Modelling and Analysis Mobile Systems Using π-calculus (EFCP)

Victor Khomenko and Vasileios Germanos

Newcastle University, Newcastle upon Tyne NE1 7RU, UK
{Victor.Khomenko,V.Germanos}@ncl.ac.uk

Abstract. Reference passing systems, like mobile and reconfigurable systems are common nowadays. The common feature of such systems is the possibility to form dynamic logical connections between the individual modules. However, such systems are very difficult to verify, as their logical structure is dynamic. Traditionally, decidable fragments of π-calculus, e.g. the well-known Finite Control Processes (FCP), are used for formal modelling of reference passing systems. Unfortunately, FCPs allow only 'global' concurrency between processes, and thus cannot naturally express scenarios involving 'local' concurrency inside a process, such as multicast. In this paper we propose Extended Finite Control Processes (EFCP), which are more convenient for practical modelling. Moreover, an almost linear translation of EFCPs to FCPs is developed, which enables efficient model checking of EFCPs.

Keywords: π-calculus, finite control process, extended finite control process, reconfigurable systems, mobile systems, model checking.

1 Introduction

Many contemporary systems enjoy a number of features that significantly increase their power, usability and flexibility:

- *Dynamic reconfigurability*: The overall structure of many existing systems is flexible. Nodes in ad-hoc networks can dynamically appear or disappear; individual cores in Networks-on-Chip can be temporarily shut down to save power; resilient systems have to continue to deliver (reduced) functionality even if some of their modules develop faults.
- *Logical mobility*: Mobile systems permeate our lives and are becoming ever more important. Ad-hoc networks, where devices like mobile phones and laptops form dynamic connections are common nowadays, and the vision of pervasive (ubiquitous) computing [1], where several devices are simultaneously engaged in interaction with the user and each other, forming dynamic links, is quickly becoming a reality.
- *Dynamic allocation of resources*: It is often the case that a system has several instances of the same resource (e.g., network servers or processor cores in a microchip) that have to be dynamically allocated to tasks depending on the current workload, power mode, priorities of the clients, etc.

The common feature of such systems is the possibility to form dynamic logical connections between the individual modules. It is implemented using *reference*

passing. A module can become aware of another module by receiving a *reference* (e.g., in the form of a network address) to it, which enables subsequent communication between these modules. This can be thought of as a new (logical) *channel* dynamically created between these modules. We will refer to such systems as *Reference Passing Systems* (RPS).

As people are increasingly dependent on the correct functionality of RPSs, the cost incurred by design errors in such systems can be extremely high. However, even the conventional concurrent systems are notoriously difficult to design correctly because of the complexity of their behaviour, and reference passing adds another layer of complexity due to the logical structure of the system becoming dynamical. Hence, computer-aided formal verification has to be employed in the design process to ensure the correct behaviour of RPSs. However, validation of such systems is almost always limited to simulation/testing, as their formal verification is very difficult due to either the inability of the traditional verification techniques to express reference passing[1] (at least in a natural way) or by poor scalability of the existing verification techniques for RPSs.

This is very unfortunate: As many safety-critical systems must be resilient (and hence reconfigurable), they are often RPSs and thus have very complicated behaviour. Hence, for such systems the design errors are both very likely and very costly, and formal verification must be an essential design step. This paper addresses this problem by developing a formalism that can specify RPSs and make their formal verification feasible.

There is a number of formalisms that are suitable for specification of RPSs. The main considerations and trade-offs in choosing an appropriate formalism are its expressiveness and the tractability of the associated verification techniques. Expressive formalisms (like π-calculus [2] and Ambient Calculus [3]) are Turing powerful and so not decidable in general. Fortunately, the ability to pass references *per se* does not lead to undecidability, and it is possible to put in place some restrictions (e.g., finiteness of the control) that would guarantee decidability, while still maintaining a reasonable modelling power.

Finite Control Processes (FCP) [4] are a fragment of π-calculus, where the system is constructed as a parallel composition of sequential entities (threads). Each sequential entity has a finite control, and the number of such entities is bounded in advance. The entities communicate synchronously via channels, and have the possibility to create new channels dynamically and to send channels via channels.

As π-calculus is the most well-known formalism suitable for RPS specification, we fix FCPs (as a natural decidable and reasonably expressive fragment of π-calculus) as the primary RPS specification formalism, from which a new extension will be derived.

One common feature of RPSs is *multicast*. That is, data can be transmitted from one source to more than one destinations concurrently. Using FCPs to model such systems is not inconvenient because local concurrency inside the

[1] Some existing tools like SPIN allow to send channels via channels; however, they do not allow dynamic creation of new channels, which is often essential in RPSs.

thread is forbidden. Thus, we propose a new subclass of π-calculus, the *Extended Finite Control Processes* (EFCP), that allows to model multicast in natural way and is still amenable to model checking.

The formal verification of RPSs expressed as EFCPs can be done in stepwise manner. Firstly, a translation from EFCP to FCP is performed as explained in this paper. The size of the resulting FCP is quadratic in the worst case, but often linear in practice (see Section 4.4). Then, the resulting FCP can be translated to a Petri net of polynomial size [5], and for the latter there are already efficient verification techniques.

2 Basic notions

In π-calculus [6,7] and FCPs [4], threads communicate via synchronous message exchange. The key idea of the formalism is that messages and the channels they are sent on have the same type: they are just *names* from some set $\Phi \stackrel{\text{df}}{=} \{a, b, x, y, i, f, r, \ldots\}$, which are the simplest entities of the π-calculus. This means a name that has been received as message in one communication may serve as channel in a later interaction. To communicate, processes consume *prefixes* π of the form

$$\pi ::= \overline{a}\langle b\rangle \;\mid\; a(x) \;\mid\; \tau.$$

The *output prefix* $\overline{a}\langle b\rangle$ sends name b along channel a. The *input prefix* $a(i)$ receives a name that replaces i on channel a. The input and output actions are called *visible actions* and prefix τ stands for a *silent action.*

Threads, also called *sequential processes*, are constructed as follows. A *choice process* $\sum_{i\in I} \pi_i.S_i$ over a finite set of indices I executes a prefix π_i and then behaves like S_i. The special case of choices over an empty index set $I = \emptyset$ is denoted by $\mathbf{0}$ — such a process has no behaviour. Moreover, when $|I| = 1$ we drop Σ. We use $\bigodot$ to refer to iterated prefixing, e.g. $\overline{a_1}\langle b_1\rangle.\overline{a_2}\langle b_2\rangle.\overline{a_3}\langle b_3\rangle.\overline{a_4}\langle b_4\rangle.\mathbf{0}$ can be written as $\left(\bigodot_{i=1}^{4} \overline{a_i}\langle b_i\rangle\right).\mathbf{0}$. A *restriction* $\nu r : S$ generates a name r that is different from all other names in the system. We denote a sequence of restrictions $\nu r_1 \ldots \nu r_k$ by $\nu\tilde{r}$ with $\tilde{r} = r_1 \ldots r_k$. To implement parameterised recursion, we use *calls to process identifiers* $K\lfloor\tilde{a}\rfloor$. We defer the explanation of this construct for a moment. To sum up, FCP *threads* take the form

$$S ::= K\lfloor\tilde{a}\rfloor \;\mid\; \textstyle\sum_{i\in I} \pi_i.S_i \;\mid\; \nu r : S.$$

A *finite control process (FCP)* F is a parallel composition of a fixed number of threads:

$$F ::= \nu\tilde{a} : (S_1 \mid \ldots \mid S_n).$$

Note that in FCPs the *parallel composition* operator $\mid$ is allowed at the top level, but inside the threads, whereas in general π-calculus there is no such

restriction. We use Π to denote iterated parallel composition, e.g. the above definition of an FCP can be re-written as $F ::= \nu\tilde{a} : \prod_{i=1}^{n} S_i$.

Our presentation of parameterised recursion using calls $K\lfloor\tilde{a}\rfloor$ follows [7]. Process identifiers K are taken from some set $\Psi \stackrel{\text{df}}{=} \{H, K, L, \ldots\}$ and have a *defining equation* $K(\tilde{f}) := S$. Here S can be understood as the implementation of identifier K. The process has a list of *formal parameters* $\tilde{f} = f_1, \ldots, f_k$ that are replaced by *factual parameters* $\tilde{a} = a_1, \ldots, a_k$ when a call $K\lfloor\tilde{a}\rfloor$ is executed. Note that both lists $\tilde{a}$ and $\tilde{f}$ have the same length. When we talk about an *FCP specification* F, we mean process F with all its defining equations.

To implement the replacement of $\tilde{f}$ by $\tilde{a}$ in calls to process identifiers, we use *substitutions*. A substitution is a function $\sigma : \Phi \to \Phi$ that maps names to names. If we make domain and codomain explicit, $\sigma : A \to B$ with $A, B \subseteq \Phi$, we require $\sigma(a) \in B$ for all $a \in A$ and $\sigma(x) = x$ for all $x \in \Phi \setminus A$. We use $\{\tilde{a}/\tilde{f}\}$ to denote the substitution $\sigma : \tilde{f} \to \tilde{a}$ with $\sigma(f_i) \stackrel{\text{df}}{=} a_i$ for $i \in \{1, \ldots, k\}$. The *application of substitution* σ *to* S is denoted by $S\sigma$ and defined in the standard way [7].

Input prefix $a(i)$ and restriction νr *bind* the names i and r, respectively. The *set of bound names* in a process $P = S$ or $P = F$ is $bn\,(P)$. A name which is not bound is *free*, and the *set of free names* in P is $fn\,(P)$. We permit α-conversion of bound names. Therefore, w.l.o.g., we make the following assumptions common in π-calculus theory and collectively referred to as *no clash* **(NOCLASH)** [5] henceforth. For every π-calculus specification, we require that:

- a name is bound at most once;
- a name is used at most once in formal parameter lists;
- the sets of bound names, free names and formal parameters are pairwise disjoint;
- if a substitution $\sigma = \{\tilde{a}/\tilde{x}\}$ is applied to P then $bn\,(P)$ and $\tilde{a} \cup \tilde{x}$ are disjoint.

Assuming **(NOCLASH)**, the names occurring in a π-calculus specification F can be partitioned into the following sets:

$\mathcal{P}$ public names that are free in F;
$\mathcal{R}$ names bound by restriction operators;
$\mathcal{I}$ names bound by input prefixes;
$\mathcal{F}$ names used as formal parameters in defining equations.

The *size* of a π-calculus specification is defined as the size of its initial term plus the sizes of the defining equations. The corresponding function $\|\cdot\|$ measures the number of channel names, process identifiers, the lengths of parameter lists, and the number of operators in use:

$$\|\mathbf{0}\| \stackrel{\text{df}}{=} 1$$
$$\|K\lfloor\tilde{a}\rfloor\| \stackrel{\text{df}}{=} 1 + |\tilde{a}|$$
$$\|\nu r : P\| \stackrel{\text{df}}{=} 1 + \|P\|$$
$$\|K(\tilde{f}) := S\| \stackrel{\text{df}}{=} 1 + |\tilde{f}| + \|S\|$$

$$\|\textstyle\sum_{i\in I}\pi_i.S_i\| \stackrel{\text{df}}{=} 3|I| - 1 + \sum_{i\in I}\|S_i\|$$

$$\|\prod_{i=1}^{n} S_i\| \stackrel{\text{df}}{=} n - 1 + \textstyle\sum_{i=1}^{n}\|S_i\|$$

It is not so simple to define reduction on terms of π-calculus, because two subterms of a process-term may interact despite the fact that they may not be adjacent. To define the behaviour of a process and the reduction on process terms, we rely on a relation called *structural congruence* $\equiv$. It is the smallest congruence where α-conversion of bound names is allowed, $+$ and $|$ are commutative and associative with $\mathbf{0}$ as the neutral element, and the following laws for restriction hold:

$$\nu x : \mathbf{0} \equiv \mathbf{0}$$

$$\nu x : \nu y : P \equiv \nu y : \nu x : P$$

$$\nu x : (P \,|\, Q) \equiv P \,|\, (\nu x : Q) \text{ if } x \notin fn\,(P)$$

The behaviour of π-calculus processes is determined by the *reaction relation* $\rightarrow$. The reaction relations are defined by inference rules [6, 7]:

(Tau) $\tau.S + M \rightarrow S$ (React) $(x(y).S + M) \,|\, (\overline{x}\langle z\rangle.S' + N) \rightarrow S\{z/y\} \,|\, S'$

(Res) $\dfrac{P \rightarrow P'}{\nu a : P \rightarrow \nu a : P'}$ (Struct) $\dfrac{P \rightarrow P'}{Q \rightarrow Q'}$ if $P \equiv Q$ and $P' \equiv Q'$

(Par) $\dfrac{P \rightarrow P'}{P \,|\, Q \rightarrow P' \,|\, Q}$ (Const) $K\lfloor\tilde{a}\rfloor \rightarrow S\{\tilde{a}/\tilde{f}\}$ if $K(\tilde{f}) := S$

The rule (Tau) is an axiom for silent steps. (React) describes the communication of two parallel threads, consuming their send and receive actions respectively and continuing as a process, where the name y is substituted by z in the receiving thread S. (Const) describes identifier calls, likewise using a substitution. The remaining rules define $\rightarrow$ to be closed under structural congruence, parallel composition and restriction. By $\mathcal{R}(F)$ we denote the set of all processes reachable from F. The *transition system* of FCP F factorises the reachable processes along structural congruence.

2.1 Normal form assumptions

We require that the sets of process identifiers called (both directly from F and indirectly from defining equations) by different threads are disjoint. This restriction corresponds to the notion of a *safe* FCP [8] and can be achieved by replicating some defining equations. The resulting specification is bisimilar with F and has the size $O(n\|F\|) = O(\|F\|^2)$. We illustrate the construction on the following example of an FCP specification (left) together with its replicated version (right):

$K(f_1, f_2) := \tau.L(f_1, f_2)$	$K^1(f_1^1, f_2^1) := \tau.L^1(f_1^1, f_2^1)$
$L(f_3, f_4) := \tau.K(f_3, f_4)$	$L^1(f_3^1, f_4^1) := \tau.K^1(f_3^1, f_4^1)$
	$K^2(f_1^2, f_2^2) := \tau.L^2(f_1^2, f_2^2)$
	$L^2(f_3^2, f_4^2) := \tau.K^2(f_3^2, f_4^2)$
	$K^3(f_1^3, f_2^3) := \tau.L^3(f_1^3, f_2^3)$
	$L^3(f_3^3, f_4^3) := \tau.K^3(f_3^3, f_4^3)$
$K\lfloor a, b\rfloor \mid K\lfloor b, c\rfloor \mid L\lfloor a, c\rfloor$	$K^1\lfloor a, b\rfloor \mid K^2\lfloor b, c\rfloor \mid L^3\lfloor a, c\rfloor$

Intuitively, in the resulting FCP specification each thread has its own set of defining equations. This normal form is applicable also to EFCPs introduced in Section 4.2.

2.2 Match and mismatch operators

The match and mismatch operators are a common extension of π-calculus [5]. Intuitively, the process $[x = y].P$ behaves as P if x and y refer to the same channel, and as $\mathbf{0}$ otherwise, and the process $[x \neq y].P$ behaves as P if x and y refer to different channels, and as $\mathbf{0}$ otherwise.

2.3 Polyadic communication

Polyadic communication can be used to make modelling more convenient. Using polyadic communication *tuples* of names can be exchanged in a single reaction. More precisely, a sending prefix $\overline{a}\langle x_1 \dots x_m\rangle$ (with $m \geq 0$) and a receiving prefix $a(y_1 \dots y_n)$ (with $n \geq 0$ and all y_i being different names) can synchronise iff $m = n$, and after synchronisation each y_i is replaced by x_i, $\{y_i/x_i\}$. Formally,

$$\text{(React) } (a(\tilde{y})\,;P_1 + Q_1)\,|(\overline{a}\langle\tilde{x}\rangle\,;P_2 + Q_2) \rightarrow P_1\{\tilde{x}/\tilde{y}\}\,|\,P_2 \;if \mid \tilde{y} \mid = \mid \tilde{x} \mid$$

3 Extended finite control processes

This section introduces the *Extended Finite Control Processes* (*EFCP*), which add new features to FCPs, in particular limited local concurrency within a thread, while still allowing one to formally verify such systems. Thus, practical modelling of reconfigurable systems becomes more convenient.

The threads in an FCP can communicate synchronously via channels, and are able to create new channels dynamically and send channels via channels. However, FCP threads are *sequential* processes, without any 'local' concurrency inside them. This makes FCPs too restrictive when one wants to model scenarios involving local concurrency within a thread, for instance, in case of routing protocols in multi-core processor systems. An essential feature of such protocols is multicast, i.e. the ability of a core to send a datum to several destinations concurrently.

EFCPs are sufficient for modelling many practical reconfigurable systems. Moreover, since an efficient translation from FCPs to safe Petri nets exists [5], it can be reused for EFCPs (via an intermediate translation to FCPs). Hence efficient formal verification algorithms for Petri nets can be used to verify EFCPs.

For example, the following process is an FCP, which is a parallel composition of three sequential processes (threads):

$$K_1 := a(x) \,.\, \overline{x}\langle b\rangle \,.\, \mathbf{0}$$

$$K_2 := \nu u : \overline{a}\langle u\rangle \,.\, \overline{w}\langle u\rangle \,.\, \mathbf{0}$$

$$K_3 := w(t) \,.\, t(v) \,.\, \mathbf{0}$$

$$K_1 \,|\, K_2 \,|\, K_3$$

Note that in FCPs the parallel composition operator '|' can be used only in the initial term, i.e. the threads are fully sequential and their number is bounded in advance.

To be able to model a wide range of RPSs, a higher degree of freedom in the syntax is required to specify their various behavioural scenarios. To that end, an extension of FCPs, the EFCP, is introduced, which allows local concurrency and replaces the prefixing operator '.' with a more powerful *sequential composition* operator ';'. The full EFCP syntax is defined in Section 3.1, and an example is given below.

$$K_1 := \nu r : \big((\overline{a}\langle r\rangle \,|\, \overline{b}\langle r\rangle) \,;\, (r(x) \,|\, r(y))\big)$$

$$K_2 := a(z) \,;\, \overline{z}\langle c\rangle$$

$$K_3 := b(w) \,;\, \overline{w}\langle d\rangle$$

$$K_1 \,|\, K_2 \,|\, K_3$$

3.1 The Syntax of Extended Finite Control Processes

To define the EFCP syntax the notion of *finite processes* is required. Such processes have special syntax ensuring that the number of actions they can execute is bounded in advance.

The arguments of the parallel composition operator, when used inside a thread, are limited to finite processes only. Similarly, the left hand side of sequential composition must be a finite process, but the right hand side is not required to be such.

This new subcalculus is defined by a context-free grammar consisting of two sub-grammars, one for the *finite executed processes* and one for generic processes.

Definition 1 (Grammar for finite processes).

$$F ::= \mathbf{0} \;\mid\; \pi \;\mid\; F + F \;\mid\; F \,|\, F \;\mid\; \nu\tilde{r} : F \;\mid\; F \,;\, F$$

The syntax of generic processes includes that of finite processes, but also allows for extra features like recursive definitions.

Definition 2 (EFCP grammar). *Let F be a finite process defined above. The syntax of an EFCP thread is then*

$$P ::= K\lfloor\tilde{x}\rfloor \mid F \mid P + P \mid \nu\tilde{r} : P \mid F\,;P$$

An EFCP specification is comprised of a set of defining equations of the form $K(\tilde{f}) := P$ and an initial term of the form $\nu\tilde{r} : \prod_{i=1}^{n} P_i$, where P and all P_i are EFCP threads.

Note that an EFCP thread cannot contain the construction $P \,|\, P$ (only the initial term can have it), but it can contain $F \,|\, F$.

3.2 Structural congruence and operational semantics

The structural congruence relation is used in the definition of the behaviour of a process term. The choice '+' and the parallel composition '|' are commutative and associative with **0** as the neutral element. Sequential composition ';' is associative with **0** as the neutral element, but not commutative.

Definition 3 (Structural congruence). *The structural congruence $\equiv$ is the smallest congruence that satisfies the following axioms:*

Alpha-convertion:

$\nu r : P \equiv \nu r' : P\{r'/r\}$ if $r' \notin fn\,(P)$.

$a(x)\,;P \equiv a(x')\,;P\{x'/x\}$ if $x' \notin fn\,(P)$.

Laws for sequential composition:

$$\mathbf{0}\,;P \equiv P$$

$$F\,;\mathbf{0} \equiv F$$

$$(F_1\,;F_2)\,;P \equiv F_1\,;(F_2\,;P)$$

Laws for restriction:

$$\nu r : \mathbf{0} \equiv \mathbf{0}$$

$$\nu\alpha : \nu\beta : P \equiv \nu\beta : \nu\alpha : P$$

Laws for parallel composition:

$$P_1 \,|\, (P_2 \,|\, P_3) \equiv (P_1 \,|\, P_2) \,|\, P_3$$

$$P_1 \,|\, P_2 \equiv P_2 \,|\, P_1$$

$$P \mid \mathbf{0} \equiv P$$

Laws for summation:

$$P_1 + (P_2 + P_3) \equiv (P_1 + P_2) + P_3$$

$$P_1 + P_2 \equiv P_2 + P_1$$

$$P + \mathbf{0} \equiv P$$

Definition 4 (Structural operational semantics). *The transition system of EFCP is defined by the following rules:*

$$\text{(Seq)}\ \frac{F \to F'}{F\,;Q \to F'\,;Q} \qquad \text{(Tau)}\ \tau\,;P + Q \to P$$

$$\text{(Res)}\ \frac{P \to P'}{\nu r : P \to \nu r : P'} \qquad \text{(Struct)}\ \frac{P \to P'}{Q \to Q'}\ \text{ if } P \equiv Q \text{ and } P' \equiv Q'$$

$$\text{(Par)}\ \frac{P \to P'}{P \mid Q \to P' \mid Q} \qquad \text{(Const)}\ K\lfloor \tilde{a} \rfloor \to P\{\tilde{a}/\tilde{f}\}\ \text{ if } K(\tilde{f}) := P$$

$$\text{(React)}\ (x(y)\,;P_1 + Q_1) \mid (\overline{x}\langle z\rangle\,;P_2 + Q_2) \to P_1\{z/y\} \mid P_2$$

Note that these rules are similar to the π-calculus rules in Section 2 with the exception of the (Seq) rule expressing the semantics of our more powerful sequential composition operator ' ; '.

4 Translation of EFCPs to FCPs

In this section, a formal description of the new formalism is presented, and an almost linear translation from EFCP to FCP is introduced. The purpose of translating EFCP to FCP is for the latter to be translated to safe low-level Petri nets [8], for which efficient verification techniques can be applied.

4.1 Description

The translation has to eliminate the parallel composition operator inside threads and the use of sequential composition. Since an FCP consists of sequential processes (*threads*), any thread of an EFCP that is not sequential must be converted to a sequential one. This can be done by shifting all the concurrency to the initial term. Moreover, sequential composition has to be replaced by prefixing. To avoid blow up in size, new declarations are introduced during this process.

To ensure that the order of actions is preserved and that the context (binding of channel names) is correct, extra communication between threads may be

required. New process definitions are introduced in two cases. The first is when local concurrency exists within a thread, e.g.:

$$\begin{aligned} K[x] &:= \nu r : ((\overline{a}\langle x\rangle \,|\, \overline{b}\langle r\rangle)\,;\tau) \\ &K\lfloor u \rfloor \end{aligned}$$

The translation result is:

$$\begin{aligned} K[x] &:= \nu r : (\overline{begin_1}\langle x\rangle \,.\, \overline{begin_2}\langle r\rangle \,.\, end_1() \,.\, end_2() \,.\, \tau \,.\, \mathbf{0}) \\ K_1 &:= begin_1(x) \,.\, \overline{a}\langle x\rangle \,.\, \overline{end_1}\langle\rangle \,.\, K_1 \\ K_2 &:= begin_2(r) \,.\, \overline{b}\langle r\rangle \,.\, \overline{end_2}\langle\rangle \,.\, K_2 \\ &K\lfloor u \rfloor \,|\, K_1 \,|\, K_2 \end{aligned}$$

Here K_1 and K_2 are fresh PIDs and $begin_1$, $begin_2$, end_1, end_2 are fresh public names. Note that the necessary context is passed to the auxiliary FCP threads K_1 and K_2 using communication on $begin_1$ and $begin_2$.

The second case is when there is a sequential composition with a non-trivial left-hand side, e.g.:

$$\begin{aligned} K[x] &:= \nu r : \big(\underbrace{(\overline{a}\langle x\rangle + \overline{b}\langle r\rangle)}_{\text{l.h.s.}} \,;\, \underbrace{(\overline{c}\langle x\rangle + \overline{d}\langle r\rangle)}_{\text{r.h.s.}}\big) \\ &K\lfloor u \rfloor \end{aligned}$$

This process is translated as follows:

$$\begin{aligned} K[x] &:= \nu r : (\overline{a}\langle x\rangle \,.\, K_1\lfloor x, r\rfloor + \overline{b}\langle r\rangle \,.\, K_1\lfloor x, r\rfloor) \\ K_1[x,r] &:= \overline{c}\langle x\rangle \,.\, \mathbf{0} \;+\; \overline{d}\langle r\rangle \,.\, \mathbf{0} \\ &K\lfloor u \rfloor \end{aligned}$$

Here K_1 is a fresh PID. Note that the initial term did not change and that the context is passed via parameters of a call.

4.2 Formal Definition of EFCP to FCP Translation

In this section, the translation is defined in a formal way. EFCP has the form

$$\begin{aligned} K_1[\tilde{x_1}] &:= P_1 \\ &\vdots \\ K_n[\tilde{x_n}] &:= P_n \\ \nu\tilde{r} &: (Q_1 \,|\, \ldots \,|\, Q_k) \end{aligned}$$

where the syntax of each P_i is given by Definitions 1 and 2, and we assume that no Q_i in the initial term uses '|' or ';'. Note that the EFCP is assumed to be safe

and to satisfy **(NOCLASH)**. Safe EFCPs are defined similarly to safe FCPs, see Sect. 2.1.

Definition 5 (Translation). *The translation $[\![\cdot]\!]_B$ from EFCP to FCP is defined inductively on the syntactical structure of the EFCP. Here B is the parameter of the translation. It defines the* context*, i.e. the set of names that were bound prior to the occurrence of the term to be translated. The translation is applied to each process declaration separately:*

$$[\![K[\tilde{x}] := P]\!]_{\emptyset} \stackrel{\text{df}}{=} K[\tilde{x}] := [\![P]\!]_{\tilde{x} \cap fn(P)} \tag{Decl}$$

Base cases:

$$[\![\mathbf{0}]\!]_B \stackrel{\text{df}}{=} \mathbf{0} \tag{Stop}$$

$$[\![\pi]\!]_B \stackrel{\text{df}}{=} \pi \,.\, \mathbf{0} \tag{Pref}$$

$$[\![K\lfloor\tilde{x}\rfloor]\!]_B \stackrel{\text{df}}{=} K\lfloor\tilde{x}\rfloor \tag{Call}$$

Parallel composition:

$$\left[\!\!\left[\prod_{i=1}^{k} P_i\right]\!\!\right]_B \stackrel{\text{df}}{=} \left(\bigodot_{i=1}^{n} \overline{begin_i}\langle B \cap fn\,(P_i)\rangle\right) . \bigodot_{i=1}^{n} end_i() \tag{Par}$$

where $begin_i$ and end_i are fresh public names and K_i are fresh PIDs
$K_i := begin_i(B \cap fn\,(P_i)).[\![P_i \,;\, \overline{end_i}\langle\rangle \,;\, K_i]\!]_{B \cap fn(P_i)}, \quad i = 1 \ldots k$
$\prod_{i=1}^{k} K_i$ *is added concurrently to the initial term.*

Restriction:

$$[\![\nu\tilde{r} : P]\!]_B \stackrel{\text{df}}{=} \nu\tilde{r} : [\![P]\!]_{(B \cup \tilde{r}) \cap fn(P)} \tag{Restr}$$

Choice composition:

$$\left[\!\!\left[\sum_{i=1}^{k} P_i\right]\!\!\right]_B \stackrel{\text{df}}{=} \sum_{i=1}^{k} [\![P_i]\!]_{B \cap fn(P_i)} \tag{Choice}$$

Match and mismatch:

$$[\![[a = x] \,.\, P]\!]_B \stackrel{\text{df}}{=} [a = x] \,.\, [\![P]\!]_{B \cap fn(P)} \tag{Match}$$

$$[\![[a \neq x] \,.\, P]\!]_B \stackrel{\text{df}}{=} [a \neq x] \,.\, [\![P]\!]_{B \cap fn(P)} \tag{Mismatch}$$

Sequential composition base cases:

$$[\![\mathbf{0} \,;\, P]\!]_B \stackrel{\text{df}}{=} [\![P]\!]_B \tag{SeqStop}$$

$$[\![\tau \,;\, P]\!]_B \stackrel{\text{df}}{=} \tau \,.\, [\![P]\!]_B \tag{SeqTau}$$

$$\left[\!\left[\overline{a}\langle\tilde{b}\rangle \;;\, P\right]\!\right]_B \stackrel{\text{df}}{=} \overline{a}\langle\tilde{b}\rangle.[\![P]\!]_{B\cap fn(P)} \tag{SeqSend}$$

$$\left[\!\left[a(\tilde{b}) \;;\, P\right]\!\right]_B \stackrel{\text{df}}{=} a(\tilde{b}).[\![P]\!]_{(B\cup\tilde{b})\cap fn(P)} \tag{SeqRec}$$

Sequential composition inductive cases:

$$\left[\!\left[\left(\sum_{i=1}^{k} P_i\right) \;;\, P\right]\!\right]_B \stackrel{\text{df}}{=} \left[\!\left[\sum_{i=1}^{k}\Big(P_i \;;\, K[B\cap fn\,(P)]\Big)\right]\!\right]_B \tag{SeqChoice}$$

where K is a fresh PID (not added to the initial process)
$K[B\cap fn\,(P)] := [\![P]\!]_{B\cap fn(P)}$

$$\left[\!\left[\left(\prod_{i=1}^{k} P_i\right) \;;\, P\right]\!\right]_B \stackrel{\text{df}}{=} \left[\!\left[\prod_{i=1}^{k} P_i\right]\!\right]_{B\cap\bigcup_{i=1}^{k} fn(P_i)} . \; [\![P]\!]_{B\cap fn(P)} \tag{SeqPar}$$

$$[\![(\nu\tilde{r} : P) \;;\, P']\!]_B \stackrel{\text{df}}{=} [\![\nu\tilde{r} : (P \;;\, P')]\!]_B \tag{SeqRestr}$$

4.3 An Example of translation from EFCP to FCP

The following EFCP process models a client that communicates with a server.

$$\begin{aligned} C\lfloor url, ip\rfloor &:= \nu q : (\overline{url}\langle ip, q\rangle \;;\, ip(a) \;;\, C\lfloor url, ip\rfloor) \\ S\lfloor url'\rfloor &:= url'(ip', q') \;;\, \nu x : ((\nu r : \overline{x}\langle r\rangle \;;\, \tau \;;\, r(a) \;;\, \overline{ip'}\langle a\rangle) \,|\, \\ &\qquad x(v) \;;\, (\tau \;+\; \tau) \;;\, \overline{v}\langle a'\rangle) \;;\, S\lfloor url'\rfloor \\ \nu url'', ip'' &: (S\lfloor url''\rfloor \,|\, C\lfloor url'', ip''\rfloor) \end{aligned}$$

The server is located at some URL, $S\lfloor url'\rfloor$. A client can contact it by sending its IP address ip on the channel url. At the same time it sends a question, q, to the server, $\overline{url}\langle ip, q\rangle$. The client generates and sends a different question each time, thus q is a restricted name. The client's IP address and the question are received by the server and are stored as ip' and q', $url'(ip', q')$. The server runs two computational threads, which communicate with one another via a temporary internal channel x and produce an answer, and one of them sends the answer to the client on ip', $\nu x : ((\nu r : \overline{x}\langle r\rangle \;;\, \tau \;;\, r(a) \;;\, \overline{ip'}\langle a\rangle) \,|\, x(v) \;;\, (\tau \;+\; \tau) \;;\, \overline{v}\langle a'\rangle)$, at which point the server repeats its behaviour by calling $S\lfloor url'\rfloor$. The client receives the answer, $ip(a)$, and is able to contact the server again, $C\lfloor url, ip\rfloor$.

This specification is a safe EFCP satisfying **(NOCLASH)**. Now, we translate it to FCP in a stepwise manner. First, the declaration of C is translated according to the rules of Definition 5:

$$
\begin{aligned}
&[\![C\lfloor url, ip\rfloor := \nu q : (\overline{url}\langle ip, q\rangle \,;\, ip(a) \,;\, C\lfloor url, ip\rfloor)]\!]_{\emptyset} && = \textit{by Decl}\\
&C[url, ip] := [\![\nu q : (\overline{url}\langle ip, q\rangle \,;\, ip(a) \,;\, C\lfloor url, ip\rfloor)]\!]_{\{url,ip\}} && = \textit{by Restr}\\
&C[url, ip] := \nu q : ([\![\overline{url}\langle ip, q\rangle \,;\, ip(a) \,;\, C\lfloor url, ip\rfloor]\!]_{\{url,ip,q\}}) && = \textit{by SeqSend}\\
&C[url, ip] := \nu q : (\overline{url}\langle ip, q\rangle \;.\; [\![ip(a) \,;\, C\lfloor url, ip\rfloor]\!]_{\{url,ip\}}) && = \textit{by SeqRec}\\
&C[url, ip] := \nu q : (\overline{url}\langle ip, q\rangle \;.\; ip(a) \;.\; [\![C\lfloor url, ip\rfloor]\!]_{\{url,ip\}}) && = \textit{by Call}\\
&C[url, ip] := \nu q : \overline{url}\langle ip, q\rangle \;.\; ip(a) \;.\; C\lfloor url, ip\rfloor
\end{aligned}
$$

Finally, client process has been converted to FCP. It is now the server's turn to be translated to FCP. Again, the same procedure is followed.

$$
\begin{aligned}
[\![S\lfloor url'\rfloor := {} & url'(ip', q') \,;\, \nu x : \big((\nu r : \overline{x}\langle r\rangle \,;\, \tau \,;\, r(a) \,;\, \overline{ip'}\langle a\rangle) \,|\\
& x(v) \,;\, (\tau \;+\; \tau) \,;\, \overline{v}\langle a'\rangle\big) \,;\, S\lfloor url'\rfloor]\!]_{\emptyset} && = \textit{by Decl}\\
S[url'] := {} & [\![url'(ip', q') \,;\, \nu x : \big((\nu r : \overline{x}\langle r\rangle \,;\, \tau \,;\, r(a) \,;\, \overline{ip'}\langle a\rangle) \,|\\
& x(v) \,;\, (\tau \;+\; \tau) \,;\, \overline{v}\langle a'\rangle\big) \,;\, S\lfloor url'\rfloor]\!]_{\{url'\}} && = \textit{by SeqRec}\\
S[url'] := {} & url'(ip', q') \;.\; [\![\nu x : \big((\nu r : \overline{x}\langle r\rangle \,;\, \tau \,;\, r(a) \,;\, \overline{ip'}\langle a\rangle) \,|\\
& x(v) \,;\, (\tau \;+\; \tau) \,;\, \overline{v}\langle a'\rangle\big) \,;\, S\lfloor url'\rfloor]\!]_{\{ip'\}} && = \textit{by SeqRestr}\\
S[url'] := {} & url'(ip', q') \;.\; [\![\nu x : \big((\nu r : \overline{x}\langle r\rangle \,;\, \tau \,;\, r(a) \,;\, \overline{ip'}\langle a\rangle) \,|\\
& x(v) \,;\, (\tau \;+\; \tau) \,;\, \overline{v}\langle a'\rangle \,;\, S\lfloor url'\rfloor\big)]\!]_{\{ip'\}} && = \textit{by Restr}\\
S[url'] := {} & url'(ip', q') \;.\; \nu x : [\![(\nu r : \overline{x}\langle r\rangle \,;\, \tau \,;\, r(a) \,;\, \overline{ip'}\langle a\rangle) \,|\\
& x(v) \,;\, (\tau \;+\; \tau) \,;\, \overline{v}\langle a'\rangle \,;\, S\lfloor url'\rfloor]\!]_{\{ip',x\}} && = \textit{by SeqPar}\\
S[url'] := {} & url'(ip', q') \;.\; \nu x : \big([\![(\nu r : \overline{x}\langle r\rangle \,;\, \tau \,;\, r(a) \,;\, \overline{ip'}\langle a\rangle) \,|\\
& x(v) \,;\, (\tau \;+\; \tau) \,;\, \overline{v}\langle a'\rangle]\!]_{\{ip',x\}} \;.\; [\![S\lfloor url'\rfloor]\!]_{\emptyset}\big) && = \textit{by Call}\\
S[url'] := {} & url'(ip', q') \;.\; \nu x : \big([\![(\nu r : \overline{x}\langle r\rangle \,;\, \tau \,;\, r(a) \,;\, \overline{ip'}\langle a\rangle) \,|\\
& x(v) \,;\, (\tau \;+\; \tau) \,;\, \overline{v}\langle a'\rangle]\!]_{\{ip',x\}} \;.\; S\lfloor url'\rfloor\big) && = \textit{by Par}\\
S[url'] := {} & url'(ip', q') \;.\; \nu x : \overline{begin_1}\langle x, ip'\rangle \;.\; \overline{begin_2}\langle x\rangle \;.\\
& end_1() \;.\; end_2() \;.\; S\lfloor url'\rfloor
\end{aligned}
$$

Here $begin_1$, $begin_2$, end_1, end_2 are fresh public names, K_1 and K_2 are fresh PIDs, and $(K_1 \mid K_2)$ is added to the initial process.

$$K_1 := begin_1(x, ip') \,.\, [\![\nu r : \overline{x}\langle r\rangle \,;\, \tau \,;\, r(a) \,;\, \overline{ip'}\langle a\rangle \,;\, \overline{end_1}\langle\rangle \,;\, K_1]\!]_{\{ip', x\}} \qquad = by\ Restr$$

$$K_1 := begin_1(x, ip') \,.\, \nu r : [\![\overline{x}\langle r\rangle \,;\, \tau \,;\, r(a) \,;\, \overline{ip'}\langle a\rangle \,;\, \overline{end_1}\langle\rangle \,;\, K_1]\!]_{\{ip', x, r\}} \qquad = by\ SeqSend$$

$$K_1 := begin_1(x, ip') \,.\, \nu r : \overline{x}\langle r\rangle \,.\, [\![\tau \,;\, r(a) \,;\, \overline{ip'}\langle a\rangle \,;\, \overline{end_1}\langle\rangle \,;\, K_1]\!]_{\{ip', r\}} \qquad = by\ SeqTau$$

$$K_1 := begin_1(x, ip') \,.\, \nu r : \overline{x}\langle r\rangle \,.\, \tau \,.\, [\![r(a) \,;\, \overline{ip'}\langle a\rangle \,;\, \overline{end_1}\langle\rangle \,;\, K_1]\!]_{\{ip', r\}} \qquad = by\ SeqRec$$

$$K_1 := begin_1(x, ip') \,.\, \nu r : \overline{x}\langle r\rangle \,.\, \tau \,.\, r(a) \,.\, [\![\overline{ip'}\langle a\rangle \,;\, \overline{end_1}\langle\rangle \,;\, K_1]\!]_{\{ip', a\}} \qquad = by\ SeqSend$$

$$K_1 := begin_1(x, ip') \,.\, \nu r : \overline{x}\langle r\rangle \,.\, \tau \,.\, r(a) \,.\, \overline{ip'}\langle a\rangle \,.\, [\![\overline{end_1}\langle\rangle \,;\, K_1]\!]_{\emptyset} \qquad = by\ SeqSend$$

$$K_1 := begin_1(x, ip') \,.\, \nu r : \overline{x}\langle r\rangle \,.\, \tau \,.\, r(a) \,.\, \overline{ip'}\langle a\rangle \,.\, \overline{end_1}\langle\rangle \,.\, [\![K_1]\!]_{\emptyset} \qquad = by\ Call$$

$$K_1 := begin_1(x, ip') \,.\, \nu r : \overline{x}\langle r\rangle \,.\, \tau \,.\, r(a) \,.\, \overline{ip'}\langle a\rangle \,.\, \overline{end_1}\langle\rangle \,.\, K_1$$

$$K_2 := begin_2(x) \,.\, [\![x(v) \,;\, (\tau \;+\; \tau) \,;\, \overline{v}\langle a'\rangle \,;\, \overline{end_2}\langle\rangle \,;\, K_2]\!]_{\{x\}} \qquad = by\ SeqRec$$

$$K_2 := begin_2(x) \,.\, x(v) \,.\, [\![(\tau \;+\; \tau) \,;\, \overline{v}\langle a'\rangle \,;\, \overline{end_2}\langle\rangle \,;\, K_2]\!]_{\{v\}} \qquad = by\ SeqChoice$$

$$K_2 := begin_2(x) \,.\, x(v) \,.\, [\![(\tau \,;\, K_3\lfloor v\rfloor \;+\; \tau \,;\, K_3\lfloor v\rfloor)]\!]_{\{v\}} \qquad = by\ Choice$$

Here K_3 is a fresh PID (not added to the initial process).

$$K_2 := begin_2(x) \;.\; x(v) \;.\; ([\![\tau \,;\, K_3\lfloor v\rfloor]\!]_{\{v\}} \;+\; [\![\tau \,;\, K_3\lfloor v\rfloor]\!]_{\{v\}}) \qquad = by\ SeqTau$$

$$K_2 := begin_2(x) \;.\; x(v) \;.\; (\tau \;.\; [\![K_3\lfloor v\rfloor]\!]_{\{v\}} \;+\; \tau \;.\; [\![K_3\lfloor v\rfloor]\!]_{\{v\}}) \qquad = by\ Call$$

$$K_2 := begin_2(x) \;.\; x(v) \;.\; (\tau \;.\; K_3\lfloor v\rfloor \;+\; \tau \;.\; K_3\lfloor v\rfloor)$$

$$K_3[v] := [\![\overline{v}\langle a'\rangle \,;\, \overline{end_2}\langle\rangle \,;\, K_2]\!]_{\{v\}} \qquad = by\ SeqSend$$

$$K_3[v] := \overline{v}\langle a'\rangle \;.\; [\![\overline{end_2}\langle\rangle \,;\, K_2]\!]_{\emptyset} \qquad = by\ SeqSend$$

$$K_3[v] := \overline{v}\langle a'\rangle \;.\; \overline{end_2}\langle\rangle \;.\; [\![K_2]\!]_{\emptyset} \qquad = by\ Call$$

$$K_3[v] := \overline{v}\langle a'\rangle \;.\; \overline{end_2}\langle\rangle \;.\; K_2$$

Finally, the resulting FCP is:

$$C[url, ip] := \nu q : \overline{url}\langle ip, q\rangle \;.\; ip(a) \;.\; C\lfloor url, ip\rfloor$$

$$S[url'] := url'(ip', q') \;.\; \nu x : \overline{begin_1}\langle x, ip'\rangle \;.\; \overline{begin_2}\langle x\rangle \;.\; end_1() \;.\; end_2() \;.\; S\lfloor url'\rfloor$$

$$K_1 := begin_1(x, ip') \;.\; \nu r : \overline{x}\langle r\rangle \;.\; \tau \;.\; r(a) \;.\; \overline{ip'}\langle a\rangle \;.\; \overline{end_1}\langle\rangle \;.\; K_1$$

$$K_2 := begin_2(x) \;.\; x(v) \;.\; (\tau \;.\; K_3\lfloor v\rfloor \;+\; \tau \;.\; K_3\lfloor v\rfloor)$$

$$K_3[v] := \overline{v}\langle a' \rangle \ . \ \overline{end_2}\langle\rangle \ . \ K_2$$

$$\nu url'', ip'' : (S\lfloor url'' \rfloor \mid C\lfloor url'', ip'' \rfloor \mid K_1 \mid K_2)$$

4.4 Size of the translation

One can easily check that every translation rule except (SeqChoice) yields a linear size result, and that (SeqChoice) yields at most quadratic result. This quadratic blow-up happens when it is necessary to pass a large number of bound names as parameters of a call, as shown in the following example.

$$K := a(\tilde{x}) \, ; \left(\sum_{i=1}^{N} \tau \right) ; \overline{b}\langle \tilde{x} \rangle$$
$$K$$

The translated process is:

$$K := a(\tilde{x}). \left(\sum_{i=1}^{N} \tau . K_1\lfloor \tilde{x} \rfloor \right)$$
$$K_1[\tilde{x}] := \overline{b}\langle \tilde{x} \rangle \ . \ \mathbf{0}$$
$$K$$

If $|\tilde{x}| = N$ then the size of the translated specification is quadratic, as N calls with N parameters each are created.

Note that this quadratic blow-up in (SeqChoice) is isolated, and the subsequent translation of these calls by the (Call) rule cannot create any further blow-up, and so the overall size of the translated process is at most quadratic. Furthermore, one needs a rather artificial process for this quadratic blow-up to occur, and we conjecture that for practical EFCP models the translation will usually be linear.

5 Case Study

In this section, the applicability of the proposed formalism and its translation to safe FCP is demonstrated using SpiNNaker [9] as a case study.

5.1 SpiNNaker architecture

SpiNNaker is a massively parallel architecture designed to model large-scale spiking neural networks in real-time [10]. Its design is based around *ad-hoc* multi-core System-on-Chips, which are interconnected using a two-dimensional toroidal triangular mesh [10,11]. Neurons are modelled in software and their spikes generate packets that propagate through the on- and inter-chip communication fabric relying on custom-made on-chip multicast routers [12, 13]. The aim of SpiNNaker

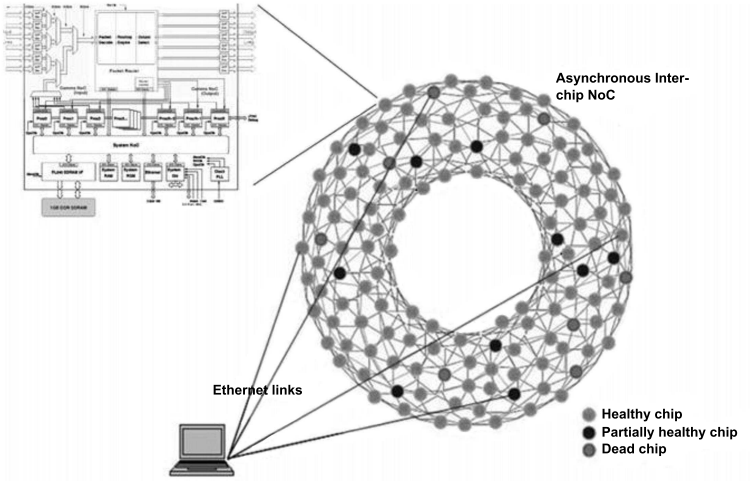

Fig. 1. The SpiNNaker architecture [17].

project is to simulate a billion spiking neurons in real time [14–16]. The SpiNNaker architecture is illustrated in Figure 1.

Every node of the network consists of a SpiNNaker Chip multiprocessor (CMP), which constitutes the basis of the system [18, 19]. It comprises 20 processing cores and SDRAM memory. For the cores, synchronous ARM9 processors were used because of their high power efficiency [14]. One of the processors is called monitor processor and its role is to perform system management tasks and to allow the user to track the on-chip activity. The other processors run independent event-driven neural processes and each of them simulates a group of neurons. Each processor core models up to around one thousand individual neurons.

The communication network-on-chip (NoC) provides an on- and off- chip packet switching infrastructure [20], see Figure 2. Its main task is to carry neural-event packets between the processors that can be located on the same or different chips. Also, it transports system configurations and monitoring information [18, 20]. The receiver of the data must be able to manage how long the sender keeps the data stable in order to complete a Delay-Insensitive communication. This is achieved by handshaking. The receiver uses an acknowledgement to show that data has been accepted. The acknowledgement follows a return-to-zero protocol [18, 20].

Figure 3 illustrates a SpiNNaker system composed of 25 SpiNNaker chips at a high level of abstraction. They are linked with each other by channels (e.g., *c1*, *c2*, ...). According to the routing protocol [20] of SpiNNaker's system, every chip can generate and propagate a datum. Every chip is connected to six other chips by bidirectional links as shown in Figure 3. This structure forms a

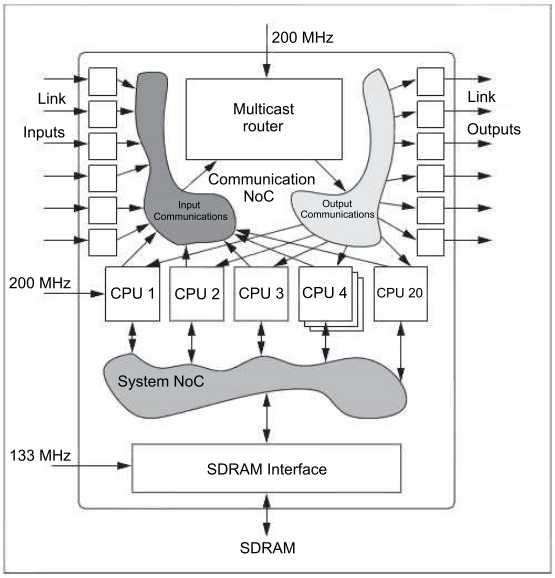

Fig. 2. The SpiNNaker chip organisation [20].

Cartesian coordinate system. For instance, P_0 can communicate only with P_1, P_6, P_5, P_4, P_{24} and P_{20}. Thus, the communication happens in the first and third quadrant. Every chip has a pair of coordinates. These coordinates are needed for the routing plan of the system. It is possible for some chips to be faulty or congested. In such a case, an emergency routing plan is followed to bypass this kind of issues [21, 22]. Thus, the redundancy of the SpiNNaker chips enhances the fault tolerance of the system [23].

5.2 Modelling SpiNNaker Interconnection Network

The flow-control mechanism of the interconnection network (IN) of SpiNNaker is as follows. When a packet arrives to an input port, one or more output ports are selected, and the router tries to transmit the packet through them. If the packet cannot be forwarded, the router will keep trying, and after a given period of time it will also test the clockwise emergency route. It will try both the regular and the emergency route. Finally, if a packet stays in the router for longer than a given threshold (waiting time), the packet will be dropped to avoid deadlocks. To avoid livelocks, packets have an age field in their header. When two ages pass and the packet is still in the IN, it is considered outdated and dropped [10].

The following EFCP models a 5×5 SpiNNaker configuration. A healthy processor, *HP*, can execute either of the following scenarios:

– It can generate a new message, m, and process it by calling an auxiliary declaration *MSEND*.

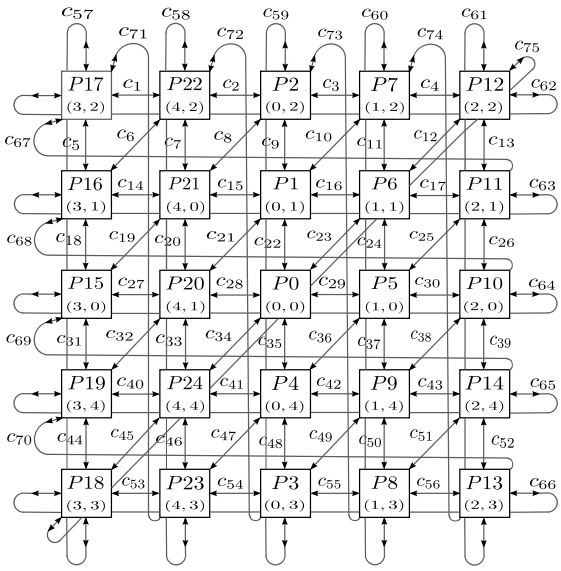

Fig. 3. SpiNNaker network topology [20].

– It can receive a message on any of its channels and process it using an auxiliary declaration *REC_MSEND*.
– It can become permanently faulty by calling an auxiliary declaration *FP*.

The definition of *HP* has six formal parameters corresponding to the six channels connecting it to the neighbours, see Fig. 3. These parameters are named after points of the compass, e.g. 'n' stands for 'north', 'ne' stands for 'north-east', etc.

$$\begin{aligned} HP[n, ne, e, s, sw, w] := \nu m : MSEND\lfloor m, n, ne, e, s, sw, w\rfloor \; + \\ REC\_MSEND\lfloor n, n, ne, e, s, sw, w\rfloor \; + \\ REC\_MSEND\lfloor ne, n, ne, e, s, sw, w\rfloor \; + \\ REC\_MSEND\lfloor e, n, ne, e, s, sw, w\rfloor \; + \\ REC\_MSEND\lfloor s, n, ne, e, s, sw, w\rfloor \; + \\ REC\_MSEND\lfloor sw, n, ne, e, s, sw, w\rfloor \; + \\ REC\_MSEND\lfloor w, n, ne, e, s, sw, w\rfloor \; + \\ FP\lfloor n, ne, e, s, sw, w\rfloor \end{aligned}$$

The auxiliary declarations are as follows:

$MSEND[m, n, ne, e, s, sw, w]$ sends message m on 0 or more of the channels and becomes $HP[n, ne, e, s, sw, w]$. In particular, the message can be consumed, forwarded or multicast. Clockwise emergency routes are used in case of negative acknowledgement *nack*.

$$
\begin{aligned}
MSEND[m,n,ne,e,s,sw,w] := \Big(& \\
& \Big(\tau \ + \ \overline{n}\langle m\rangle \,;\, n(a) \,;\, \big([a = ack] \,;\, \tau \ + \ [a = nack] \,;\, \overline{ne}\langle m\rangle \,;\, ne(a)\big)\Big) \mid \\
& \Big(\tau \ + \ \overline{ne}\langle m\rangle \,;\, ne(a) \,;\, \big([a = ack] \,;\, \tau \ + \ [a = nack] \,;\, \overline{e}\langle m\rangle \,;\, e(a)\big)\Big) \mid \\
& \Big(\tau \ + \ \overline{e}\langle m\rangle \,;\, e(a) \,;\, \big([a = ack] \,;\, \tau \ + \ [a = nack] \,;\, \overline{s}\langle m\rangle \,;\, s(a)\big)\Big) \mid \\
& \Big(\tau \ + \ \overline{s}\langle m\rangle \,;\, s(a) \,;\, \big([a = ack] \,;\, \tau \ + \ [a = nack] \,;\, \overline{sw}\langle m\rangle \,;\, sw(a)\big)\Big) \mid \\
& \Big(\tau \ + \ \overline{sw}\langle m\rangle \,;\, sw(a) \,;\, \big([a = ack] \,;\, \tau \ + \ [a = nack] \,;\, \overline{w}\langle m\rangle \,;\, w(a)\big)\Big) \mid \\
& \Big(\tau \ + \ \overline{w}\langle m\rangle \,;\, w(a) \,;\, \big([a = ack] \,;\, \tau \ + \ [a = nack] \,;\, \overline{n}\langle m\rangle \,;\, n(a)\big)\Big) \\
\Big) \,;\, & HP\lfloor n, ne, e, s, sw, w\rfloor
\end{aligned}
$$

$REC\_MSEND[c, n, ne, e, s, sw, w]$ receives a message on channel c and either negatively acknowledges (*nack*) it to simulate congestion or positively acknowledges (*ack*) it and then consumes, forwards or multicasts it by calling *MSEND*.

$$
\begin{aligned}
& REC\_MSEND[c,n,ne,e,s,sw,w] := c(m) \,;\, (\\
& \quad \overline{c}\langle nack\rangle \,;\, HP\lfloor n, ne, e, s, sw, w\rfloor + \overline{c}\langle ack\rangle \,;\, MSEND\lfloor m, n, ne, e, s, sw, w\rfloor \\
&)
\end{aligned}
$$

$FP[n, ne, e, s, sw, w]$ models a faulty process that does not send any messages and negatively acknowledges (*nack*) all the received messages.

$$
\begin{aligned}
& FP[n,ne,e,s,sw,w] := (\\
& \quad n(m) \,;\, \overline{n}\langle nack\rangle + ne(m) \,;\, \overline{ne}\langle nack\rangle + e(m) \,;\, \overline{e}\langle nack\rangle + \\
& \quad s(m) \,;\, \overline{s}\langle nack\rangle + sw(m) \,;\, \overline{sw}\langle nack\rangle + w(m) \,;\, \overline{w}\langle nack\rangle \\
&) \,;\, FP\lfloor n, ne, e, s, sw, w\rfloor
\end{aligned}
$$

The initial term creates 25 concurrent instances of HP, $\prod_{i=1}^{25} HP\lfloor \ldots \rfloor$, and connects them by channels as shown in Figure 3:

$$HP\lfloor c_{22}, c_{23}, c_{29}, c_{35}, c_{34}, c_{28}\rfloor \ \mid \ HP\lfloor c_9, c_{10}, c_{16}, c_{22}, c_{21}, c_{15}\rfloor \ \mid$$

$$
\begin{aligned}
HP\lfloor c_{59}, c_{73}, c_3, c_9, c_8, c_2\rfloor \mid HP\lfloor c_{48}, c_{49}, c_{55}, c_{59}, c_{72}, c_{54}\rfloor \mid \\
HP\lfloor c_{35}, c_{36}, c_{42}, c_{48}, c_{47}, c_{41}\rfloor \mid HP\lfloor c_{24}, c_{25}, c_{30}, c_{37}, c_{36}, c_{29}\rfloor \mid \\
HP\lfloor c_{11}, c_{12}, c_{17}, c_{24}, c_{23}, c_{16}\rfloor \mid HP\lfloor c_{60}, c_{74}, c_4, c_{11}, c_{10}, c_3\rfloor \mid \\
HP\lfloor c_{50}, c_{51}, c_{56}, c_{60}, c_{73}, c_{55}\rfloor \mid HP\lfloor c_{37}, c_{38}, c_{43}, c_{50}, c_{49}, c_{42}\rfloor \mid \\
HP\lfloor c_{26}, c_{68}, c_{64}, c_{39}, c_{38}, c_{30}\rfloor \mid HP\lfloor c_{13}, c_{67}, c_{63}, c_{26}, c_{25}, c_{17}\rfloor \mid \\
HP\lfloor c_{61}, c_{75}, c_{62}, c_{13}, c_{12}, c_4\rfloor \mid HP\lfloor c_{52}, c_{70}, c_{66}, c_{61}, c_{74}, c_{56}\rfloor \mid \\
HP\lfloor c_{39}, c_{69}, c_{65}, c_{52}, c_{51}, c_{43}\rfloor \mid HP\lfloor c_{18}, c_{19}, c_{27}, c_{31}, c_{69}, c_{64}\rfloor \mid \\
HP\lfloor c_5, c_6, c_{14}, c_{18}, c_{68}, c_{63}\rfloor \mid HP\lfloor c_{57}, c_{71}, c_1, c_5, c_{67}, c_{62}\rfloor \mid \\
HP\lfloor c_{44}, c_{45}, c_{53}, c_{57}, c_{75}, c_{66}\rfloor \mid HP\lfloor c_{31}, c_{32}, c_{40}, c_{44}, c_{70}, c_{65}\rfloor \mid \\
HP\lfloor c_{20}, c_{21}, c_{28}, c_{33}, c_{32}, c_{27}\rfloor \mid HP\lfloor c_7, c_8, c_{15}, c_{20}, c_{19}, c_{14}\rfloor \mid \\
HP\lfloor c_{58}, c_{72}, c_2, c_7, c_6, c_1\rfloor \mid HP\lfloor c_{46}, c_{47}, c_{54}, c_{58}, c_{71}, c_{53}\rfloor \mid \\
HP\lfloor c_{33}, c_{34}, c_{41}, c_{46}, c_{45}, c_{40}\rfloor
\end{aligned}
$$

The above specification is an EFCP and below its translation to FCP is given. It has been obtained with the help of the developed tool Efcp2Fcp. First of all, this EFCP must be translated to a safe EFCP. This is done automatically by the tool by replicating the process declarations, HP, $MSEND$, REC_MSEND, and FP, so that each of the 25 threads has its own copies of these declarations: HP^i, $MSEND^i$, REC_MSEND^i, FP^i, $i = 1 \ldots 25$. Also, the tool enforces the **(NOCLASH)** assumptions by renaming the formal parameters and bound names. However, below we disregard this renaming for the sake of clarity.

The translation of HP and REC_MSEND is straightforward as they do not use any special features of EFCP:

$$
\begin{aligned}
HP^i[n, ne, e, s, sw, w] := \nu m : MSEND^i\lfloor m, n, ne, e, s, sw, w\rfloor \ + \\
REC\_MSEND^i\lfloor n, n, ne, e, s, sw, w\rfloor \ + \\
REC\_MSEND^i\lfloor ne, n, ne, e, s, sw, w\rfloor \ + \\
REC\_MSEND^i\lfloor e, n, ne, e, s, sw, w\rfloor \ + \\
REC\_MSEND^i\lfloor s, n, ne, e, s, sw, w\rfloor \ + \\
REC\_MSEND^i\lfloor sw, n, ne, e, s, sw, w\rfloor \ + \\
REC\_MSEND^i\lfloor w, n, ne, e, s, sw, w\rfloor \ + \\
FP^i\lfloor n, ne, e, s, sw, w\rfloor
\end{aligned}
$$

$$REC\_MSEND^i[c,n,ne,e,s,sw,w] := c(m)\ .\ (\ \overline{c}\langle nack\rangle\ .\ HP^i\lfloor n,ne,e,s,sw,w\rfloor + \overline{c}\langle ack\rangle\ .\ MSEND^i\lfloor m,n,ne,e,s,sw,w\rfloor\ )$$

The translation of FP can be obtained by applying the (SeqChoice) rule:

$$FP^i[n,ne,e,s,sw,w] := n(m).\overline{n}\langle nack\rangle.K_{FP^i}\lfloor n,ne,e,s,sw,w\rfloor + ne(m).\overline{ne}\langle nack\rangle.K_{FP^i}\lfloor n,ne,e,s,sw,w\rfloor + e(m).\overline{e}\langle nack\rangle.K_{FP^i}\lfloor n,ne,e,s,sw,w\rfloor + s(m).\overline{s}\langle nack\rangle.K_{FP^i}\lfloor n,ne,e,s,sw,w\rfloor + sw(m).\overline{sw}\langle nack\rangle.K_{FP^i}\lfloor n,ne,e,s,sw,w\rfloor + w(m).\overline{w}\langle nack\rangle.K_{FP^i}\lfloor n,ne,e,s,sw,w\rfloor$$

$$K_{FP^i}[n,ne,e,s,sw,w] := FP^i\lfloor n,ne,e,s,sw,w\rfloor$$

Here K_{FP^i} is a fresh PID.

The translation of $MSEND$ is the most interesting one as it contains some local concurrency that is not allowed in FCPs (below K^i_j, L^i_j and M^i_j are fresh PIDs and $begin^i_j$ and end^i_j are fresh public names):

$$MSEND^i[m,n,ne,e,s,sw,w] := \overline{begin^i_1}\langle m,n,ne\rangle.\overline{begin^i_2}\langle m,ne,e\rangle.\overline{begin^i_3}\langle m,e,s\rangle.\overline{begin^i_4}\langle m,s,sw\rangle.\overline{begin^i_5}\langle m,sw,w\rangle.\overline{begin^i_6}\langle m,w,n\rangle.end^i_1().end^i_2().end^i_3().end^i_4().end^i_5().end^i_6().HP^i\lfloor n,ne,e,s,sw,w\rfloor$$

$$K^i_1 := begin^i_1(m,n,ne).\big(\ \tau.L^i_1 + \overline{n}\langle m\rangle.n(a).([a=ack].\tau.M^i_1 + [a=nack].\overline{ne}\langle m\rangle.ne(a).M^i_1)\ \big)$$

$$L^i_1 := \overline{end^i_1}\langle\rangle\ .\ K^i_1$$

$$M^i_1 := L^i_1$$

$$K^i_2 := begin^i_2(m,ne,e).\big(\ \tau.L^i_2 + \overline{ne}\langle m\rangle.ne(a).([a=ack].\tau.M^i_2 + [a=nack].\overline{e}\langle m\rangle.e(a).M^i_2)\ \big)$$

$$L^i_2 := \overline{end^i_2}\langle\rangle\ .\ K^i_2$$

$$M^i_2 := L^i_2$$

$$K^i_3 := begin^i_3(m,e,s).\big($$

$$\tau.L_3^i + \overline{e}\langle m\rangle.e(a).([a = ack].\tau.M_3^i + [a = nack].\overline{s}\langle m\rangle.s(a).M_3^i)$$
$$)$$

$$L_3^i := \overline{end_3^i}\langle\rangle \;.\; K_3^i$$

$$M_3^i := L_3^i$$

$$K_4^i := begin_4^i(m, s, sw).($$
$$\tau.L_4^i + \overline{s}\langle m\rangle.s(a).([a = ack].\tau.M_4^i + [a = nack].\overline{sw}\langle m\rangle.sw(a).M_4^i)$$
$$)$$

$$L_4^i := \overline{end_4^i}\langle\rangle \;.\; K_4^i$$

$$M_4^i := L_4^i$$

$$K_5^i := begin_5^i(m, sw, w).($$
$$\tau.L_5^i + \overline{sw}\langle m\rangle.sw(a).([a = ack].\tau.M_5^i + [a = nack].\overline{w}\langle m\rangle.w(a).M_5^i)$$
$$)$$

$$L_5^i := \overline{end_5^i}\langle\rangle \;.\; K_5^i$$

$$M_5^i := L_5^i$$

$$K_6^i := begin_6^i(m, w, n).($$
$$\tau.L_6^i + \overline{w}\langle m\rangle.w(a).([a = ack].\tau.M_6^i + [a = nack].\overline{n}\langle m\rangle.n(a).M_6^i)$$
$$)$$

$$L_6^i := \overline{end_6^i}\langle\rangle \;.\; K_6^i$$

$$M_6^i := L_6^i$$

The initial process is now as follows:

$$\prod_{i=1}^{25} HP^i\lfloor\ldots\rfloor \quad | \quad \prod_{i=1}^{25}\prod_{j=1}^{6} K_j^i$$

5.3 Formal verification

As outlined in the introduction, formal verification is an important motivation of this paper. It was performed as follows. First, the EFCP model of the 2x2 SpiNNaker network was automatically translated into an FCP model by the EFCP2FCP tool. Then the resulting FCP was then translated into a safe low-level Petri net using the FCP2PN tool [5]. Some small adaptations had to be done for the latter: FCP2PN requires choices to be *guarded,* i.e. each summand must

start with a prefix, match or mismatch. This was achieved by inlining the calls to REC_MSEND^i and prefixing the first and last summands in the body of HP^i with τ. We also inlined the calls to L^i_j and M^i_j as an optimisation – the same effect could have been achieved automatically during the translation if rule (SeqChoice) were avoiding the creation of a new PID whenever the size of P does not exceed some pre-defined constant. The translation runtimes were negligible (<2sec) in both cases, and the resulting Petri net contained 14844 places, 38864 transitions and 292336 arcs.

Then deadlock checking was performed with the LoLA tool,[2] configured to assume safeness of the Petri net (`CAPACITY 1`), use the stubborn sets and symmetry reductions (`STUBBORN`, `SYMMETRY`), compress states using P-invariants (`PREDUCTION`), use a light-weight data structure for states (`SMALLSTATE`), and check for deadlocks (`DEADLOCK`).

The verification runtime was 3223sec, and LoLA reported that the model had a deadlock. In hindsight, this is quite obvious, as the model allows all the processors to become faulty, after which they stop generating new messages and the system quickly reaches a deadlock state.

6 Conclusion

The initial motivation of this research was the development of a formalism allowing for convenient modelling and formal verification of Reference Passing Systems. To that end, a new fragment of π-calculus, the Extended Finite Control Processes, is presented in this paper. EFCPs is an extension of the well-known fragment of π-calculus, the Finite Control Processes. FCPs were used for formal modelling of reference passing systems; however, they cannot express scenarios involving 'local' concurrency inside a process. EFCPs remove this limitation. As a result, practical modelling of mobile systems becomes more convenient, e.g. multicast can be naturally expressed. To this end, also a more powerful sequential composition operator ' ; ' is used instead of prefixing. The SpiNNaker case study demonstrates that EFCPs allow for a concise expression of multicast communication, and is suitable for practical modelling. Furthermore, an almost linear translation from safe EFCP to safe FCP has been developed, which forms the basis of formal verification of RPSs.

In our future work we intend to investigate the relationship between the transition systems generated by EFCPs and those generated by the corresponding FCPs, with the view to prove the correctness of the proposed translation. We would also like to evaluate the scalability of the proposed approach on a range of models and optimise the translation, e.g. by reducing the number of generated defining equations and by lifting it to non-safe processes.

[2] Available from http://service-technology.org/tools/lola.

References

1. Kwiatkowska, M.Z., Rodden, T., (Editors), V.S.: From computers to ubiquitous computing by 2020. In: Proc. of Philosophical Transactions of the Royal Society. Volume 366. (2008)
2. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, part I. Inf. Comp. **100** (1992) 1–40
3. Cardelli, L., Gordon, A.: Mobile ambients. In Nivat, M., ed.: Foundations of Software Science and Computation Structures. Volume 1378 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (1998) 140–155
4. Dam, M.: Model checking mobile processes. Inf. Comp. **129** (1996) 35–51
5. Meyer, R., Khomenko, V., Hüchting, R.: A polynomial translation of π-calculus (FCP) to safe Petri nets. Logical Methods in Computer Science **9** (2013) 1–36
6. Milner, R.: Communicating and Mobile Systems: the π-Calculus. CUP (1999)
7. Sangiorgi, D., Walker, D.: The π-calculus: A Theory of Mobile Processes. CUP (2001)
8. Meyer, R., Khomenko, V., Strazny, T.: A practical approach to verification of mobile systems using net unfoldings. Fundam. Inf. **94** (2009) 439–471
9. Furber, S., Temple, S.: Neural systems engineering. In Fulcher, J., Jain, L., eds.: Computational Intelligence: A Compendium. Volume 115 of Studies in Computational Intelligence. Springer Berlin Heidelberg (2008) 763–796
10. Navaridas, J., Luján, M., Miguel-Alonso, J., A.Plana, L., Furber, S.: Understanding the interconnection network of SpiNNaker. In: Proceedings of the 23rd International Conference on Supercomputing. ICS '09, ACM (2009) 286–295
11. Camara, J., Moreto, M., Vallejo, E., Beivide, R., Miguel-Alonso, J., Martinez, C., J.Navaridas: Mixed-radix twisted torus interconnection networks. In: Parallel and Distributed Processing Symposium IPDPS, IEEE (2007) 1–10
12. Plana, L., Bainbridge, J., Furber, S., Salisbury, S., Yebin, S., Jian, W.: An on-chip and inter-chip communications network for the SpiNNaker massively-parallel neural net simulator. In: Networks-on-Chip, 2008. NoCS 2008. Second ACM IEEE International Symposium on, IEEE Computer Society (2008) 215–216
13. Furber, S., Temple, S., Brown, A.: On-chip and inter-chip networks for modeling large-scale neural systems. In: Circuits and Systems. ISCAS Proceedings, IEEE (2006) 21–24
14. Rast, A., Yang, S., Khan, M., Furber, S.: Virtual synaptic interconnect using an asynchronous network-on-chip. In: Proceedings of Intelligence Joint Conference on Neural Networks (IJCNN2008), IEEE (2008) 2727–2734
15. Jin, X., Furber, S., Woods, J.: Efficient modelling of spiking neural networks on a scalable chip multiprocessor. In: Neural Networks, IJCNN World Congress on Computational Intelligence, IEEE (2008) 2812–2819
16. Asanovic, K., Beck, J., Feldman, J., Morgan, N., Wawrzynek, J.: A supercomputer for neural computation. In: Neural Networks, World Congress on Computational Intelligence. Volume 1., IEEE (1994) 5–9
17. Furber, S., Brown, A.: Biologically inspired massively parallel architectures computing beyond a million processors. In: Application of Concurrency to System Design, 2009. ACSD'09, IEEE (2009) 3–12
18. Plana, L., Furber, S., Temple, S., Khan, M., Shi, Y., Wu, J., Yang, S.: A GALS infrastructure for a massively parallel multiprocessor. Design Test of Computers, IEEE **24** (2007) 454–463

19. Farber, P., Asanovic, K.: Parallel neural network training on multi-spert. In: Algorithms and Architectures for Parallel Processing, ICAPP, 3rd International Conference on, IEEE (1997) 659–666
20. Wu, J., Furber, S.: A multicast routing scheme for a universal spiking neural network architecture. Comput. J. **53** (2010) 280–288
21. Puente, V., Gregorio, J.: Immucube: scalable fault-tolerant routing for k-ary n-cube networks. Parallel and Distributed Systems, IEEE Transactions on **18** (2007) 776–788
22. Puente, V., Izu, C., Beivide, R., Gregorio, J.A., Vallejo, F., M.Prellezo, J.: The adaptive bubble router. J. Parallel Distrib. Comput. **61** (2001) 1180–1208
23. Gomez, M., Nordbotten, N., Flich, J., Lopez, P., Robles, A., Duato, J., Skeie, T., Lysne, O.: A routing methodology for achieving fault tolerance in direct networks. Computers, IEEE Transactions on **55** (2006) 400–415