]

(i) $\Longrightarrow$ is the reflexive and transitive closure of $\xrightarrow{\tau}$ (i.e. $\Longrightarrow$ is $\xrightarrow{\tau}{}^*$). Thus, $p \Longrightarrow p'$expresses that p can evolve to p' by performing zero or more internal actions.

(ii) $\stackrel{a}{\Longrightarrow}$ is $\Longrightarrow\xrightarrow{a}\Longrightarrow$, for $a \in \Sigma$. Thus, $p \stackrel{a}{\Longrightarrow} p'$expresses that p can evolve to p' by performing the visible action a with any number, possibly zero, of internal actions before and after a.

(iii) $\Longrightarrow^*$ denotes a sequence of 0 or more steps using $\Longrightarrow$ or $\stackrel{a}{\Longrightarrow}$.

Definition 2.2 [Bisimilarity] The *bisimilarity* relation, denoted by $\approx$, is the largest symmetric relation such that whenever $p \approx q$, then:

(i) If $p \xrightarrow{a} p'$, then $q \stackrel{a}{\Longrightarrow} q'$ and $p' \approx q'$.

$$\text{(skip)}\ \frac{}{SKIP \xrightarrow{\surd} STOP} \qquad \text{(prefix)}\ \frac{}{(a \to p) \xrightarrow{a} p}\ a \in \Sigma$$

$$\text{(prefix-out)}\ \frac{}{(a!x \to p) \xrightarrow{a.x} p}\ a.x \in \Sigma \qquad \text{(prefix-in)}\ \frac{}{(a?x \to p) \xrightarrow{a.v} p[v/x]}\ a.v \in \Sigma$$

$$\text{(exch1,2)}\ \frac{p \xrightarrow{a} p'}{p \,\Box\, q \xrightarrow{a} p'} \quad \frac{q \xrightarrow{b} q'}{p \,\Box\, q \xrightarrow{b} q'}\ (a, b \in \Sigma^{\surd}) \qquad \text{(exch3,4)}\ \frac{p \xrightarrow{\tau} p'}{p \,\Box\, q \xrightarrow{\tau} p' \,\Box\, q} \quad \frac{q \xrightarrow{\tau} q'}{p \,\Box\, q \xrightarrow{\tau} p \,\Box\, q'}$$

$$\text{(inch1,2)}\ \frac{}{p \sqcap q \xrightarrow{\tau} p} \quad \frac{}{p \sqcap q \xrightarrow{\tau} q} \qquad \text{(rec)}\ \frac{}{\mu p.f(p) \xrightarrow{\tau} f[\mu p.f(p)/p]}$$

$$\text{(hid1,2,3)}\ \frac{p \xrightarrow{a} p'}{p \setminus A \xrightarrow{a} p' \setminus A}\ (a \notin A) \quad \frac{p \xrightarrow{a} p'}{p \setminus A \xrightarrow{\tau} p' \setminus A}\ (a \in A) \quad \frac{p \xrightarrow{\surd} STOP}{p \setminus A \xrightarrow{\surd} STOP}$$

$$\text{(rem1,2,3)}\ \frac{p \xrightarrow{a} p'}{p[\![R]\!] \xrightarrow{b} p'[\![R]\!]}\ (a\, R\, b) \quad \frac{p \xrightarrow{\tau} p'}{p[\![R]\!] \xrightarrow{\tau} p'[\![R]\!]} \quad \frac{p \xrightarrow{\surd} STOP}{p[\![R]\!] \xrightarrow{\surd} STOP}$$

$$\text{(seq1,2)}\ \frac{p \xrightarrow{a} p'}{p\,;q \xrightarrow{a} p'\,;q}\ (a \in \Sigma^{\tau}) \quad \frac{p \xrightarrow{\surd} p'}{p\,;q \xrightarrow{\tau} q} \qquad \text{(intv1,2)}\ \frac{p \xrightarrow{a} p'}{p \,|||\, q \xrightarrow{a} p' \,|||\, q} \quad \frac{q \xrightarrow{a} q'}{p \,|||\, q \xrightarrow{a} p \,|||\, q'}\ (a \in \Sigma^{\tau\surd})$$

$$\text{(par1,2,3)}\ \frac{p \xrightarrow{a} p'}{p \parallel_A q \xrightarrow{a} p' \parallel_A q} \quad \frac{q \xrightarrow{a} q'}{p \parallel_A q \xrightarrow{a} p \parallel_A q'}\ (a \notin A) \quad \frac{p \xrightarrow{a} p' \quad q \xrightarrow{a} q'}{p \parallel_A q \xrightarrow{a} p' \parallel_A q'} \quad (a \in A)$$

$$\text{(parT1,2,3)}\ \frac{p \xrightarrow{\surd} STOP}{p \parallel_A q \xrightarrow{\tau} \Omega \parallel_A q} \quad \frac{q \xrightarrow{\surd} STOP}{p \parallel_A q \xrightarrow{\tau} p \parallel_A \Omega} \quad \frac{}{\Omega \parallel_A \Omega \xrightarrow{\surd} STOP}$$

Figure 1. CSP's operational semantics

(ii) If $p \xrightarrow{\tau} p'$, then $q \Longrightarrow q'$ and $p' \approx q'$.

We say two processes p and q are bisimilar if $p \approx q$.

Definition 2.3 [Similarity] The *similarity* relation, denoted by $\preceq$, is the largest relation such that whenever $p \preceq q$, then:

(i) If $p \xrightarrow{a} p'$, then $\exists q'.\, q \overset{a}{\Longrightarrow} q'$ and $p' \preceq q'$.

(ii) If $p \xrightarrow{\tau} p'$, then $\exists q'.\, q \Longrightarrow q'$ and $p' \preceq q'$.

We say that process q simulates process p if $p \preceq q$.

3 Mixed Synchronous/Asynchronous Communications in CSPa

In CSP, the parallel composition operator does not force events to be synchronised. Instead, it is parameterised by an interface set, which governs the synchronisation between participants. Events inside the interface set should be simultaneously evaluated, whereas events outside the set can be evaluated independently even if they are shared. Evaluating events independently does not pass channels' data from inputting to outputting processes. If designers want asynchronicity (i.e., data to be transmitted with delay), they need to define and maintain an explicit buffer. To avoid this burden, we include built-in buffers in CSPa, associated with events in Σ, and extend the semantics of CSP to model asynchronous communications. In this way, asynchronicity becomes a primitive notion in the calculus, at the same level as synchronicity, and designers do not need to deal with the implementation details of creating and maintaining buffers.

To implement this solution, we extend CSP's channel syntax with two events: $a!{<}x$, which denotes adding data x as the last element in a's buffer (B_a), and

$a?>x$, which denotes the consumption of the first element in a's buffer (B_a). We also extend the transition relation as follows: The prefix $a?>x$ in a process indicates that this process is ready to accept any value of a's type (i.e the type of data that channel a can accept). This input transition rule is similar to CSP's input rule except that the label here is $a{\leftarrow}.v$ instead of $a.v$. The $a{\leftarrow}.x$ label can be read as input x into channel a (inputting into a channel means outputting from the buffers).

$$\text{(asy-in)} \quad \frac{}{a?>x \rightarrow p \xrightarrow{a\leftarrow.v} p[v/x]}$$

If the event $a!<x$ is used within a process, then this process is ready to send x. Here x can be a value or a variable of a's type. This output transition rule is similar to CSP's output rule except that the label here is $a{\rightarrow}.x$ instead of $a.x$. The $a{\rightarrow}.x$ can be read as output the value x from channel a (outputting from a channel means inputting into the buffers).

$$\text{(asy-out)} \quad \frac{}{a!<x \rightarrow p \xrightarrow{a\rightarrow x} p}$$

The symbols $>$ and $<$ in these events denote a communication with B_a, where B_a denotes the built-in buffer for channel a. B_a appears in the semantics but it is transparent for designers. B_a is formally defined as follows:

Definition 3.1 [Event Buffer] For each $a \in \Sigma$, an unbounded buffer B_a with FIFO (first come first out) policy is defined as a process parameterised by a list s, as indicated below. We write $B_a(\langle\rangle)$ to represent an empty buffer, $B_a(\langle x\rangle^\frown s)$ represents a buffer containing an element x followed by the elements in s. The operator $^\frown$ represents list concatenation.

$$\begin{aligned} B_a(s) = \; & if \quad null(s) \\ & then \;\; (a?<x \rightarrow B_a(\langle x\rangle))\Box SKIP \\ & else \;\; (a!>head(s) \rightarrow B_a(tail(s)) \;\Box\; a?<x \rightarrow B_a(s^\frown\langle x\rangle)) \\ & \quad \Box \;\; SKIP \end{aligned}$$

where $null(s)$, $head(s)$, and $tail(s)$ are standard functions on lists, which check if a list is empty, retrieve the first element in a list, and return all elements except the first element in a list respectively.

In the definition of B_a, $a!>x$ and $a?<x$ are the complementary events of $a?>x$ and $a!<x$ and are reserved for buffers only, that is, they cannot be used by designers. They are defined as follows:

$$\text{(buffer-in)} \quad \frac{}{a?<x \rightarrow p \xrightarrow{a\rightarrow v} p[v/x]}$$

where data is accepted from processes which evaluate $a!<x$. $a?<x$ and $a!<x$ will be synchronised using the original parallel composition of CSP.

$$\text{(buffer-out)} \quad \frac{}{a!>x \rightarrow p \xrightarrow{a\leftarrow x} p}$$

where data is consumed by processes which evaluate $a?>x$. $a!>x$ and $a?>x$ will be synchronised using the original parallel composition of CSP.

To allow these new events to be evaluated by the parallel composition, rules (par1-par3) in Figure 1, and the other CSP operators in Figure 1, we create the set $\mathcal{B}$ to include the event types: $a \leftarrow .x$ and $a \rightarrow .x$ as follows:

$$\mathcal{B} = \{a\leftarrow, a\rightarrow |\ a \in\ channels(\Sigma)\}$$

where $channels(\Sigma)$ is the function which extracts the channel name from events.

After that, we update Σ to become $\Sigma \cup \mathcal{B}$. Thus, the original operator of the CSP can now evaluate the new types of events.

According to Definition 3.1, an unbounded buffer has three options: (i) to input data unconditionally; (ii) to output data if the buffer is not empty; (iii) to terminate if the system terminates.

In designing B_a, FIFO policy is implemented by attaching new data to the end of the buffer list and processes always consume the first element in the list.

The intuition behind introducing implicit buffers is to introduce delay between sending and receiving messages. This will allow processes to continue their execution without waiting for the receiver to get the message.

In two-way communications (between two processes) if buffers are introduced in the middle of a communication then the sent message from the first process will be stored in the buffer until the receiver process is ready to get the message. However, in multi-way communications (multiple processes communicate) as in CSP, we should explain what asynchronicity means. In our model, we want to retain the CSP model of multi-way communications with the addition of asynchronicity. In multi-way communications, buffered channels may introduce non-determinism, as Example 3.2 shows.

Example 3.2 Consider the system

$$a?>x \rightarrow SKIP \parallel a?>y \rightarrow SKIP \parallel a!<3 \rightarrow SKIP$$

where the processes are not synchronised (i.e. interleaving); we write $\parallel$ for parallel composition with an empty synchronisation set.

According to our model the communications can happen in any order. If the output is done first then the value 3 will be stored in B_a then retrieved by the input event. However, which input event will get the value from the buffer ($a?>x$ or $a?>y$) is not specified. Either $a?>x$ gets the value and $a?>y$ will be waiting for a new value, or the other way around.

In CSPa, we can avoid such non-determinism by synchronising between input events or output events and the buffers, as shown in the example below.

Example 3.3 Let the following communications take place in a system:

$a?>x \rightarrow SKIP \parallel_{\{a?>\}} a?>y \rightarrow SKIP \parallel a!<3 \rightarrow SKIP$

Assuming the buffer is empty, then according to our model the value 3 will

be stored in B_a then retrieved by the input event. Here, the inputs events are synchronised, therefore $a?>x$ and $a?>y$ should get the same value (which is 3).

Consequently, in CSPa, two kinds of asynchronicity are available:

(i) Synchronous communication, but with a delay between inputting and outputting values. This can be achieved by synchronising on input events and output events. Thus, if buffers are empty, then all output events happen at the same time and upload data to buffers, then all input events get the same value from the buffers. Note that, CSPa adheres to CSP synchronisation rules, where the whole a.n event is considered in synchronisation. For instance, in CSP, an event $a.3$ can only be synchronised with $a.3$ or $a.x$ if x is an input variable (in the latter case, x will be substituted by 3).

A less strict synchronous communications can be achieved by synchronising on input events only (as Example 3.3) or output events.

(ii) Interleaving communication, with values not lost but stored in the buffers. However, in this case the order of execution of buffer's events is non-deterministic, as Example 3.2 demonstrates.

The event buffers can be considered as an area of a shared memory which can be accessed by all processes. We define below a process, which we call *buffered*-Σ to be the parallel composition of all Σ's event buffers.

Definition 3.4 [Buffered-Σ] The process *buffered*-Σ is the parallel composition of all the individual event buffers, that is, $B_\Sigma = |||_{i\in\Sigma} B_i$. The interleaving operators are used to emphasise that event buffers do not synchronise on inputting or outputting. The process *buffered*-Σ evolves by performing transitions ($\Longrightarrow^*$) to new states: $B'_\Sigma, B''_\Sigma, \ldots$

To allow the buffered-Σ to work with the whole system we will install it in parallel with the system as a preprocessing step, thus allowing the execution of the system in asynchronous mode. The buffered-Σ will work in parallel with the whole system and synchronise on all events which have $\rightarrow$ or $\leftarrow$. The buffered-Σ is installed using a parallel composition operator due to the fact that communications between processes take place only if they are working in parallel.

Definition 3.5 [Buffered System] If p is a CSPa process then the *buffered process* associated with p is $p\|_{\{a\leftarrow, a\rightarrow | a\in\Sigma\}} B_\Sigma$.

We use the silent action τ to represent the preprocessing step where the system is placed in parallel with the buffered-Σ. This is safe as long as the externally visible behaviour of the system is unchanged by the addition of an extra starting state with a τ action, where this system has no option but to take this invisible action and behave like a buffered system. Additionally, this will conceal the effect of adding buffers to a system.

Finally, it is important to ensure that buffers do not introduce non-termination, i.e., if the system terminates then buffers should terminate as well. To achieve this, we introduce a new termination method for CSPa processes, namely *synchronised*

termination, which replaces *distributed termination* in CSP. In *synchronised termination*, CSPa processes synchronise on evaluating the event $\surd$, i.e., termination. Therefore, if processes terminate then the buffered-Σ will be forced to terminate as well, see Proposition 3.8 for details (this feature is also significant in other parts of our calculus, to deal with compensations and sessions). The following rule implements the *synchronised termination*:

$$\text{(parST)}\ \frac{p \xrightarrow{\surd} STOP \quad q \xrightarrow{\surd} STOP}{p \parallel_A q \xrightarrow{\surd} STOP}$$

Note that, rule (parST) replaces rules (parT1,2,3) in Figure 1.

We devote the rest of this section to prove that our buffered system is simulated by the original system. That is, our buffered system does not introduce new transitions which the original system cannot do.

Theorem 3.6 *For every $p \in CSPa$, $(p \parallel_{\{a\leftarrow, a\rightarrow \mid a\in\Sigma\}} B'_\Sigma) \preceq p$, where B'_Σ is any state of the process B_Σ : $B_\Sigma \Longrightarrow^* B'_\Sigma$.*

Proof To prove simulation according to Definition 2.3 we need to consider in turn the transitions performed by the buffered system and match them with the transition performed by process p.

According to Definitions 3.1 and 3.4, the buffered system can do the following transitions:

- The buffered process performs τ as follows:
  $(p \parallel_{\{a\leftarrow, a\rightarrow \mid a\in\Sigma\}} B'_\Sigma) \xrightarrow{\tau} (p \parallel_{\{a\leftarrow, a\rightarrow \mid a\in\Sigma\}} B''_\Sigma)$ by rule (rec,par1)
  The process p does not change, therefore: $p \Longrightarrow p$
- The buffered process performs $a \leftarrow .x$ as follows:
  $(p \parallel_{\{a\leftarrow, a\rightarrow \mid a\in\Sigma\}} B'_\Sigma) \xrightarrow{a\leftarrow .x} (p' \parallel_{\{a\leftarrow, a\rightarrow \mid a\in\Sigma\}} B''_\Sigma)$ by rule (exch1,par3)
  Then ,this transition is matched by the process p in the following way: $p \xrightarrow{a\leftarrow .x} p'$
- The buffered process performs $a \rightarrow .x$ as follows:
  $(p \parallel_{\{a\leftarrow, a\rightarrow \mid a\in\Sigma\}} B'_\Sigma) \xrightarrow{a\rightarrow .x} (p' \parallel_{\{a\leftarrow, a\rightarrow \mid a\in\Sigma\}} B''_\Sigma)$ by rule (exch1,par3)
  Then, this transition is matched by the process p in the following way: $p \xrightarrow{a\rightarrow .x} p'$
- The buffered process performs $\surd$ as follows:
  $(p \parallel_{\{a\leftarrow, a\rightarrow \mid a\in\Sigma\}} B'_\Sigma) \xrightarrow{\surd} STOP$ by rule (exch1,parST)
  This is because, according to Definition 3.1 and 3.4, buffered-Σ should evaluate and synchronise on $\surd$ if the environment offers this event and terminates.
  Then, this transition is matched by the process p in the following way:
  $p \xrightarrow{\surd} STOP$

□

3.1 Direction, Deadlock, and Termination

In CSP, the unit of data can be divided into several components (e.g. $ItemID.ItemQ$) and these components, in synchronous mode, can be simultaneously conveyed in both directions (e.g. $a?ItemID!ItemQ$, where $ItemID$ is re-

ceived and $ItemQ$ is transmitted at the same time on channel a). In CSPa, similar to [13,20], channels have a definite direction, where the whole unit of data is loaded or consumed in one direction at a time (e.g. $a?ItemID.ItemQ$,$a!ItemID.ItemQ$).

Adding buffers to a calculus can significantly affect the performance of systems and introduce extra design errors. For instance, a system may end up with a deadlock if an output cannot be evaluated because a channel's buffer is full. To avoid introducing deadlocks, Hoare suggests these buffers to be unbounded. We have followed Hoare's approach here.

Thanks to rule (parST), the buffered Σ, which has been installed in parallel with the system, will terminate if the system terminates. This is due to the termination option that is activated externally by the environment.

Definition 3.7 [Terminated Process] A terminated process is a process which cannot evolve more (i.e. can not make any transition). In CSPa, as the original CSP, a process p is terminating if there exists $p \xrightarrow{a_0} p_1 \xrightarrow{a_1} \ldots \xrightarrow{a_k} p_n \xrightarrow{a_n} STOP$ where $n \geqslant 0$ and $STOP$ is the primitive process which can not make any transition.

If $a_n = \surd$ then this process successfully terminated, otherwise the process is blocked with no further transitions.

Note 1 *In CSP, if a process cannot evolve more, then it is equivalent to STOP; see [20] for details.*

Proposition 3.8 *Buffers do not introduce deadlock or non-terminating sequences of transitions, that is: (i) If the buffered-Σ terminates successfully then p terminates successfully. (ii) If channel buffers are not empty when inputs are made, then the buffered-Σ blocks only if p blocks. (iii) If the buffered-Σ has an infinite steps of transitions then p also has it.*

Proof We prove the proposition cases as follows:

(i) If the buffered-Σ terminates successfully then the process p also terminates successfully. This is because the $\surd$ events are observable in the environment if the process p successfully terminates. The $\surd$ events will resolve the external choice with the $SKIP$ option as Definition 3.1 shows, using rule (parST).

(ii) If channel buffers are not empty, That is the system will not block because there is no data to retrieve, then as a consequence of Theorem 3.6, if $p \| _{\{a\leftarrow, a\rightarrow | a\in\Sigma\}} B_\Sigma$ blocks or has infinite sequence of transitions, then p has this block or infinite sequence of transitions.

□

4 The Relationship Between CSPa and CSP

In this section, we first discuss the encoding of CSPa into CSP, and then we reason on the correctness of our encoding by proving that the encoded processes are weakly bisimilar to the original processes of our calculus (Theorem 4.3). This results show that CSPa does not strictly enhance the expressiveness of CSP in the sense that the new constructs in CSPa can be encoded in CSP, however, CSPa simplifies the

specification and verification of asynchronous communications, in the context of CSP, by providing explicit primitives for modelling them.

Thanks to the encoding, we can use the denotational models of CSP to reason on the correctness of CSPa systems. Informally speaking, the main differences between CSPa and CSP are the hidden communications with the buffered-Σ and the synchronised termination.

To facilitate the encoding we define the new termination signal *term* which will be used to enforce synchronisation when processes terminate. This termination signal will be hidden later in the system.

Moreover, we encode B_a (defined in Definition 3.1) to be the following CSP process:

$$\begin{aligned} B_a(s) = \ & if \quad null(s) \\ & then \ ((a{\leftarrow}.x \longrightarrow B_a(\langle x \rangle)) \Box (term \rightarrow SKIP)) \\ & else \ ((a{\rightarrow}.head(s) \longrightarrow B_a(tail(s)) \Box (a{\leftarrow}.x \longrightarrow B_a(s^\frown\langle x \rangle))) \\ & \Box \ (term \rightarrow SKIP)) \end{aligned}$$

where $null(s)$, $head(s)$, and $tail(s)$ are standard functions on lists, which check if a list is empty, retrieve the first element in a list, and return all elements except the first element in a list, respectively.

We also encode B_Σ to be the following CSP process: $B_\Sigma = |||_{i \in \Sigma} B_i$.

We define the encoding from CSPa into CSP as follows:

Definition 4.1 The encoding $[.] : CSPa \rightarrow CSP$ is defined as: $[.] = [\![p]\!] \backslash \{term\}$, where $[\![p]\!]$ is defined homomorphically except for the following:

$$\begin{aligned} [\![SKIP]\!] &= term \rightarrow SKIP \\ [\![a?> \rightarrow p]\!] &= a{\leftarrow}.v \rightarrow [\![p[v/x]]\!] \\ [\![a!< \rightarrow p]\!] &= a{\rightarrow}.x \rightarrow [\![p]\!] \\ [\![p \|_A q]\!] &= ([\![p]\!] \|_{\{term\} \cup A} [\![q]\!]) \end{aligned}$$

Lemma 4.2 *If $q = p\backslash A$ then:*

(i) *if $\sigma \notin A$ then $p \xrightarrow{\sigma} p' \Leftrightarrow q \xrightarrow{\sigma} q'$, where $q' = p'\backslash A$.*

(ii) *if $\sigma \in A$ then $p \xrightarrow{\sigma} p' \Rightarrow q \xrightarrow{\tau} q'$, where $q' = p'\backslash A$. Also, $q \xrightarrow{\tau} q'$ implies either $p \xrightarrow{\sigma} p'$ for $\sigma \in A$ or $p \xrightarrow{\tau} p'$.*

Proof To prove (i), we first prove if $\sigma \notin A$ then $p \xrightarrow{\sigma} p' \Rightarrow q \xrightarrow{\sigma} q'$ and $q' = p'\backslash A$. If $\sigma \notin A$ then σ has two cases:

(i) If σ is of the form $a, a{\leftarrow}, a{\rightarrow}$ or τ, then according to rule (hid1) if $p \xrightarrow{\sigma} p'$ and $\sigma \notin A$ then $q \xrightarrow{\sigma} q'$ and $q' = p'\backslash A$.

(ii) If $\sigma = \surd$, then according to rule (hid3) if $p \xrightarrow{\surd} STOP$ then $q \xrightarrow{\surd} STOP$, which implies that $q' = STOP$, however, in CSP, $STOP\backslash A = STOP$ (see [20]), therefore, $q' = STOP\backslash A$.

Secondly, we prove that if $\sigma \notin A$ then $q \xrightarrow{\sigma} q' \Rightarrow p \xrightarrow{\sigma} p'$ where $q' = p'\backslash A$.

If $\sigma \notin A$ then σ has two cases:

(i) If σ is of the form $a, a\leftarrow, a\rightarrow$ or τ, then by rule (hid1) if $q = p\backslash A \xrightarrow{\sigma} p'\backslash A$, this implies that $p \xrightarrow{\sigma} p'$.

(ii) If $\sigma = \surd$, then by rule (hid3) if $p\backslash A \xrightarrow{\surd} STOP$, this implies that $p \xrightarrow{\surd} STOP$. However, in CSP, $STOP\backslash A = STOP$ (see [20]), therefore, $q' = STOP\backslash A$.

Thus, if $\sigma \notin A$ then $p \xrightarrow{\sigma} p' \Leftrightarrow q \xrightarrow{\sigma} q'$ where $q' = p'\backslash A$.

To prove (ii), we first prove that if $\sigma \in A$ then $p \xrightarrow{\sigma} p' \Rightarrow q \xrightarrow{\tau} q'$ and $q' = p'\backslash A$, which is a direct result of rule (hid2).

Secondly, we prove that $q \xrightarrow{\tau} q' \Rightarrow p \xrightarrow{\sigma} p'$, for $\sigma \in A$, where $q' = p'\backslash A$, or $p \xrightarrow{\tau} p'$.

$q \xrightarrow{\tau} q'$ can take place in two cases:

(i) If $p \xrightarrow{\tau} p'$ by rule (hid1), then we are done.

(ii) If $p \xrightarrow{\sigma} p'$ and $\sigma \in A$ by rule (hid2).

□

Theorem 4.3 *Let* $[.] : CSPa \rightarrow CSP$ *be the encoding in Definition 4.1. For every* $p \in CSPa$, $p \approx [p]$.

Proof We give the proof only for the four non-homomorphic cases of the encoding. The homomorphic ones follow trivially.

We will show that the above four cases of the encoding are weakly bisimilar to their source processes. These four cases are described as follows:

(i) $SKIP$

We prove that: if $SKIP \xrightarrow{\surd} STOP$, then $[SKIP] \xrightarrow{\tau}{}^{*} \xrightarrow{\surd} STOP$.

The transition $SKIP \xrightarrow{\surd} STOP$ follows by rule (skip) and is the only transition for $SKIP$.

This transition is matched by the encoded process in the following way: $[SKIP] \xrightarrow{\tau}\xrightarrow{\surd} STOP$ by rules (prefix,hid2,skip).

In the other direction, the only possible transition for $[SKIP]$ is $[SKIP] \xrightarrow{\tau} SKIP$ by rules (prefix,hid2).

This transition is matched by: $SKIP \Longrightarrow SKIP$.

Therefore, according to Definition 2.2 $SKIP \approx [SKIP]$.

(ii) $a?{>}x \rightarrow p$

We prove that: if $a?{>}x \rightarrow p \xrightarrow{a\leftarrow v} p[v/x]$, then $[a?{>}x \rightarrow p] \xrightarrow{a\leftarrow v} [p[v/x]]$.

The only possible transition for $(a?{>}x \rightarrow p)$ is: $(a?{>}x \rightarrow p) \xrightarrow{a\leftarrow v} p[v/x]$ by rule (asy-in).

This transition is matched by the encoded process in the following way: $[a?{>}x \rightarrow p] \xrightarrow{a\leftarrow v} [p[v/x]]$ by rules (prefix-in,hid1).

In the other direction, the only possible transition for $[a?{>}x \rightarrow p]$ is: $[a?{>}x \rightarrow p] \xrightarrow{a\leftarrow v} [p[v/x]]$ by rules (prefix-in,hid1).

This transition is matched by the CSPa process in the following way:

$(a?{>}x \to p) \xrightarrow{a \leftarrow .v} p[v/x]$ by rule (asy-in).

Therefore, according to Definition 2.2 $(a?{>}x \to p) \approx [a?{>}x \to p]$

(iii) $a!{<}x \to p$

We prove that: if $a!{<}x \to p \xrightarrow{a \rightarrow .x} p$, then $[a!{<}x \to p] \xrightarrow{a \rightarrow .x} [p]$.

The only possible transition for $(a!{<}x \to p)$ is: $(a!{<}x \to p) \xrightarrow{a \rightarrow .x} p$ by rule (asy-out).

This transition is matched by the encoded process in the following way: $[a!{<}x \to p] \xrightarrow{a \rightarrow .x} [p]$ by rules (prefix-out,hid1).

In the other direction, the only possible transition for $[a!{<}x \to p]$ is: $[a!{<}x \to p] \xrightarrow{a \rightarrow .x} [p]$ by rules (prefix-out,hid1).

This transition is matched by the CSPa process in the following way: $(a!{<}x \to p) \xrightarrow{a \rightarrow .x} p$ by rule (asy-out).

Therefore, according to Definition 2.2 $(a?{>}x \to p) \approx [a?{>}x \to p]$

(iv) $p \|_A q$

We prove that if $\sigma \neq \surd$ then $p\|_A q \xrightarrow{\sigma} r$ implies $[p\|_A q] \xrightarrow{\sigma} [r]$, otherwise, $p\|_A q \xrightarrow{\surd} r$ implies $[p\|_A q] \xrightarrow{\tau}{}^{*} \xrightarrow{\surd} [r]$.

Let first consider the possible transitions for $(p\|_A q)$:

- If $p \xrightarrow{\sigma} p'$ and $\sigma \notin A$ then $(p\|_A q) \xrightarrow{\sigma} (p'\|_A q)$ by rule (par1).

  This transition is matched by the encoded process in the following way:

  We know that, if $[p] \xrightarrow{\sigma} [p']$ then $[\![p]\!] \xrightarrow{\sigma} [\![p']\!]$ by Lemma 4.2.

  This implies that, if $[\![p]\!] \xrightarrow{\sigma} [\![p']\!]$ and $\sigma \notin A$ then $([\![p]\!]\|_{\{term\}\cup A}[\![q]\!]) \setminus \{term\} \xrightarrow{\sigma} ([\![p']\!]\|_{\{term\}\cup A}[\![q]\!]) \setminus \{term\}$ by rules (par1,hid1).
- If $q \xrightarrow{\sigma} q'$ and $\sigma \notin A$ then $(p\|_A q) \xrightarrow{\sigma} (p\|_A q')$ by rule (par2).

  This transition is matched by the encoded process in the following way:

  We know that, if $[q] \xrightarrow{\sigma} [q']$ then $[\![q]\!] \xrightarrow{\sigma} [\![q']\!]$ by Lemma 4.2.

  This implies that, if $[\![q]\!] \xrightarrow{\sigma} [\![q']\!]$ and $\sigma \notin A$ then $([\![p]\!]\|_{\{term\}\cup A}[\![q]\!]) \setminus \{term\} \xrightarrow{\sigma} ([\![p]\!]\|_{\{term\}\cup A}[\![q']\!]) \setminus \{term\}$ by rules (par2,hid1).
- If $p \xrightarrow{\sigma} p'$ and $q \xrightarrow{\sigma} q'$ then $(p\|_A q) \xrightarrow{\sigma} (p'\|_A q')$ if $\sigma \in A$ by rule (par3).

  This transition is matched by the encoded process in the following way:

  We know that, if $[p] \xrightarrow{\sigma} [p']$ and $[q] \xrightarrow{\sigma} [q']$ then $[\![p]\!] \xrightarrow{\sigma} [\![p']\!]$ and $[\![q]\!] \xrightarrow{\sigma} [\![q']\!]$ by Lemma 4.2.

  This implies that, if $[\![p]\!] \xrightarrow{\sigma} [\![p']\!]$ and $[\![q]\!] \xrightarrow{\sigma} [\![q']\!]$ then $([\![p]\!]\|_{\{term\}\cup A}[\![q]\!]) \setminus \{term\} \xrightarrow{\sigma} ([\![p']\!]\|_{\{term\}\cup A}[\![q']\!]) \setminus \{term\}$ if $\sigma \in A$ by rules (par3,hid1).
- If $p \xrightarrow{\surd} STOP$ and $q \xrightarrow{\surd} STOP$ then $(p\|_A q) \xrightarrow{\surd} STOP$ by rule (parST).

  This transition is matched by the encoded process in the following way:

  We know that, if $[p] \xrightarrow{\tau}{}^{*} \xrightarrow{\surd} STOP$ and $[q] \xrightarrow{\tau}{}^{*} \xrightarrow{\surd} STOP$ then $[\![p]\!] \xrightarrow{term} \xrightarrow{\tau}{}^{*} \xrightarrow{\surd} STOP$ and $[\![q]\!] \xrightarrow{term} \xrightarrow{\tau}{}^{*} \xrightarrow{\surd} STOP$ by Lemma 4.2.

  This implies that, if $[\![p]\!] \xrightarrow{term} \xrightarrow{\tau}{}^{*} \xrightarrow{\surd} STOP$ and $[\![q]\!] \xrightarrow{term} \xrightarrow{\tau}{}^{*} \xrightarrow{\surd} STOP$ then $([\![p]\!]\|_{\{term\}\cup A}[\![q]\!]) \setminus \{term\} \xrightarrow{\tau}$

$(SKIP \|_{\{term\}\cup A} SKIP)\setminus\{term\} \xrightarrow{\tau}\xrightarrow{\tau}\xrightarrow{\surd} STOP$ by rules (par3,hid2,parT1,hid1,parT2,hid1,parT3,hid3).

In the other direction, since $[p\|_A q] = (\llbracket p\rrbracket \|_{\{term\}\cup A} \llbracket q\rrbracket)\setminus\{term\}$ by Definition 4.1, the possible transitions for $[p\|_A q]$ are:

- If $\sigma \notin \{term\}$ and $[p] \xrightarrow{\sigma} [p']$, then by Lemma 4.2 if $[p] \xrightarrow{\sigma} [p']$ then $\llbracket p\rrbracket \xrightarrow{\sigma} \llbracket p'\rrbracket$.
  This implies that, if $\llbracket p\rrbracket \xrightarrow{\sigma} \llbracket p'\rrbracket$ then $(\llbracket p\rrbracket \|_{\{term\}\cup A} \llbracket q\rrbracket)\setminus\{term\} \xrightarrow{\sigma} (\llbracket p'\rrbracket \|_{\{term\}\cup A} \llbracket q\rrbracket)\setminus\{term\}$ where $\sigma \notin A$ by rules (par1,hid1).
  This transition is matched by the CSPa process in the following way:
  If $p \xrightarrow{\sigma} p'$ then $(p\|_A q) \xrightarrow{\sigma} (p'\|_A q)$ if $\sigma \notin A$ by rule (par1).
- If $\sigma \notin \{term\}$ and $[q] \xrightarrow{\sigma} [q']$ then by Lemma 4.2 if $[q] \xrightarrow{\sigma} [q']$ then $\llbracket q\rrbracket \xrightarrow{\sigma} \llbracket q'\rrbracket$.
  This implies that, if $\llbracket q\rrbracket \xrightarrow{\sigma} \llbracket q'\rrbracket$ then $(\llbracket p\rrbracket \|_{\{term\}\cup A} \llbracket q\rrbracket)\setminus\{term\} \xrightarrow{\sigma} (\llbracket p\rrbracket \|_{\{term\}\cup A} \llbracket q'\rrbracket)\setminus\{term\}$ if $\sigma \notin A$ by rules (par2,hid1).
  This transition is matched by the CSPa process in the following way:
  If $q \xrightarrow{\sigma} q'$ then $(p\|_A q) \xrightarrow{\sigma} (p\|_A q')$ if $\sigma \notin A$ by rule (par2).
- If $\sigma \notin \{term\}$, $[p] \xrightarrow{\sigma} [p']$ and $[q] \xrightarrow{\sigma} [q']$ then by Lemma 4.2 if $[p] \xrightarrow{\sigma} [p']$ and $[q] \xrightarrow{\sigma} [q']$ then $\llbracket p\rrbracket \xrightarrow{\sigma} \llbracket p'\rrbracket$ and $\llbracket q\rrbracket \xrightarrow{\sigma} \llbracket q'\rrbracket$
  This implies that, if $\llbracket p\rrbracket \xrightarrow{\sigma} \llbracket p'\rrbracket$ and $\llbracket q\rrbracket \xrightarrow{\sigma} \llbracket q'\rrbracket$ then $(\llbracket p\rrbracket \|_{\{term\}\cup A} \llbracket q\rrbracket)\setminus\{term\} \xrightarrow{\sigma} (\llbracket p'\rrbracket \|_{\{term\}\cup A} \llbracket q'\rrbracket)\setminus\{term\}$ if $\sigma \in A$ by rules (par3,hid1).
  This transition is matched by the encoded process in the following way:
  If $p \xrightarrow{\sigma} p'$ and $q \xrightarrow{\sigma} q'$ then $(p\|_A q) \xrightarrow{\sigma} (p'\|_A q')$ if $\sigma \in A$ by rule (par3).
- If $\sigma \in \{term\}$, $\sigma \neq \tau$, $[p] \xrightarrow{\tau} [p']$ and $[q] \xrightarrow{\tau} [q']$ then by Lemma 4.2 if $[p] \xrightarrow{\tau} SKIP$ and $[q] \xrightarrow{\tau} SKIP$ then $\llbracket p\rrbracket \xrightarrow{term} SKIP$ and $\llbracket q\rrbracket \xrightarrow{term} SKIP$
  This implies that, if $\llbracket p\rrbracket \xrightarrow{term} SKIP$ and $\llbracket q\rrbracket \xrightarrow{term} SKIP$ then $(\llbracket p\rrbracket \|_{\{term\}\cup A} \llbracket q\rrbracket)\setminus\{term\} \xrightarrow{\tau} (SKIP \|_{\{term\}\cup A} SKIP)\setminus\{term\}$ by rules (par3,hid2).
  This transition is matched by: $(SKIP \|_{\{term\}\cup A} SKIP)\setminus\{term\} \Rightarrow (SKIP \|_{\{term\}\cup A} SKIP)\setminus\{term\}$
  Therefore, according to Definition 2.2 $(p\|_A q) \approx [p\|_A q]$

□

5 Implementing CSPa in FDR

We provide, in this section, an implementation of the buffered model of CSPa in FDR [11] which is the model checker of CSP. FDR uses CSP_M [22] as the input language. CSP_M is the machine readable version of CSP. Encoding CSPa into CSP_M allows CSPa users to use other CSP tools, like the trace animator ProBE [10].

To implement CSPa in FDR we use the encoding statements (listed in Definition 4.1). We also encode B_Σ to be the CSP process defined in Section 4. We use *in* and *out* keywords in front of channel names to encode $a \leftarrow$ and $a \rightarrow$ events respectively. Consequently, $in.a?x$ and $out.a!x$ represent $a?>x$ and $a!<x$ events respectively in the theoretical model.

```
bufferedEvents =  let
bf(a,s) = if null(s) then ( (out.a?x -> bf(a,<x>)) [] SKIPP )
                     else ((in.a!head(s) -> bf(a,tail(s))[]
                          #s<N & (out.a?x -> bf(a,s^<x>))) [] SKIPP)
                  within (||| x:ev @ bf(x,<>))
```

According to our model this buffer should be created for all events in Σ. However, to reduce the memory consumption we create this buffer for events in *ev* only, where *ev* is the set of asynchronous events provided by the user. If the user is happy to include all events then set *Sigma* can be used. All the buffers start empty.

In our implementation, we prefer to design the buffered-Σ to be bounded and for a set of events instead of the whole Σ. This is to reduce the consumption of memory and accelerate the assertion time in FDR. Although these limits are not requirements of FDR we prefer to give the user the ability to accelerate the assertion time and reduce memory consumption.

The preprocessing step in our theoretical model, which installs the buffered-Σ, is encoded in our implementation by the function *asy*. This function puts the system process in parallel with the buffered-Σ.

```
asy(p)= p  [| {|in.a , out.a, term | a<-ev |} |]   bufferedEvents
```

To deal with termination, users should use $SKIPP$ instead of $SKIP$ to evaluate the new terminal signal *term* first, and use *par* function instead of $\|_A$ to enforce synchronisation on successful terminations of processes. Synchronisation is done through *term*.

```
SKIPP = term -> SKIP
par(p,A,q)= p  [| union({|term|},A) |]  q
```

If a system terminates the buffers will be forced to terminate due to the external choice with $SKIPP$. This choice is resolved by the environment. Thus, the buffered-Σ will terminate as soon as the system terminates. This will avoid the problem of dangling buffers when the system terminates.

6 Example

In this section, we provide an example of a simple ordering system to demonstrate how the asynchronous communications can be used.

A simple ordering system for a company requires departments to send an order of their needed item with the quantity needed to the purchasing department. The purchasing department then sets the connection to the internet and sends the order to the supplier as shown in Figure 2.

Assuming the data media is reliable in the company, we design the communications inside the company to be done in a synchronous mode to ensure fast exchange of information between the company's departments (e.g. the event *porder*). However, if the department communicates with other organisations outside the company the communication is done in an asynchronous mode assuming the media is unreli-

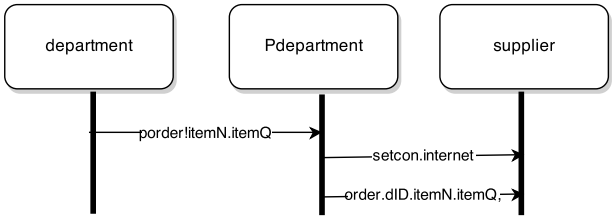

Figure 2. Ordering system state diagram

able (e.g. the event *order*). Events which do not need any mode of synchronisation can be done in interleaving mode (e.g. the event *setcon*).

This system can be written in CSPa as follows:

$$\begin{aligned} department &= porder!itemN.itemQ \longrightarrow department \\ Pdepartment &= porder?itemN.itemQ \longrightarrow setcon.internet \longrightarrow \\ &\quad order!{<}dID.itemN.itemQ \longrightarrow Pdepartment \\ supplier &= setcon.internet \longrightarrow order?{>}dID.iN.iQ \longrightarrow supplier \\ system &= (department \|_{\{|porder|\}} Pdepartment) ||| supplier \end{aligned}$$

6.1 Evaluating the example using FDR

In this section, we reason on the correctness of our example by encoding it in our implementation. In this way, we can verify properties such as deadlock freedom, divergence freedom, or check if a process refines another.

In our implementation, the example should be written as follows:

```
department = porder!2.15 -> department
Pdepartment = porder?itemN.itemQ  -> setcon.internet ->
out.order!1.itemN.itemQ  ->  Pdepartment
supplier= setcon.internet -> in.order?dID.iN.iQ ->  supplier

sys =  par( par(department, {|porder|}, Pdepartment), {}, supplier)
system = asy(sys)
```

First we run our code in the trace animator. It shows the possible orders of communications (here we evaluate four consecutive orders to show what will happen if the buffer is full or empty), and shows that if the system terminates then the buffers will terminate (here we replace the recursive call in processes with $SKIP$ to test termination). Figure 6.1 presents a screen-shot of the possible traces of our example produced by ProBE.

Figure 4 presents a screen-shot of our example executed in FDR with the following assertions tested:

```
assert sys [T= system
assert bufferedEvents:[deadlock free]
assert bufferedEvents:[livelock free]
assert system:[deadlock free]
```

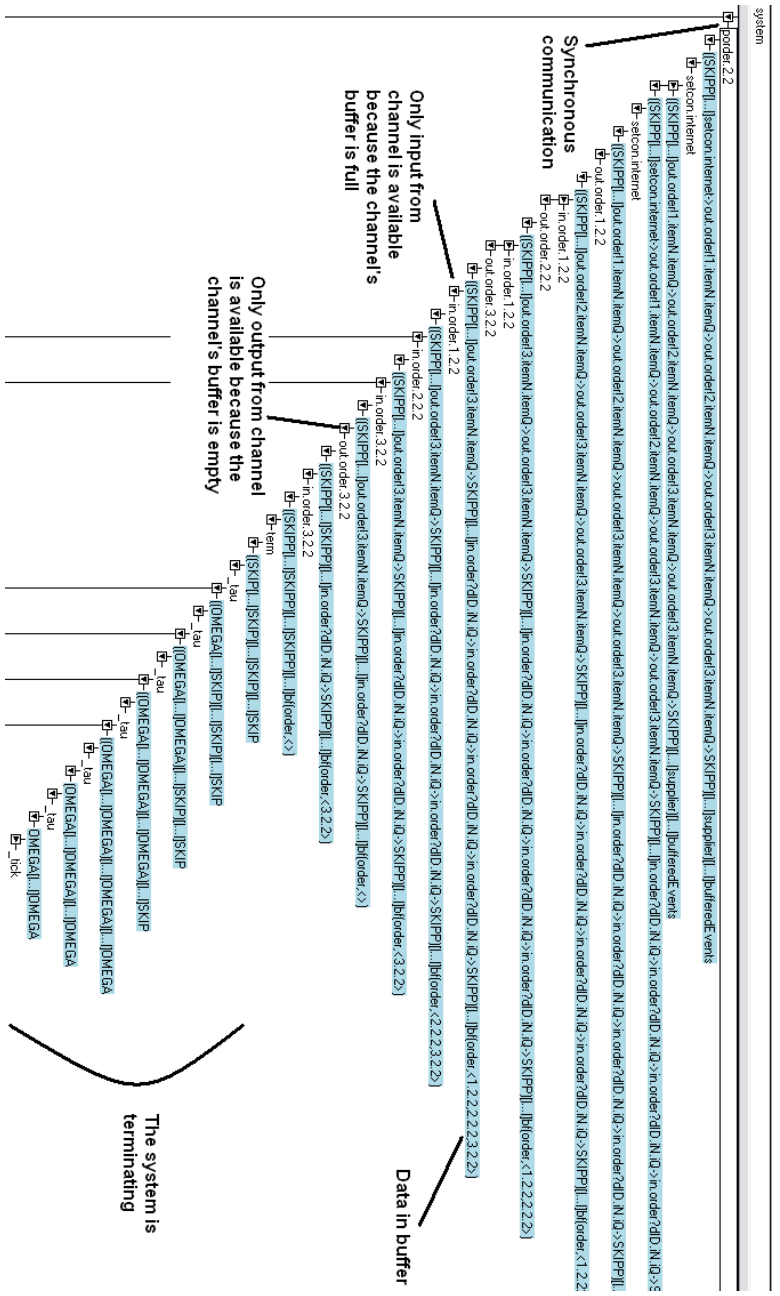

Figure 3. The possible execution of the example in ProBE

```
assert system:[livelock free]
assert sys:[deadlock free]
assert sys:[livelock free]
```

Most importantly these assertions (in particular, $assert \quad sys\,[T = system)$ show that the system (in the example) trace refines our buffered version of the system. This means that our model produces the same set of traces as the example

system without buffers. Additionally, we checked that the different processes in our system are deadlock free and livelock free as shown in the assertion statements listed in the figure, which shows that our model does not introduce deadlocks or livelocks if the original system does not have them.

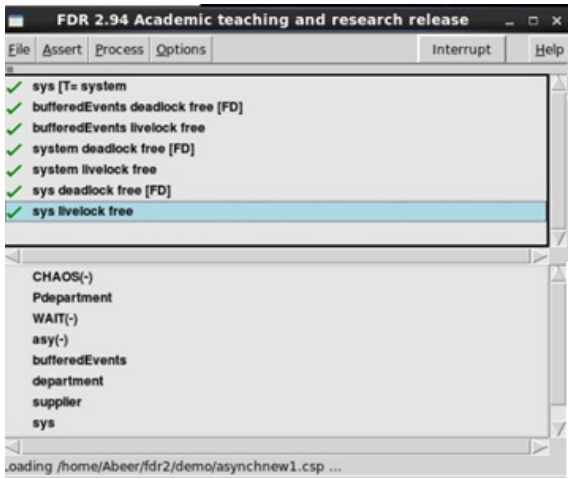

Figure 4. Assertion statements to check system properties

7 Related Work

Buffers are a well known mechanism to implement asynchronous communications, and they have been intensively used in process calculi literature.

In CSP, the use of buffered channels to facilitate asynchronous communications has been previously discussed by Hoare [13]. This model is not formally implemented, and assumes asynchronous communications only.

Buffered channels within CSP have been discussed also in [20], where all or a set of channels can be selected to be buffered between two processes. Installing buffers in two-way communications was enough for the purpose of this model which is proving that the correctness of a network of processes is independent of the amount of buffering added. However, in our model, we aim to provide a practical model where asynchronous communications are available as primitive communications in addition to the standard synchronous and interleaving communications. Therefore, we extend CSP with built-in buffers which can be used anywhere in the system. Additionally, our model retains the multi-way communication mode of CSP.

CSP# (the input language of PAT [23]) also extends CSP's syntax with buffered channels to support mixed synchronous/asynchronous communications. However, CSP# does not support the generalised parallel composition operator, therefore all shared channels are communicating in synchronous mode with no option to change this. In our model, designers have the option to choose which channel to synchronise on.

An asynchronous version of an early CSP-based language was mentioned in [7].

However, our approach is different. Our calculus preserves the notion of an output guard, allowing the same process to use synchronous and asynchronous communications (see the example in Section 6), whereas the language in [7] does not include output guard, like in the asynchronous π-calculus [15], thus synchronous communications are not allowed. Moreover, the language in [7] has no generalised parallel composition and the parallel composition is only allowed at the top-level (see [19] for more details).

Early proposals which support asynchronous communications by forcing interactions between two processes to be always mediated by buffers are described in [8,4], and in [3] buffers have been introduced to the π-calculus [21] to facilitate asynchronous communications as an alternative to the no-output-guards approach implemented in [15]. In [3], the encodability between the two calculi has been studied. While we use the same concept by introducing buffers in the middle of communications, in our model, we buffered the communications between multiple processes and our calculus supports mixed synchronous/asynchronous communications.

In the context of SOC, to the best of our knowledge our model is the first to mix synchronous/asynchronous communications; other calculi support only synchronous communications [5,6] or only asynchronous communications [16,24].

8 Conclusion and Future Work

Asynchronous communications are crucial in environments with unreliable media (like the internet). Additionally, synchronous communications are important to transfer critical data. For this reason, in this paper we define a calculus which supports mixed asynchronous/synchronous communication, by introducing an implicit buffer with each channel. We formally extended CSP's syntax and operational semantics with buffer loading and consuming primitives. In future work we will investigate how these buffered channels may work in mobile environments where channels can be sent as data. Additionally, we plan to introduce a model for creating and maintaining sessions within the framework of CSP.

References

[1] Al-Humaimeedy, A. S. and M. Fernández, *General dynamic recovery for compensating csp*, in: B. Löwe and G. Winskel, editors, *DCM*, EPTCS **143**, 2014, pp. 3–16.

[2] Bae, K. and J. Meseguer, *A rewriting-based model checker for the linear temporal logic of rewriting*, Electr. Notes Theor. Comput. Sci. **290** (2012), pp. 19–36.

[3] Beauxis, R., C. Palamidessi and F. D. Valencia, *On the asynchronous nature of the asynchronous pi-calculus*, in: P. Degano, R. De Nicola and J. Meseguer, editors, *Concurrency, Graphs and Models*, Lecture Notes in Computer Science **5065** (2008), pp. 473–492.

[4] Bergstra, J. A., J. W. Klop and J. V. Tucker, *Process algebra with asynchronous communication mechanisms*, in: S. D. Brookes, A. W. Roscoe and G. Winskel, editors, *Seminar on Concurrency*, Lecture Notes in Computer Science **197** (1984), pp. 76–95.

[5] Boreale, M., R. Bruni, L. Caires, R. De Nicola, I. Lanese, M. Loreti, F. Martins, U. Montanari, A. Ravara, D. Sangiorgi, V. T. Vasconcelos and G. Zavattaro, *Scc: A service centered calculus*, in: M. Bravetti, M. Núñez and G. Zavattaro, editors, *WS-FM*, Lecture Notes in Computer Science **4184** (2006), pp. 38–57.

[6] Boreale, M., R. Bruni, R. De Nicola and M. Loreti, *Sessions and pipelines for structured service programming*, in: G. Barthe and F. S. de Boer, editors, *FMOODS*, Lecture Notes in Computer Science **5051** (2008), pp. 19–38.

[7] Bougé, L., *On the existence of symmetric algorithms to find leaders in networks of communicating sequential processes*, Acta Inf. **25** (1988), pp. 179–201.
URL http://dx.doi.org/10.1007/BF00263584

[8] de Boer, F. S., J. W. Klop and C. Palamidessi, *Asynchronous communication in process algebra*, in: *LICS* (1992), pp. 137–147.

[9] Erl, T., "Service-Oriented Architecture: concepts, Technology and Design," Printice-Hall, 2005.

[10] Formal Systems, *ProBE Manual*, Formal Systems (Europe) Ltd. , pp. http://www.fsel.com/documentation/probe/probe–doc–html/html/index.html.

[11] Formal Systems, *FDR2 Manual*, Formal Systems (Europe) Ltd., 2009-2010 Oxford University (1992-2009), p. http://www.fsel.com/software.html.

[12] Forouzan, B. A., "TCP/IP Protocol Suite," McGraw-Hill, Inc., New York, NY, USA, 2002, 2 edition.

[13] Hoare, C., "Communicating Sequential Processes," Prentice Hall, 1985.

[14] Holzmann, G., "The SPIN Model Checker," Addison Wesley, 2003.

[15] Honda, K. and M. Tokoro, *An object calculus for asynchronous communication*, in: P. America, editor, *ECOOP*, Lecture Notes in Computer Science **512** (1991), pp. 133–147.

[16] Lapadula, A., R. Pugliese and F. Tiezzi, *A calculus for orchestration of web services*, in: R. De Nicola, editor, *ESOP*, Lecture Notes in Computer Science **4421** (2007), pp. 33–47.

[17] Milner, R., "Communication and concurrency," PHI Series in computer science, Prentice Hall, 1989, I-XI, 1-260 pp.

[18] OASIS, *Web services business process execution language version 2.0 oasis standard 11 april 2007*, OASIS WSBPEL TC (2007), pp. http://docs.oasis–open.org/wsbpel/2.0/OS/wsbpel–v2.0–OS.html.

[19] Palamidessi, C., *Comparing the expressive power of the synchronous and asynchronous pi-calculi*, Mathematical Structures in Computer Science **13** (2003), pp. 685–719.
URL http://dx.doi.org/10.1017/S0960129503004043

[20] Roscoe, A., "Understanding Concurrent Systems," Springer, 2010.

[21] Sangiorgi, D. and D. Walker, "The Pi-Calculus - a theory of mobile processes," Cambridge University Press, 2001.

[22] Scattergood, J., "Tools for CSP and Timed CSP," Ph.D. thesis, Oxford University (1998).

[23] Sun, J., Y. Liu, J. S. Dong and J. Pang, *Pat: Towards flexible verification under fairness*, in: A. Bouajjani and O. Maler, editors, *CAV*, Lecture Notes in Computer Science **5643** (2009), pp. 709–714.

[24] Vieira, H. T., L. Caires and J. C. Seco, *The conversation calculus: A model of service-oriented computation*, in: S. Drossopoulou, editor, *ESOP*, Lecture Notes in Computer Science **4960** (2008), pp. 269–283.

[25] W3C, *Web services choreography description language version 1.0 W3C candidate recommendation 9 november 2005* (2005), pp. http://www.w3.org/TR/2005/CR–ws–cdl–10–20051109/.