Mobile CSP

Jim Woodcock*, Andy Wellings, and Ana Cavalcanti

Department of Computer Science

University of York

Abstract. We describe an extension of imperative CSP with primitives to declare new event names and to exchange them by message passing between processes. We give examples in Mobile CSP to motivate the language design, and describe its semantic domain, based on the standard failures-divergences model for CSP, but also recording a dynamic event alphabet. The traces component is identical to the separation logic semantics of Hoare & O'Hearn. Our novel contribution is a semantics for mobile channels in CSP, described in Unifying Theories of Programming, that supports: compositionality with other language paradigms; channel faults, nondeterminism, deadlock, and livelock; multi-way synchronisation; and many-to-many channels. We compare and contrast our semantics with other approaches, including the π-calculus, and consider implementation issues. As well as modelling reconfigurable systems, our extension to CSP provides semantics for techniques such as dynamic class-loading and the full use of dynamic dispatching and delegation.

1 Introduction and Overview

Model-driven systems engineering is gaining popularity in large-scale industrial applications; it relies for its success on modelling languages that provide efficient domain-specific abstractions for design, analysis, and implementation. There is no single modelling language that can cover every aspect of a significant system, let alone the complexities of systems of systems or cyber-physical systems. Adequate modelling techniques must inevitably involve heterogeneous semantics, and this raises the scientific question of how to fit these different semantics together: *the model integration problem* [15]. Our approach to understanding the integration of models with diverse semantics is to study the different paradigms in isolation and then find ways of composing them. We are not looking for a unified language to encompass all language paradigms, but rather we seek to unify theories to explain how they fit together and complement each other. This is the research agenda of Unifying Theories of Programming (UTP) [12].

In this paper, we explore the paradigm of reconfigurable systems and programs, where we model interaction between system components (and even between systems themselves) by message passing along channels that form a flexible topology that changes over time. We describe an extension of CSP [11, 22] with

* Corresponding author: Jim.Woodcock@york.ac.uk.

primitives to declare new event names and to exchange them in messages over channels. Our semantics is an extended predicative form of the standard failures-divergences semantic model for CSP, enhanced with imperative programming features [12]. The traces component of this model is identical to the concurrent separation logic semantics proposed earlier by Hoare & O'Hearn [13]. The novel contribution of our work is a semantics in UTP that supports the following:

1. Compositionality with other language paradigms. A key feature of UTP is the ability to combine different language features using Galois connections.
2. Formalisation of channel faults, nondeterminism, deadlock, livelock, multi-way synchronisation. and many-to-many channels.

We start the paper in Sect. 2 by recalling the principal features of CSP and of the π-calculus, its process-algebraic cousin with mobile channels. In Sect. 3, we list a series of examples of increasing complexity that display the use of mobile channels in modelling. Having motivated the use of mobility, we define our semantic domain in Unifying Theories of Programming in Sect. 4, and give the semantics of four key operators in Sect. 5: value and channel communications, the creation of new channels, and parallel composition. In Sect. 6, we describe the implementation in Java of an architectural pattern that uses mobile channels. We conclude the paper by describing related and future work in Sects 7 and 8.

2 CSP

CSP is a formal language for describing patterns of interaction in concurrent systems [11, 22]; it is a process algebra based on message passing via channels. It has formed the basis of the following languages: occam, the native programming language for the inmos transputer microprocessors [32], and occam-π, its extension with mobile processes and data [36]; Ada's rendezvous mechanism [14]; JCSP, the CSP library for Java [35]; PyCSP, the CSP library for Python [1]; Scala, the strongly typed functional programming language [20], with its message-passing semantics and Communicating Scala Objects; the *Circus* family of specification and refinement languages [38, 39], including *OhCircus* [5], *SlottedCircus* [3], *CircusTime* [27], and *TravellingCircus* [30, 31]; *CML*, the COMPASS Modelling Language [41, 37]; CSP‖B and Mobile CSP‖B [33, 26, 34]; CSP-OZ and CSP$^{\#}$, stateful, object-oriented versions of the language [10, 29]; rCOS, the component modelling language [16]; and Ptolemy, the embedded systems modelling language [28]. The main elements of CSP are described in Tab. 1.

Different aspects of semantics, such as determinism, nondeterminism, livelock, timing, and fairness, are dealt with in a hierarchy of semantic models, all based on refinement as inverse behavioural inclusion: every implementation behaviour must be specified. A powerful refinement model-checker for CSP, FDR3 [19], supports the language and a basic extension to timed systems.

The π-calculus [18] differs significantly from CSP in permitting channel names to be communicated along the channels themselves, and in this way it is able to describe concurrent computations whose network configurations may change

prefix	$a \rightarrow P$	input	$c?x \rightarrow P(x)$
output	$c!e \rightarrow P$	internal choice	$P \sqcap Q$
external choice	$P \Box Q$	sequence	$P \ ; \ Q$
parallel	$P \parallel Q$	abstraction	$P \setminus S$
recursion	$\mu\, X \bullet F(X)$	deadlock	$STOP$
termination	$SKIP$	divergence	$CHAOS$

Table 1. The main elements of CSP.

during the computation. As well as treating channel names as first-class citizens, the π-calculus has a further primitive, $(\nu x)\, P$, that allows for the creation of a new name allocated as a constant within P. An axiom, known as scope extrusion,

$$(\nu\ x)P \mid Q \ = \ (\nu\ x)(P \mid Q) \qquad \text{if } x \text{ is not a free name of } Q$$

describes how the scope of a bound name x may be extruded, as would be necessary before an action outputting the name x from P to Q.

In CSP, channel communications are events, and input and output commands are merely abbreviations for choices over event synchronisations:

$$c?x : T \rightarrow P(x) \ \widehat{=} \ \Box\, x : T \bullet c.x \rightarrow P(x) \qquad c!e \rightarrow Q \ \widehat{=} \ c.e \rightarrow Q$$

So a theory of mobile events underpins a theory of mobile channels. In this paper, we propose an extension of imperative CSP with mobile events; this language supports *MobileCircus*, an extension of the *Circus* modelling language. Both language extensions are based on a natural notion of refinement of failures-divergences, which distinguishes them from the π-calculus.

3 Motivation and Examples

One of the main areas underpinned by research in formal methods is software for high-integrity and safety-critical systems. For example, recent work on a subset of Java for safety-critical systems (SCJ) is based on the programming model being defined in *SCJ-Circus* [6, 42, 7, 8], which is an extension to *Circus* whose semantics is defined using UTP. Together with formal models of the SCJ virtual machine, this allows the full semantics of an SCJ application to be defined [9].

SCJ is conservative in order to comply with guidelines for certification, such as DO-178C [25]; however, within the SCJ development team, there is the recognition that high-integrity software is generally becoming progressively more complex. To this end, they define different compliance levels. The most expressive programming model is supported at Level 2 compliance, and the SCJ team accept that certification of Level 2 applications requires significantly more effort and evidence than at Level 0 or Level 1 compliance. Even the Level 2 programming model is unable to exploit fully the power of the Java programming language due to the concerns over the ability to produce convincing certification evidence for

programs that support dynamic class-loading, potentially across a network. It is also anticipated that some certification authorities may limit the use of other Java features to constrain the amount of dynamic dispatching and delegation that can occur in object-oriented programming languages (although the recent work in DO-178C shows that such techniques are becoming more accepted [25]).

The examples in this section illustrate some of the more dynamic behaviour that programs can exhibit, for which concise and intuitive formal models are required. The extension to CSP proposed in this paper provides semantics for techniques such as dynamic class-loading and the full use of dynamic dispatching and delegation. This can then be used in supporting evidence to allow certification of more complex systems to be considered in the future.

Example 1 (Frequent Flyer). Meyer gives an example of dynamic binding in Eiffel [17]: a person who is in a frequent flyer programme connects to a server with their membership number; they receive in reply a connection to another server according to their membership level: *Blue*, *Silver*, or *Gold*, and connections are made over the corresponding mobile channels *blue*, *silver*, and *gold*.

$$
\begin{aligned}
FFConnect = connect?p : MemberNo \rightarrow \\
\quad \textbf{if } p \in Blue \textbf{ then } service!blue \rightarrow SKIP \\
\quad \textbf{else if } p \in Silver \textbf{ then } service!silver \rightarrow SKIP \\
\quad \textbf{else if } p \in Gold \textbf{ then } service!gold \rightarrow SKIP \\
\quad \textbf{else } STOP
\end{aligned}
$$

The process *FFConnect* serves a one-shot transaction. It waits for a membership number p input on the *connect* channel; it then analyses the value of p, and returns an appropriate channel name on the *service* channel. The code is a specification of the implementation in Eiffel that uses dynamic binding. □

Example 2 (Airline Check-in). An airline check-in system behaves as follows. The system consists of a collection of passengers, a clerk who assigns passengers to check-in desks, and the employees at the desks themselves. The behaviour of a passenger who wants to travel to a particular destination is as follows:

$$Passenger(dest) = \textbf{new } p \bullet checkin!(p, dest) \rightarrow p?bc \rightarrow P(bc)$$

The passenger generates a fresh channel p for the visit to a desk, and then communicates that channel and the destination over the *checkin* channel to the clerk. The passenger then waits to receive a boarding card over channel p. The clerk receives the channel name and destination from a passenger and then waits for a desk to become free, which is signalled on the *next* channel with a channel name cd. The passenger then goes and does something else ($P(bc)$).

$$Clerk = checkin?(p, d) \rightarrow next?cd \rightarrow cd!(p, d) \rightarrow Clerk$$

The clerk then uses cd to inform the desk about the next passenger and their destination. $Desk(i)$ describes the behaviour of an airline representative. The representative generates the fresh channel name cd and sends it over the *next*

channel for use by the clerk. The representative then waits to receive a communication on cd that tells them of the next passenger and their destination. The transaction is finalised by a reply on the passenger's channel p giving details of the boarding card $bcard(d)$.

$$Desk(i) = \mathbf{new}\, cd \bullet next!cd \rightarrow cd?(p, d) \rightarrow p!bcard(d) \rightarrow Desk(i)$$

The system is then given by

$$CheckIn = \left(\|\!|\!|\, i : Desks \bullet Desk(i)\right) \parallel Clerk \parallel \left(\|\!|\!|\, d : Today \bullet Passenger(d)\right)$$

Today's destinations is a bag, with one destination for each passenger. □

In contrast to the previous example, this system does not involve dynamic binding as in OO languages, but instead a kind of dynamic binding of resources.

In occam-π, processes exchange the ends of channels [36]; as we see below, our theory is more powerful than this and involves mobile events. In spite of this, it is useful to describe a process's use of a channel in terms of the read-end or the write-end, and this usage can be checked syntactically. The following example uses Mobile CSP to model a simple two-place buffer using mobile channel ends. Of course, the obvious implementation would involve a single process using a linked-list data structure programmed using pointers. This may not be appropriate in a system with distributed memory, where a pointer in one memory space would have to address memory in another space.

Example 3 (Two-place Buffer). A user wants to read and write to a two-place buffer, and to do this, the user holds the input end of the *write* channel and the output end of the *read* channel. The buffer is made of two parallel processes connected by two channels, *chw* and *chr*. Between them, the two buffer processes hold the output end of the *write* channel and the input end of the *read* channel, and they swap ownership between themselves on the *chw* and *chr* channels, respectively. The state-transition for the buffer is pictured in Fig. 1, where the starting state has the left-hand process holding the buffer's ends of the *write* and *read* channels. The behaviour is:

$$\begin{aligned}
D_0(w, r) &= w?x \rightarrow chw!w \rightarrow D_1(x, r) \\
D_1(x, r) &= r!x \rightarrow chr!r \rightarrow D_2 \,\Box\, chw? \rightarrow D_2(x, w, r) \\
D_2 &= chw?w \rightarrow D_4(w) \\
D_3(x, w, r) &= r!x \rightarrow chr!r \rightarrow D_4(w) \\
D_4(w) &= chr?r \rightarrow D_0(w, r) \,\Box\, w?x \rightarrow D_5(x, w, r) \\
D_5(x) &= chr?r \rightarrow D_1(x, r)
\end{aligned}$$

The buffer has some invariant properties: where the buffer contains two elements or none (even parity), both ends reside in the same half of the buffer; where there is just a single element, the two channel ends reside in different halves, with the *read* end in the element's half. □

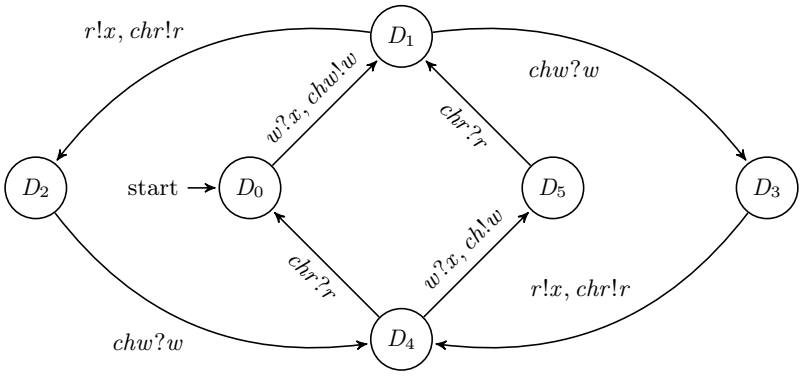

Fig. 1. STD for one-place buffer with mobile channel ends.

Example 4 (Ring Buffer). We can generalise Ex. 3 to an n-place ring buffer. Each cell in the buffer behaves as follows:

$$Cell(i) = ring.i?(c?) \rightarrow c?x \rightarrow ring.(i+1 \bmod n)!(c?) \rightarrow \\ ring.i?(d!) \rightarrow d!x \rightarrow ring.(i+1 \bmod n)!(d!) \rightarrow Cell(i)$$

The cell starts by receiving the input end of a channel $c?$ over the channel $ring.i$; it then uses channel c to input a value x, which it buffers. The cell passes the channel end $c?$ on to the next cell in the ring; it does this by using the channel $ring.(i+1 \bmod n)$. The cell waits for the output end of another channel $d!$, which it receives again on channel $ring.i$. It then outputs the value x, which it has been buffering, on channel d, before passing the channel end $d!$ to the next cell in the ring using $ring.(i+1 \bmod n)$. Every cell is identical, starting with receiving the input end of a channel from its neighbour; so how does the buffer start being useful? The answer is to use one of the cells (it might as well be $Cell(0)$) in process $S(in?, out!)$. This process waits for the first input on the in channel, then passes the channel on to $Cell(1)$; it then waits to output its buffered value, passing the output channel on to $Cell(1)$; it then behaves like $Cell(0)$. In this way, the channel ends get into the ring.

$$S(in?, out!) = in?x \rightarrow ring.1!(in?) \rightarrow out!x \rightarrow ring.1!(out!) \rightarrow Cell(0)$$

The ring is then constructed from the cells, treating $Cell(0)$ to its initialisation:

$$CellBuffer(n) = S(in?, out?) \parallel \Big\| i : 1 \mathinner{..} n-1 \bullet Cell(i)$$

Process D in the previous example is simply a special case of this definition. □

Example 5 (Sieve of Eratosthenes). Primes models the Sieve of Eratosthenes and generates prime numbers on the channel c; it is composed initially of just two processes, one that generates natural numbers, and one that sifts them to

$\mathcal{A}$	event alphabet	physical and logical capabilities
$ok, ok' : \mathbb{B}$	stability	freedom from divergence
$wait, wait' : \mathbb{B}$	quiescence	waiting for interaction
$tr, tr' : \mathcal{A}^*$	trace	history of interaction
$ref, ref' : \mathbb{P}\,\mathcal{A}$	refusals set	events refused during wait
v, v'	program variables	imperative state

Table 2. Alphabet for CSP processes.

remove composite numbers:

$$Primes(c) = \mathbf{new}\ d \bullet Nats(2, d) \parallel Sift(d, c)$$
$$Nats(n, d) = d!n \rightarrow Nats(n + 1, d)$$

The process *Sift* spawns a series of filters, each removing composites:

$$Sift(in, out) = \mathbf{new}\ d \bullet in?p \rightarrow out!p \rightarrow Filter(p, in, d) \parallel Sift(d, out)$$
$$Filter(p, in, out) = \mu\, X \bullet in?x \rightarrow (out!x \rightarrow X \lhd x \bmod p \neq 0 \rhd X)$$

The mobile channels are used to build an unbounded process structure: we can start *Primes* as a prime number server in some larger system, knowing that it will run indefinitely (well, until the underlying resources required for channels are exhausted). The obvious alternative implementation is to declare a sufficiently large number of channels in advance, and then to use these one by one. The difference between these approaches is similar to lazy versus eager evaluation in functional programming, and the advantages are the same. □

4 Semantic Domain

Hoare & He give the semantic domain for CSP in UTP [12, Chap. 8] (see also [40, 4] for tutorial introductions). In their semantics, each process is represented by an alphabetised predicate arranged in a lattice ordered by refinement, which is defined as universally closed inverse implication. The alphabet describes the observations that can be made of processes, and these are summarised in Tab. 2. Each predicate in the lattice is actually a relation between a before-state (ok, $wait$, tr, ref, and v) and an after-state (ok', $wait'$, tr', ref', and v'); the alphabet $\mathcal{A}$ is a constant. Membership of the lattice is defined by the fixed-points of five functions representing healthiness conditions, and these are summarised in Tab. 3. The predicates in this lattice can also be expressed as $\boldsymbol{R}$-healthy precondition-postcondition pairs: "reactive designs" [21] (where $\boldsymbol{R}$ is the composition of ***R1*** to ***R3***). The precondition describes the conditions under which the process does not diverge, while the postcondition describes its failures.

Now we can define the semantic domain for mobile CSP. The obvious idea is to make the alphabet a dynamic variable; however, a moment's thought shows this is inadequate because of compositionality: we need the dynamic alphabet's

never undo	***R1***	$P \;=\; P \wedge (tr \leq tr')$
ignore history	***R2***	$P(tr, tr') \;=\; \prod s \bullet P(s, s \frown (tr' - tr))$
wait!	***R3***	$P \;=\; (\mathit{II}_{\mathbf{R}} \lhd wait \rhd P)$
diverge	***CSP1***	$(\neg\, ok \wedge tr \leq tr') \vee P$
ok'-monotonicity	***CSP2***	$P \;;\; J$

$\mathit{II}_{\mathbf{R}} \;\widehat{=}\; (\neg\, ok \wedge tr \leq tr') \vee (ok' \wedge (tr' = tr) \wedge \cdots \wedge (v' = v))$
$J \;\widehat{=}\; (ok \Rightarrow ok') \wedge (wait' = wait) \wedge (tr' = tr) \wedge (refr' = ref) \wedge (v' = v)$

Table 3. Healthiness conditions for CSP processes.

history when we come to compose the traces of two parallel processes. For example, consider the process that executes an event a and then enters a state with alphabet $\{a, b\}$. What can we say about the alphabet *before* the a event? It certainly must include a itself, but what about b? We need to know the answer in order to know whether the process has right of veto over b in any composition. A better strategy is to record a process's alphabet before and after each event.

Definition 1 (Dynamic Alphabetised Traces (DATs)). *A DAT is non-empty and alternates alphabets and events, starting with an alphabet:*

$$DAT^{\epsilon} \;\widehat{=}\; \{\, s \mid \#s \in Odd \wedge odds(s) \in (\mathbb{P}\, \Sigma)^* \wedge evens(s) \in (\Sigma \cup \{\epsilon\})^* \,\}$$

The silent event ϵ is used below to define the **new** *and* **dispose** *commands that manipulate a process's alphabet.* □

Example 6 (Dynamic Alphabetised Traces). The following are all valid DATs

$$\langle \{b\} \rangle \qquad \langle \{b\}, b, \{a, b\} \rangle \qquad \langle \{b\}, b, \{a, b\}, a, \{a, b\} \rangle$$

(Here, *Odd* is the set of odd integers; $odd(s)$ is the sequence of s's odd-indexed elements; and $even(s)$ is the sequence of s's even-indexed elements. □

We can now express some simple properties over DAT traces:

- start, owning c: $\langle \{c\} \rangle$
- acquire event c: $\langle \cdots, \{a, b\}, a, \{a, b, c\}, \cdots \rangle$
- release event c: $\langle \cdots, \{a, b, c\}, a, \{a, b\}, \cdots \rangle$

Mobile processes satisfy two healthiness conditions on their DAT, tr.

Definition 2 (Ownership). *A mobile process can engage in an event only if it has already acquired it, but not released it.*

$$\textbf{\textit{M1}} \quad P \;=\; P \wedge (\forall\, s : \mathbb{P}\, \Sigma;\; e : \Sigma \bullet \langle s, e \rangle \in \operatorname{ran} tr' \Rightarrow e \in s\,)$$

This is defined using a monotonic idempotent healthiness condition. □

Definition 3 (Refusalship). *A mobile process can refuse only those events it has acquired.*

$$\textbf{\textit{M2}} \quad P \;=\; P \wedge ref' \subseteq last\ tr'$$

Again, this is enforced by a monotonic idempotent healthiness condition. □

The two healthiness conditions commute. The reactive healthiness conditions must hold, but with ***R2*** for $(Even \lhd tr)$ and $(Even \lhd tr')$, (the operator $\lhd$ is domain restriction of a function). ***R2***(P) ensures compositionality by insisting that P does not depend on particular values for tr. In the sequence $P \;;\; Q$, the final alphabet in the trace of P must match the initial alphabet of Q. Finally, concatenation between dynamic alphabet traces is a partial function:

$$last\ t = head\ u \Rightarrow t \overset{\frown}{\rightarrow} u = t \frown (tail\ u)$$

The ***M*** healthiness conditions commute with the ***R*** healthiness conditions.

5 Semantics of Operators

In this section, space allows us to give the semantics for a few key constructs. We do this in the style of reactive designs [21], as described in Sect. 4. The definition of a UTP design with precondition P and postcondition Q is [12]:

$$P \vdash Q \;\widehat{=}\; (ok \wedge P \Rightarrow ok' \wedge Q)$$

That is, if the design is started in a stable state (ok) and the precondition is true (P), then it must reach a stable state (ok') and when it does so, the postcondition will be true (Q). This is a statement of total correctness.

5.1 Event Prefixes (Value + Channel)

The semantics of an event-prefixed process, $a \rightarrow SKIP$, depends on whether the event a is the communication of a channel name. If it is not, then the UTP semantics is similar to that in standard CSP [12]:

Definition 4 (Event Prefix (Value)).

$$\textbf{\textit{M}} \circ \textbf{\textit{R}}(a \in last\ tr \vdash tr' = tr \wedge a \notin ref' \lhd wait' \rhd tr' = tr \frown \langle a, last\ tr \rangle)$$

The precondition requires that a is in the current alphabet, the last entry of the trace preceding execution of the process $(last\ tr)$; the precondition in standard CSP is simply **true**. Since tr is a DAT, it ends with an alphabet, so $last\ tr$ is well defined. The postcondition has a small difference too: if the process terminates, then $\langle a, last\ tr \rangle$ is appended to the trace ($\overset{\frown}{\rightarrow}$ is not needed here); the current alphabet is unchanged by this kind of event. If the event a is not in the current alphabet, then the design aborts and the process diverges: this is a channel fault.

Now consider $c?n \rightarrow SKIP$, which inputs channel name n over c.

Definition 5 (Event Prefix (Channel Name)).

$$\boldsymbol{M} \circ \boldsymbol{R}\begin{pmatrix} c.n \in last\ tr \\ \vdash tr' = tr \wedge c.n \notin ref' \lhd wait' \rhd tr' = tr \frown \langle c.n, (last\ tr) \cup \{\!|n|\!\}\rangle \end{pmatrix}$$

The difference is the expansion of the alphabet with $\{\!|n|\!\}$, which is the set of all events communicable over n. Outputting a name is complementary.

5.2 New and Dispose

In UTP, a block-structured declaration **var** $x \bullet P$ is semantically equivalent to the predicate **var** x ; P ; **end** x, where the beginning and end of the scope of x are treated separately [12, Chap. 2]. We adopt a similar approach to the block-structured allocation of fresh channels **new** $c \bullet P$, and deal separately with the allocation and disposal of a channel: **new** c ; P ; **dispose** c.

Definition 6 (New Channel). *For fresh* c,

$$\textbf{new}\ c\ ;\ P\ \mathrel{\widehat{=}}\ \boldsymbol{M} \circ \boldsymbol{R}(\,\textbf{true} \vdash \neg\, wait \wedge tr' = tr \frown \langle \epsilon, (last\ tr) \cup \{\!|c|\!\}\rangle\,)$$

Definition 7 (Dispose Channel).

$$\textbf{dispose}\ c\ ;\ P\ \mathrel{\widehat{=}}\ \boldsymbol{M} \circ \boldsymbol{R}(\,\textbf{true} \vdash \neg\, wait \wedge tr' = tr \frown \langle \epsilon, (head\ tr') \setminus \{\!|c|\!\}\rangle\,)$$

Example 7 (Channel Allocation). Consider the $Desk(i)$ process in Ex. 2:

$$\textbf{new}\ cd \bullet next!cd \rightarrow cd?(p, d) \rightarrow p!bcard(d) \rightarrow Desk(i)$$

The process must initially own the *next* channel's events: $\{\!|next|\!\}$; these events are all channel name communications. Here is an example trace:

$$\begin{array}{l} \langle\, \{\!|next|\!\}, \\ \quad \epsilon, \{\!|next|\!\} \cup \{\!|cd|\!\}, \\ \quad\quad next.cd, \{\!|next|\!\} \cup \{\!|cd|\!\}, \\ \quad\quad\quad cd.(p, d), \{\!|next|\!\} \cup \{\!|cd|\!\} \cup \{\!|p|\!\}, \\ \quad\quad\quad\quad p.bcard(d), \{\!|next|\!\} \cup \{\!|cd|\!\} \cup \{\!|p|\!\}, \\ \quad\quad\quad\quad\quad \epsilon, \{\!|next|\!\} \cup \{\!|p|\!\}\,\rangle \end{array}$$

In this trace, cd is a fresh channel name; p is bound to a channel name input as part of the pair on the cd channel. Notice how we automatically dispose of cd at the end of the process; however, there is no automatic disposal of the channel denoted by p. A better definition for the process would tidy this up:

$$\textbf{new}\ cd \bullet next!cd \rightarrow cd?(p, d) \rightarrow p!bcard(d) \rightarrow \textbf{dispose}\ p;\ Desk(i)$$

Here, the events of whichever channel is denoted by p are removed from the alphabet when the scope of the channel variable p ends. □

5.3 Parallel Composition

In UTP, parallel composition uses the parallel-by-merge semantic pattern taken from Hoare & He's UTP semantics for ACP, CCS, and CSP [12]: two processes have their overlapping alphabets separated by renaming; they are then run in parallel producing two states, which are then merged to give the meaning of the composition. We need to specify only the merge for DAT traces.

Definition 8 (Parallel Merge). *Define a "catset" operator that concatenates its left-hand sequence operand with every sequence in its right-hand set operand:*

$$s \mathbin{⍟} T \mathrel{\widehat{=}} \{ u : T \bullet s \frown u \}$$

We use this operator to define the parallel composition of two DAT traces:

$$\langle s \rangle \frown xs \parallel \langle t \rangle \frown ys \mathrel{\widehat{=}} \langle s \cup t \rangle \mathbin{⍟} N(s, xs, t, ys)$$

$$N(s, \langle\rangle, t, ys) = \{ys\}$$

$$N(s, xs, t, \langle\rangle) = \{xs\}$$

$$N(s, \langle x \rangle \frown xs, t, \langle y \rangle \frown ys) =$$

$$\begin{aligned} &\textbf{if } x = y \neq \epsilon \textbf{ then } \langle x \rangle \mathbin{⍟} (xs \parallel ys) \\ &\textbf{else } (\textbf{ if } x \notin t \textbf{ then } \langle x \rangle \mathbin{⍟} (xs \parallel \langle t, y \rangle \frown ys)\,) \\ &\qquad \cup (\textbf{ if } y \notin s \textbf{ then } \langle y \rangle \mathbin{⍟} (\langle s, x \rangle \frown xs \parallel ys)\,) \end{aligned}$$

Silent events occur independently: if two parallel processes each allocate a new channel, then there is no synchronisation of the two **new** *commands. It is easily shown by induction that the merge operator is closed on DAT traces.* □

Example 8 (Parallel Merge). Consider the two Mobile CSP processes: $P = a \rightarrow SKIP$ and $Q = get.a \rightarrow a \rightarrow SKIP$.

P's behaviour includes the following trace: $\langle \{a\}, a, \{a\} \rangle$; Q's behaviour includes $\langle \{get.a\}, get.a, \{get.a, a\}, a, \{get.a, a\} \rangle$. The parallel composition of the two traces describes two behaviours: the first has P executing a *before* Q gets hold of a ($\langle \{get.a, a\}, a, \{get.a, a\}, get.a, \{get.a, a\} \rangle$); the second has P executing a *afterwards* Q ($\langle \{get.a, a\}, get.a, \{get.a, a\}, a, \{get.a, a\} \rangle$. In the first trace, P executes a independently; in the second trace P and Q synchronise on a. □

6 Implementation

The ring buffer described in Ex. 4 can be implemented in pure CSP:

$$\begin{aligned} Imp(i) = ring.i?c \rightarrow chan.c?x \rightarrow ring.((i+1) \bmod n)!c \rightarrow \\ ring.i?d \rightarrow chan.d!x \rightarrow ring.((i+1) \bmod n)!d \rightarrow Imp(i) \end{aligned}$$

Here, we simulate mobile channels by passing around tokens so that the end of a channel can be used only by the process that holds the token for that channel end, whilst those processes without a token do not block. We use a token for the

```
package mobileCode;
public interface ServerInterface {
   public void useService(String parameters);
}
```

Fig. 2. `ServerInterface.java`

```
public enum MembershipLevel {Blue, Silver, Gold}
public class Broker {
   // directory of servers implemented via a Java Map
   public ServerInterface lookUpService(MembershipLevel l) {
     ServerInterface server;
     // lookup server
     return server;
  }
   public synchronized void register(ServiceProvider serverThread,
                                     MembershipLevel level) {
     // save details in map
  }
}
```

Fig. 3. `MembershipLevel.java`

relevant channel name to index an array of channels *chan*; interleaving is needed to share the channel ends (see [11] on shared resources).

Examples 1 and 2 use the *broker* architectural pattern [2], suitable for distributed systems where clients invoke remote services, but are unconcerned with the details of remote communication. In systems engineering, there are many practical reasons to adopt a distributed architecture. The system may need to take advantage of multiple processors or a cluster of low-cost computers. Certain software may be available only on specific computers, or provided by third parties and available on the cloud. The broker pattern can hide many of these implementation issues by encapsulating required services into a separate layer.

Example 9 (Broker Pattern). In SCJ, the broker pattern is an application that requires dynamic dispatching through interfaces, as illustrated in Fig. 2. The broker itself simply maintains a directory of service providers. In Ex. 1, there are three service providers for *Blue*, *Silver* and *Gold* membership levels, as shown in Fig. 3. These register with the broker via a synchronised method. To simplify the example, assume that the system runs on a multiprocessor server, and the service providers have their own resources allocated. The application is encapsulated in an SCJ mission (subsystem) and the service providers are managed threads that implement the service interface, as shown in Fig. 4. The threads are Java daemon threads, terminating automatically. Finally, the clients are also managed threads

```
public class ServiceProvider extends ManagedThread implements ServerInterface {
   @Override
   public void useService(String s) {
     // add to queue of requests
     // wait until search has been performed
     return;
   }
   @Override
   public void run() {
     broker.register(this, level);
     while (true) {
       // perform services while needed
     }
   }
   public ServiceProvider(Broker b, MembershipLevel l) {
     super();
     this.broker = b;
     this.level = l;
     this.setDaemon(true);
   }
   private Broker broker;
   private MembershipLevel level;
}
```

Fig. 4. `ServiceProvider.java`

that are assigned a particular membership level when created (Fig. 5). To analyse an program with the broker pattern requires all possible classes implementing `ServiceInterface` to provide equivalent functionality. For a large system, where the same broker is used to provide the interface between many service providers and clients, this may be difficult to guarantee and to provide evidence that each service provider is being used in the correct context. □

The broker acts as a messenger: locating an appropriate server; forwarding requests to that server, possibly marshalling data; and transmitting results to the client, possibly demarshalling data. Clients are applications that access servers, and they call the remote service by forwarding requests to the broker and receiving responses or exceptions in reply. Widely used broker patterns include OMG's CORBA standard, Microsoft's Active X, and the World-Wide Web, where browsers act as brokers and servers act as service providers.

7 Related Work

Our traces component is essentially the same as that of Hoare & O'Hearn [13]; they explore the unification of CSL (concurrent separation logic) and CSP by

```
public class Client extends ManagedThread {
   private Broker broker;
   private MembershipLevel level;
   public Client(Broker b, MembershipLevel l) {
     broker = b; level = l;
   }
   @Override
   public void run() {
     // code for client
     ServerInterface myProvider = broker.lookUpService(level);
     myProvider.useService(params);
   }
   private String params;
}
```

Fig. 5. `Client.java`

adding temporal separation to CSL and mobile channels to CSP, restricting to the traces model, excluding nondeterminism, deadlock, and livelock. Their interest lies in point-to-point communication, using ideas from separation logic to reason about exclusive use of channel ends, used as in occam-π [36]. Processes that send a channel end automatically relinquish ownership. This differs from our work, where channels may have many ends and ownership must be handled explicitly. Actually, the structure of a trace is slightly different: instead of simple events, they allow sets of events, giving a semantics for true concurrency and having no need for ϵ. Their work, with its deliberate restrictions, leads to a very simple notion of event composition using point-wise disjoint union.

Roscoe discusses a version of CSP with mobility [24, Sect. 20.3], so that all processes that do not presently have the right to use a particular action always accept the event and never change their state when the event occurs. This is the same as our route to implementation in Sect. 6. He has shown how a full semantics of the π-calculus can be given in CSP [23]: for each π-calculus agent, there is a CSP process that models it accurately.

In Mobile CSP‖B [34], controllers can work with different machines during execution. Controllers can exchange machines between each other by exchanging machine references and manage concurrent state updates. This work goes further than the current paper and the references cited in this section by dealing with the fusion of concurrency, communication, state, and channel mobility.

8 Conclusions and Future Work

This paper has explored a semantics for mobile CSP for specifying and verifying safety-critical code. It permits the use of dynamic programming features, such as the use of the broker architectural and programming pattern, in a controlled

fashion so that evidence for assurance can be collected and relied upon. The work started as a contribution to occam-π; more recently, it has contributed to Safety-Critical Java and on reasoning about systems of systems and cyber-physical systems. The work is still at an early stage, and there are plenty of directions for future work. We will give a complete account of the operators of mobile CSP and extend this to *MobileCircus*. We need to prove closure of all these operators with respect to $\boldsymbol{R}$ and $\boldsymbol{M}$, and perhaps devise additional healthiness conditions. We have hinted at the possibility of translating mobile CSP into plain CSP; we need to record a Galois connection between the two and use it as the basis of a translator. Finally, since our intention is to verify system architectures and programs, we will devise a Hoare logic for Mobile CSP and prove it sound.

9 Acknowledgements

This work has been supported by the EPSRC hiJaC and the EU H2020 INTO-CPS projects. Thanks to Philippa Gardiner for an application of the π-calculus that led to Ex. 2; to Peter Welch for discussions more than a decade ago on mobile channels in occam-π that led to the semantic model in this paper; and to three anonymous referees for prompting clarifications in the paper.

References

1. John Bjørndalen et al. PyCSP — CSP for Python. In Alistair McEwan et al., editors, *CPA*, pages 229–248, 2007.
2. Frank Buschmann. *Pattern-Oriented Software Architecture*. Wiley, 1996.
3. Andrew Butterfield et al. *Slotted-Circus*. In Jim Davies et al., editors, *IFM*, volume 4591 of *LNCS*, pages 75–97. Springer, 2007.
4. Ana Cavalcanti et al. A tutorial introduction to CSP in *Unifying Theories of Programming*. In Ana Cavalcanti et al., editors, *Refinement Techniques in Software Engineering*, volume 3167 of *LNCS*, pages 220–268. Springer, 2004.
5. Ana Cavalcanti et al. Unifying classes and processes. *SoSyM*, 4(3):277–296, 2005.
6. Ana Cavalcanti et al. Safety-critical Java in *Circus*. In Andy Wellings et al., editors, *JTRES*, pages 20–29, 2011.
7. Ana Cavalcanti et al. The Safety-critical Java memory model: A formal account. In M. J. Butler et al., editors, *FM 2011*, volume 6664 of *LNCS*, pages 246–261. Springer, 2011.
8. Ana Cavalcanti et al. The Safety-critical Java memory model formalised. *Formal Asp. Comput.*, 25(1):37–57, 2013.
9. Ana Cavalcanti et al. Safety-critical Java programs from *Circus* models. *Real-Time Systems*, 49(5):614–667, 2013.
10. Clemens Fischer. Combining Object-Z and CSP. In Adam Wolisz et al., editors, *Formale Beschreibungstechniken für verteilte Systeme*, volume 315 of *GMD-Studien*, pages 119–128, 1997.
11. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
12. C. A. R. Hoare and J. He. *Unifying Theories of Programming*. Prentice Hall, 1998.
13. Tony Hoare and Peter W. O'Hearn. Separation logic semantics for communicating processes. *ENTCS*, 212:3–25, 2008.

14. ISO. *ISO/IEC 8652:2012 Information technology — Prog. languages — Ada*, 2012.
15. Gabor Karsai. Unification or integration? The challenge of semantics in heterogeneous modeling languages. In Benoît Combemale et al., editors, *The Globalization of Modeling Languages*, pages 2–6, 2014.
16. Z. Liu et al. rCOS: Refinement of component and object systems. In F. de Boer et al., editors, *FMCO*, volume 3657 of *LNCS*, pages 183–221. Springer, 2004.
17. Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1997.
18. Robin Milner et al. A calculus of mobile processes. *Inf. Comput.*, 100:41–77, 1992.
19. FDR3 model checker. `www.cs.ox.ac.uk/projects/fdr/`.
20. Martin Odersky. Programming in Scala, 2008. Mountain View, California.
21. Marcel Oliveira et al. A denotational semantics for *Circus*. *Electr. Notes Theor. Comput. Sci.*, 187:107–123, 2007.
22. A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1997.
23. A .W .Roscoe. CSP is expressive enough for π. In C. B. Jones, A. W. Roscoe, and K. R. Wood, editors, *Reflections on the Work of C. A. R. Hoare*. Springer, 2010.
24. A. W. Roscoe. *Understanding Concurrent Systems*. Springer, 2010.
25. RTCA. Object-oriented technology and related techniques supplement to DO-178C [ED-12C] and DO-178A. Technical Report DO-332/ED-217, [ED-109A], 2011.
26. Steve Schneider and Helen Treharne. Communicating B machines. In Didier Bert et al., editors, *ZB 2002*, volume 2272 of *LNCS*, pages 416–435. Springer, 2002.
27. Adnan Sherif et al. A process algebraic framework for specification and validation of real-time systems. *Formal Asp. Comput.*, 22(2):153–191, 2010.
28. Neil Smyth. Communicating Sequential Processes in Ptolemy II. Tech. Memo. UCB/ERL M98/70, Electronics Research Laboratory, Berkeley, December 1998.
29. Jun Sun et al. Model checking CSP revisited: Introducing a process analysis toolkit. In Tiziana Margaria et al., editors, *ISoLA*, pages 307–322. Springer, 2008.
30. Xinbei Tang et al. Towards mobile processes in unifying theories. In *SEFM*, pages 44–53, 2004.
31. Xinbei Tang et al. Travelling processes. In *MPC*, volume 3125 of *LNCS*, pages 381–399. Springer, 2004.
32. inmos. *occam Programming Manual*. Prentice Hall, 1984.
33. H. Treharne and S. Schneider. How to drive a B machine. In Jonathan P. Bowen et al., editors, *ZB 2000*, volume 1878 of *LNCS*, pages 188–208. Springer, 2000.
34. Beeta Vajar et al. Mobile CSP‖B. *ECEASST*, 23, 2009.
35. P. H. Welch et al. The JCSP (CSP for Java) Home Page, 1999.
36. Peter H. Welch et al. Mobile barriers for occam-π: Semantics, implementation and application. In Jan F. Broenink et al., editors, *CPA*, pages 289–316, 2005.
37. Jim Woodcock. Engineering UToPiA—Formal semantics for CML. In *FM 2014*, volume 8442 of *LNCS*, pages 22–41. Springer, 2014.
38. Jim Woodcock et al. A concurrent language for refinement. In Andrew Butterfield et al., editors, *5th Irish Workshop on Formal Methods*, 2001.
39. Jim Woodcock et al. The semantics of *Circus*. In *ZB 2002*, volume 2272 of *LNCS*, pages 184–203. Springer, 2002.
40. Jim Woodcock et al. A tutorial introduction to designs in Unifying Theories of Programming. In Eerke A. Boiten et al., editors, *Integrated Formal Methods, IFM 2004*, volume 2999 of *LNCS*, pages 40–66. Springer, 2004.
41. Jim Woodcock et al. Features of CML: A formal modelling language for systems of systems. In *ICOSE*, pages 445–450, 2012.
42. Frank Zeyda et al. The Safety-critical Java mission model: A formal account. In Shengchao Qin et al., editors, *ICFEM 2011*, volume 6991 of *LNCS*, pages 49–65. Springer, 2011.