THE ϕ-CALCULUS – A HYBRID EXTENSION OF THE π-CALCULUS TO EMBEDDED SYSTEMS

WILLIAM C. ROUNDS AND HOSUNG SONG

1. Introduction

Embedded systems are software systems which reside in a physical environment and must react with that environment in real time. The problem of designing such systems in a principled way is one which has become increasingly important as computational technology has been integrated into physical systems.

In this paper we begin to address this problem by drawing on research traditions from computer science and control theory for the long-range purpose of making a mathematical model of embedded software systems. The theory of *process algebras* has been extensively developed within computer science, to the point where control of distributed processes is well-understood, even to the point where these processes are *mobile* (reconfigurable). Similarly, there is now a large body of work on the topic of *hybrid systems*: automaton-based techniques for specifying discrete control of continuous processes. Very recently, this work has been extended to proposed programming languages for specifying distributed control of hybrid systems [1, 10]. Our work is in this latter direction: we propose a simple method for integrating process-algebraic techniques with hybrid automaton-based techniques so that one can use the integrated theory to specify the semantics of real programming languages for *concurrent* and *reconfigurable* embedded systems.

The ϕ-calculus is a hybrid extension of Milner's π*-calculus* [17] which allows processes to interact with continuous environments. We choose the π-calculus to extend to the hybrid setting because it has already been shown to be a rich language in which many interesting discrete concurrent phenomena can be expressed: a language for, and theory of, communicating processes which can *reconfigure* themselves; a language in which distributed objects and classes can be defined; and a language and theory capable not only of expressing communication, but arbitrary computation, in that the λ-calculus of Church can be translated into it. This all suggests that successful hybrid versions of the π-calculus and other process calculi will have novel and elegant ways of expressing hybrid systems – possibilities for distributed control which would be awkward, if not impossible, to express in current formalisms.

The key idea in the present paper involves adding an explicit formal model of the *active* continuous (physical) environment to an algebraic process description, and adding "environmental actions (e-actions)" to processes which allow them to explicitly manipulate their environment. We define an *embedded (hybrid) system* as a pair (E, P), where E is an environment and P is a hybrid process expression (ϕ-expression). An embedded system evolves (i) by means of π-actions, which change only the process expression, (ii) by flow actions, which change only the environment continuously, or (iii) by e-actions, which change both the environment and a process expression discretely.

The power of this idea can be seen by noting that Petri nets, in most of their flavors, are but simple special cases of embedded systems, where (in the standard flavor) a process is a parallel composition of net transitions, and an environment is a marking of the places; firing a transition is an e-action. This example suggests that environments can be of arbitrary types, each possessing interesting ways of interacting with concurrent algebraically specified systems of processes. Mathematically speaking, basic actions of a hybrid process are (possibly nondeterministic) actions of two monoids on a cartesian product of sets, with constraints restricting arbitrarily free combinations of actions. In the ϕ-calculus, the two monoids are the monoid consisting of real time segments, constrained (as in hybrid automata) by differential equations and flow regions, and the monoid of strings of ϕ and π-action names, constrained by process expressions using structural operational semantics. We have extended this latter method to specify both time-transitions and discrete actions of an embedded system. In this sense, we already have a compositional semantics for our system, because structural operational semantics defines the transitions of any system in terms of the transitions of its components.

The specific contributions of the paper are the following.

- The definition of embedded systems;
- Definitions of the behavior of mobile hybrid processes in a continuous environment, via structural operational semantics;
- New definitions of strong and weak bisimulation for ϕ-processes, with congruence lemmas;
- A method for declaring local environmental variables;
- Obtaining recursive definition capabilities by hybridizing Milner's replication operator;
- Translation of Klavins and Koditschek's *threaded Petri nets* [13] into the ϕ-calculus;
- Illustrating the mobile ϕ-calculus in action, using Klavins and Koditschek's *robotic bucket brigade*, with the addition of a recursive process supplying a continuous stream of items to be transported by the brigade.

2. Plan of paper

We begin by reviewing previous work on hybrid systems, and some work on process-algebraic systems with timed behavior. Most of this review is concentrated on hybrid systems work which is focused towards integration with programming. We ignore much work on the control theory side of hybrid systems having more to do with differential equations, and we also ignore much work on programming languages with real-time behavior.

We then review π-calculus (a simple version π_c based on Milner's CCS), and then introduce the corresponding simple version of our hybrid extension ϕ_c. We indicate a general form, informally, for e-actions, but then propose a concrete syntax based on a standard presentation of differential equations. This allows us to describe exactly the effects of e-actions on the environment, as well as to specify continuous time-transitions for the environment. We illustrate both versions with Hoare's classic example of a vending machine dispensing either coffee or tea, providing a hybrid version of this machine which accepts one-second requests for coffee only and then for tea. We also briefly indicate that Petri nets are an example of the general formalism.

A short section is then devoted to defining the notion of strong bisimulation for the ϕ_c-calculus, showing in what sense one obtains congruences with this definition, and indicating that a revision of the continuous time-transition rules is necessary in order to deal with weak bisimulation.

In the next section we introduce name-passing and recursion into our calculus, with associated operational semantics for the ϕ-calculus. We introduce an abstraction operator for environmental actions, so that variables can be hidden in modular components. Then we present another section on strong and weak bisimulation in the full ϕ-calculus, revising the flow transition rules in accordance with the notions of atomic experiments.

In the penultimate section we show by example how to translate Klavins and Koditschek's threaded Petri nets into the ϕ-calculus, using a ϕ-calculus version of their robotic bucket brigade.

This example shows mobility and recursion to good advantage. General details of the translation appear in an appendix.

A final section gives conclusions and directions for more research.

3. Other work

There is already a body of research on the process-algebraic treatment of (some) hybrid phenomena. Timed CSP [19], for example, is a well-known language in which an algebraic structure theory of timed processes is presented, and made into a specification-oriented programming language allowing various parallel and sequential compositions of processes, together with primitive timeout and interrupt operators on processes for altering real-time behavior. One can view this language as incorporating timed automata into a process-algebraic framework, with the significant addition of recursive definition capabilities. Moreover, the language has a fully compositional semantics based on the extension of "failures" [3] to the timed case. Unfortunately, it is not clear how to extend the expressiveness of this language to the hybrid case, where the continuous behavior of a system can follow control laws prescribed by arbitrary ordinary differential equations, not just (implicit) equations describing the behavior of clocks.

Kosecka's thesis [14] contains a proposal for structured programming which starts by assuming a collection of basic low-level robot controllers (as, for example in Khatib [12] or Rimon and Koditschek [18].) These controllers are composed into expressions using various automata-theoretic operations. This idea is essentially what we have in mind for composing processes in our setting, but crucially, the behavior of the continuous environment is not really represented.

Ordinary hybrid automata as in Henzinger [9], on the other hand, explicitly represent the continuous environment by means of associating a controller (differential equation, differential inclusion) with each state (control mode) of a finite automaton. Events trigger jumps from one control mode to another. There is, moreover, a general "parallel composition" operator for combining these automata, but no syntax is presented for a language which would allow program expressions.

Another approach to timed and hybrid distributed system design is via the *I-O automata* of Lynch and Tuttle [16]. The original work has been extended in many ways, notably to a process-algebraic version [20] and to the hybrid case [15]. Moreover, I-O automata have been used as the formal foundation for a distributed programming language [6]; this includes an extension to timed, but not hybrid versions. In this research vein, careful attention is paid to formal semantics, using a combination of techniques, including simulation notions. The achievements of this work are impressive, and represent in many ways the goals we would like to achieve for our work. There are, on the other hand, some possible expressive advantages and some possible simplifications with our approach. We plan to investigate the exact relationships between the two models as part of our further research.

Two languages which support both hybrid dynamics and algebraic process structuring are CHARON [1] and Masaccio [10]. These languages are both outgrowths of earlier models for reactive discrete systems [2]. At this point they both lack a full name-passing capability, and a capacity for recursive instantiation, though they do support abstraction (variable hiding). There has been a proposal as well for hybrid CSP [11], but the exact details of the semantics are a little unclear.

A very different approach to the problem of integrating a full programming language with continuous dynamical systems is taken by the language HybridCC [8]. This is a constraint programming language which extends earlier languages in the cc paradigm to the continuous setting. There are many similarities in this model to the ϕ-calculus, in that continuous constraints can be "posted" to the store by parallel agents. These constraints include both differential equations and "invariant regions", a feature of hybrid automata. The store of HybridCC is much like the environment in the ϕ-calculus. However, there is no explicit provision for mobile agents in this language, even though some mobile systems can be represented. (Intuitively, these are systems in which the mobility consists of rearranging groups of continuous variables.) A precise comparison is beyond the scope of this paper.

Klavins and Koditschek's work on threaded Petri nets [13] is another starting point for our research. This model of distributed hybrid systems uses the places of a Petri net to hold a set of continuous variables as tokens. This model has interesting applications to robotic assembly problems in factory settings. Moreover, it can be used for large-scale problems without state explosion, since the Petri net keeps interactions local. The model lacks, however, a process-algebraic syntax. Our translation of threaded nets into ϕ-calculus provides it.

Other Petri net models (timed Petri nets [7], hybrid Petri nets [5]) differ from the threaded Petri nets in that (say for the timed nets) specific delays are prescribed for the transitions, while (for hybrid nets) tokens are "counted by" either integers or real numbers. These models do not seem to be as flexible as the threaded nets; for example, it is easy to simulate a timed net by a threaded one.

4. π-calculus and ϕ-calculus

4.1. π_c.

We proceed as in Milner [17] by introducing a simple CCS fragment π_c, without name-passing, and also reviewing transition relations and the notion of structural congruence.

Definition 4.1. *The set of π_c expressions is given by the following syntax:*

$$P ::= A \mid \Sigma_{i\in I}\alpha_i.P_i \mid P_1 \parallel P_2 \mid \nu a P$$

where α ranges over a finite set $\mathcal{N}$ of positive actions a, negative actions $\overline{\mathcal{N}} = \{\overline{a} \mid a \in \mathcal{N}\}$ and the silent action *τ. The empty sum is denoted 0. We put $\mathcal{L} = \mathcal{N} \cup \overline{\mathcal{N}}$, and call elements of this set* action prefixes*.*

This syntax defines process expressions. The operator $\parallel$ represents parallel composition; summation represents a choice of actions followed by a process; ν is the operator used to restrict access to actions under its scope. We follow a simple (temporary) convention that every process identifier A has a defining equation $A ::= \Sigma_i \alpha_i A_i$, which has the effect of defining finite-state automata. The expression on the right here is called a *sum*.

Definition 4.2 (Structural congruence)**.** *The relation $\equiv$ of structural congruence is the congruence relation generated by the following equations:*

- *Change of bound names (alpha-conversion);*
- *Reordering of terms in a sum;*
- *$\parallel$ is commutative and associative; $P \parallel 0 \equiv P$;*
- *$\nu a(\ P \parallel Q\ ) \equiv\ P \parallel\ \nu a Q$ if a is not free in P; $\nu a 0 \equiv 0$; $\nu a b P \equiv \nu b a P$;*
- *$A \equiv P_A$ if $A ::= P_A$ (recall P_A is a sum.)*

Structural congruence reflects our intuitions about the basic operations; for example that parallel composition does not depend on which process is named first, and so on.

Definition 4.3 (Transition relations)**.** *Let α range over actions in $\mathcal{L} \cup \{\tau\}$, λ over $\mathcal{L}$. Consider the following expression inference rules:*

$$\textit{Sum: } \alpha.P + M \xrightarrow{\alpha} P \tag{1}$$

$$\textit{React: } \frac{P \xrightarrow{\lambda} P' \qquad Q \xrightarrow{\overline{\lambda}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'} \tag{2}$$

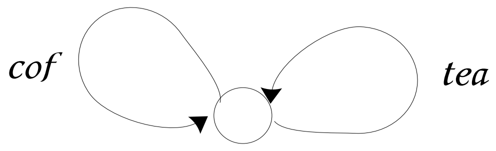
$$\textit{L-Par: } \frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q} \qquad \textit{R-Par: } \frac{Q \xrightarrow{\alpha} Q'}{P \mid Q \xrightarrow{\alpha} P \mid Q'} \tag{3}$$

$$\textit{Res: } \frac{P \xrightarrow{\alpha} P'}{\nu a P \xrightarrow{\alpha} \nu a P'} \textit{ if } \alpha \textit{ is not } a \textit{ or } \overline{a} \tag{4}$$

$$\textit{Ident: } \frac{P_A \xrightarrow{\alpha} P'}{A \xrightarrow{\alpha} P'} \textit{ if } A ::= P_A \tag{5}$$

We say that $P \xrightarrow{\alpha} Q$ if this relation can be inferred from the above inference rules.

Example 4.4 (The vending machine.)**.** We begin with a finite-state machine specified algebraically as in Milner's book. A vending machine is prepared to serve coffee or tea according to a button pressed by a user. After serving, it goes back to its prepared state. The picture of this is given by the simple FSA



Algebraically, we define

$$VM ::= cof.VM + tea.VM$$

One uses the Sum and the Ident laws to infer the actions of this machine. We have $cof.VM + tea.VM \xrightarrow{cof} VM$ by the Sum rule. Since this transition entails the hypothesis of the Ident rule, we have $VM \xrightarrow{cof} VM$. To illustrate reaction, we model a customer as someone who issues the "button" command $\overline{cof}$ to the VM. Their interaction is specified by the composition $VM \parallel \overline{cof}.0$. Since $VM \xrightarrow{cof} VM$, and $\overline{cof} \xrightarrow{\overline{cof}} 0$, we get

$$VM \parallel \overline{cof}.0 \ \xrightarrow{\tau}\ VM \parallel 0 \ \equiv VM$$

by the React rule. This amounts to a successful interaction of a (one-shot) customer with the machine.

A very different VM is one which can decide internally to accept or not requests for coffee or tea. This machine has nondeterministic internal transitions to states which can reject coffee requests for only tea requests, and reject tea requests for only coffee requests. Its specification would be

$$BADVM ::= \tau.cof.BADVM + \tau.tea.BADVM.$$

This machine can evolve using the τ "silent" transition to $tea.BADVM$, a state in which coffee requests are refused.

With the ϕ-calculus, we would like to define an intermediate creature: a machine which accepts coffee requests only, but for one second, after which it "times out" and accepts a tea request only for one second, after which it times out again to its original state. This example is trivial in Timed

CSP, where the "timeout after d seconds " operator is primitive; it is also trivial using ordinary hybrid automata. We will show how to specify it below in the ϕ-calculus.

4.2. ϕ_c.

Hybrid automata associate with each state a *flow law*, typically an ordinary differential equation over $\mathbb{R}^n$, and an *invariant condition*: a predicate over $\mathbb{R}^n$. There are also actions which are enabled by other predicates; these can change the values of the continuous variables and can change the state (also called "control mode".) In the ϕ-calculus we have both π-actions as in the last subsection, and resetting actions as just described. Hybrid automata allow π-style labels to be attached to certain resetting actions; this is what enables communication between concurrent automata via handshake synchronization. For simplicity, we assume no such labeling, so that we have pure π-actions; in this section we replace the jumps and resets by "environmental actions" on the continuous state. We begin by defining the analogue of "modes."

Definition 4.5 (Environments, informal). *An environment is a triple $\{c; F; I\}$ where $c \in \mathbb{R}^n$, I is an (invariant) predicate on $\mathbb{R}^n$, and F is a dynamic system (typically described by an autonomous differential equation, and typically over $\mathbb{R}^n$).*

Let us be specific, so we can give simple examples. Let $\mathcal{X}$ be an infinite set of "environment variables" disjoint from $\mathcal{N}$ and we let $\dot{\mathcal{X}} = \{\dot{x} \mid x \in \mathcal{X}\}$ be the "dotted versions" (formal variables for differential equations.)

Definition 4.6 (Environments, formal). *An environment consists of:*

- *the* state, *an element of $[\mathcal{V} \to \mathbb{R}]$, the set of all functions (valuations) from a finite subset $\mathcal{V}$ of $\mathcal{X}$ to $\mathbb{R}$. We call $\mathcal{V}$ the* **domain** *of the environment;*
- *the* differential equation*: a valuation from $\dot{\mathcal{V}}$ to $C^1[\mathbb{R}^{\mathcal{U}}]$, the set of continuously differentiable functions from $\mathbb{R}^{\mathcal{U}}$ to $\mathbb{R}$, for a finite set $\mathcal{U} \supseteq \mathcal{V}$;*
- *the* invariant*: a predicate, or map from $[\mathcal{W} \to \mathbb{R}]$ to $\{0, 1\}$, for some $\mathcal{W} \subseteq \mathcal{V}$.*

Example 4.7. *Here is a sample environment:*

$$\left(\begin{array}{c} c = (x : 1.5, y : 0) \\ F = (\dot{x} : x - y - x^3,\ \dot{y} : x + y - y^3) \\ I = \{(x, y) \mid (x \geq 0) \wedge (1 \leq \sqrt{x^2 + y^2} \leq \sqrt{2})\} \end{array} \right).$$

The state c and the (flow of) the differential equation F are both *parts of the environment; here we have specified the flow implicitly.*

We say that a differential equation $F \in [\dot{\mathcal{V}} \to [\mathcal{U} \to \mathbb{R}]]$ is *autonomous* if $\mathcal{U} = \mathcal{V}$. Time-transitions will only be allowed when the equation part of the environment is autonomous.

With regard to the notation for differential equations, all that we are really doing is replacing $\mathbb{R}^n$ with $\mathbb{R}^{\mathcal{U}}$, where $\mathcal{U}$ has n elements, so as to "name" the coordinates of vectors. We allow "free variables" in $\mathcal{U} \setminus \mathcal{V}$ to represent continuous controls.

Definition 4.8 (Environmental actions, informal). *An environmental action (e-action) is comprised of three types of subactions. The first type resets the continuous state (typically setting new initial conditions); the second changes the dynamic system to a new one; and the third updates the invariant predicate. We combine these three kinds into just one kind, which can be invoked conditionally using a predicate ψ on the continuous state. The application is* atomic*: all the subactions of an environmental action are performed indivisibly at the same time. A (conditional)* environmental action, *therefore, is a construct* $[\psi \to (c \doteq d; F \doteq G; I \doteq J)]$ *where* ψ *is a predicate over* $\mathbb{R}^{\mathcal{V}}$. *Any of the four parts of such a construct can be empty.*

Since environmental actions will be programmer-defined, we assume a specific syntax in which the constructs c, F, G, etc. can be written. The syntax will depend on the application, but we can think of $c \doteq d$, $F \doteq G$ as "assignment statements" so that, for example, d can depend on c We read $c \doteq d$ as "c is replaced by d". For our specific setting, defining environments coordinatewise, we have the following specific types of actions:

Definition 4.9 (Environmental actions, formal)**.**

- **"State-reset":** *Let* $c \in [\mathcal{V}_c \to \mathbb{R}]$ *and* $d \in [\mathcal{V}_d \to \mathbb{R}]$. *Then the result of* $c \doteq d$ *is the valuation* s *from* $\mathcal{V}_c \cup \mathcal{V}_d$ *to* $\mathbb{R}$ *given by*
$$s(x) = \begin{cases} c(x) & \textit{if } x \in \mathcal{V}_c \setminus \mathcal{V}_d; \\ d(x) & \textit{otherwise.} \end{cases}$$
- **"Flow-reset":** *Let* $F \in [\dot{\mathcal{V}}_F \to C^1[\mathbb{R}^{\mathcal{U}_F}]]$ *and* $G \in [\dot{\mathcal{V}}_G \to C^1[\mathbb{R}^{\mathcal{U}_G}]]$. *Then the result of* $F \doteq G$ *is*
$$H(x) = \begin{cases} F(\dot{x}) & \textit{if } x \in \mathcal{V}_F \setminus \mathcal{V}_G; \\ G(\dot{x}) & \textit{otherwise.}; \end{cases}$$
- **Invariant-reset:** *Let* $I : [\mathcal{W}_{\mathcal{I}} \to \mathbb{R}] \to \{0,1\}$ *and* $J : [\mathcal{W}_{\mathcal{J}} \to \mathbb{R}] \to \{0,1\}$. *Then the result of* $I \doteq J$ *is*
$$K(x) = \begin{cases} I(x) & \textit{if } x \in \mathcal{W}_I \setminus \mathcal{W}_J; \\ J(x) & \textit{otherwise.} \end{cases}$$

Notice that if domains are disjoint in the above definition, and that if differential equations are autonomous, then environmental replacement produces a "decoupled" autonomous equation. It is also possible to define "superposition" actions; these are obtained by simple coordinatewise addition of the state and differential components where these components are in common to two environments; and where addition of invariants I and J is given by
$$I + J = \{x + y \mid x \in I, y \in J\},$$
regarding "undefined" values as 0 for this purpose.

Environmental actions can be used as prefixes for processes.

Definition 4.10 (ϕ-expressions)**.** *Let* $\mathcal{E}$ *be the set of environmental actions. We extend the set of process prefixes by simply adjoining all of the actions in* $\mathcal{E}$ *(Any of these prefixes will be signified by* μ*.) Environmental actions do not have negatives; in this respect they are like* τ*-actions. By allowing these prefixes in any expression we obtain the set of hybrid* ϕ*-expressions.*

It is straightforward to extend the notion of structural congruence to hybrid expressions.

Definition 4.11 (Embedded system)**.** *An embedded system is a pair* (E, P) *where* E *is an environment and* P *is a* ϕ*-expression.*

Now we turn to the transitions which an embedded system can make. First, any of the π-actions are still allowed: if $P \xrightarrow{\alpha} Q$ then $(E, P) \xrightarrow{\alpha} (E, Q)$ for $\alpha \in \mathcal{L} \cup \{\tau\}$.

Definition 4.12 (Environmental action transitions)**.** *This set of transitions is given by the following rules, where e is the environmental action $\psi \to (c \doteq d; F \doteq G; I \doteq J)$, E is the environment (c, F, I), and (s, H, K) is the environment given by Definition 4.9:*

$$\text{Sum: } (E, M + e.P + N) \xrightarrow{e} ((s, H, K), P) \text{ if } \phi(c) \text{ is true;} \tag{6}$$

$$\text{LPar: } \frac{(E, P) \xrightarrow{e} (E', P')}{(E,\ P \parallel Q\ ) \xrightarrow{e} (E',\ P' \parallel Q\ )} \qquad \text{RPar: } \frac{(E, Q) \xrightarrow{e} (E', Q')}{(E,\ P \parallel Q\ ) \xrightarrow{e} (E',\ P \parallel Q'\ )}; \tag{7}$$

$$\text{Res: } \frac{(E, P) \xrightarrow{e} (E', P')}{(E, \nu a P) \xrightarrow{e} (E', \nu a P')}. \tag{8}$$

An embedded system can evolve over time. This entails a new kind of rule called a *flow transition* rule. In this section we propose provisional rules, which will have to be strengthened in a later section in order to allow weakly bisimilar processes to behave the same way in an arbitrary continuous environment.

Definition 4.13 (Flow transitions - provisional)**.** *Suppose that a state $x \in [\mathcal{V} \to \mathbb{R}]$ and that the differential equation $F \in [\dot{\mathcal{V}} \to [\mathcal{U} \to \mathbb{R}]]$ is autonomous, i.e., $\mathcal{U} = \mathcal{V}$. Then the flow $\phi(t, x)$ of the equation will be defined in some time-interval $J = [0, u)$ of $\mathbb{R}$. We then have the following flow transitions:*

$$\text{Sum: } (\{x, F, I\}, \Sigma_i \mu_i P_i) \xrightarrow{t} (\{\phi(t, x), F, I\}, \Sigma_i \mu_i P_i) \tag{9}$$

provided that for all $0 \le s < t$: (i) $\phi(s, x)$ is defined and in I ; and (ii) no μ_i is an environmental action $\psi \to (c \doteq d; F \doteq G; I \doteq J)$ with $\psi(s)$ true.

Flow transitions are extended to other ϕ-expressions by

$$\text{Par: } \frac{(E, P) \xrightarrow{t} (E', P) \qquad (E, Q) \xrightarrow{t} (E', Q)}{(E,\ P \parallel Q\ ) \xrightarrow{t} (E',\ P \parallel Q\ )} \tag{10}$$

$$\text{Res: } \frac{(E, P) \xrightarrow{t} (E', P)}{(E, \nu a P) \xrightarrow{t} (E', \nu a P)} \tag{11}$$

We illustrate transitions with a simple example.

Example 4.14. Consider the embedded system

$$(\emptyset\ , [TRUE \to x \doteq 0; F \doteq (\dot{x} : 1); I \doteq (x \le 3)].b).$$

This process runs in the *null environment* $\emptyset$, where the valuations all have empty domains. In this environment, it will have an e-transition to

$$(\begin{pmatrix} x : 0 \\ \dot{x} : 1 \\ (x \le 3) \end{pmatrix},\ b),$$

initializing a clock which runs for at most 3 seconds. During any portion of this time, it may do the discrete action b, but must do b within 3 seconds. So, for example,

$$\begin{aligned}
(\begin{pmatrix} x : 0 \\ \dot{x} : 1 \\ x \leq 3 \end{pmatrix}, b) &\overset{0.5}{\rightarrow} (\begin{pmatrix} x : 0.5 \\ \dot{x} : 1 \\ x \leq 3 \end{pmatrix}, b) \\
&\overset{b}{\rightarrow} (\begin{pmatrix} x : 0.5 \\ \dot{x} : 1 \\ x \leq 3 \end{pmatrix}, 0)
\end{aligned}$$

in which the action b happens after one-half second.

Remark 4.15 (Explaining flow transition rules)**.**

(1) The Sum rule (provisional) reflects evolution over time in a control mode; the sum represents possible exit transitions from the mode. Note that all parts of the environment must be defined for time to progress here. For later reference we abbreviate (i) by saying the flow is in I during t, and abbreviate condition (ii) by saying that there is no environmental guard enabled in P during time t. The reason for (ii) is to make enabled environmental actions "eager"; see the example below.

(2) The rule *Par* (provisional) also needs explanation. One might expect a rule which mirrors the standard notion of direct product, so would involve some "environmental product". Processes, however, can create their own environments via *Sum* environmental action. If we have a parallel composition of two sums, then each sum can manufacture the part of the environment it needs to progress over time. When both parallel sums have contributed their own local environment, then in effect the necessary product environment will have been created, and both processes then can progress simultaneously. Notice also that for parallel composition, no compatibility conditions as in Henzinger [9] need to be assumed.

Example 4.16 (Eagerness)**.** Why not let any sum evolve over time? Here's an example involving parallel composition. Consider

$$P \parallel Q \;=\; \begin{bmatrix} x \doteq 0 \\ I \doteq x \leq 3 \\ F \doteq (\dot{x} = 1) \end{bmatrix} .b \parallel \begin{bmatrix} y \doteq 0 \\ I \doteq y \leq 5 \\ F \doteq (\dot{y} = 1) \end{bmatrix} .c\,.$$

Our intention is that, starting in the null environment, P and Q both initialize their part of the global environment and then evolve simultaneously in parallel over time, so that the composition $P \parallel Q$ cannot get past time 3 without blocking because of P's invariant. If we let any sum evolve over time, then Q can put its y definition into the environment and the process $P \parallel c$ can evolve until $y = 5$, not what we want. But with the Par and restricted Sum time transition rules, both processes have to use their local environment descriptions to update the actual global environment before any time evolution can happen, so we get the desired simultaneity.

Example 4.17 (Timeouts.)**.** We illustrate the ϕ_c calculus by defining the "timeout" *VM* promised earlier. It is convenient to define the *timeout* operator on processes given in Schneider's book on timed CSP [19, page 275]. Suppose d is a non-negative real constant. Paraphrasing Schneider:

> The process $Q_1 \overset{d}{\triangleright} Q_2$ (read "Q_1 timeout d Q_2") offers a time-sensitive choice between Q_1 and Q_2. If Q_1 performs an (observable) action before d units of time have elapsed, then the choice is resolved for Q_1 and Q_2 is discarded. If Q_1 performs no such action, then the process Q_2 is enabled after d units of time and Q_1 is discarded.

To achieve this behavior, the definition is simply:

$$Q_1 \overset{d}{\triangleright} Q_2 \overset{\text{def}}{=} [x \doteq 0; F \doteq (\dot{x} = 1)].(Q_1 + [x = d].Q_2)$$

where x is not free in Q_1 or Q_2.[1] Notice that we have had to add the new variable x to the environment, which suggests that we make this variable local. We ignore this question for now, and define

$$TVM = cof.TVM \overset{1}{\triangleright} (tea.TVM \overset{1}{\triangleright} TVM).$$

(The two timeouts here will have to be defined with distinct timer variables.)

Although there seems to be a recursive definition here, it can be eliminated by writing out all the abbreviations. However, this example points out the need both for localizing environmental variables and for recursive definition capabilities. We revisit the matter in Section 6.

Example 4.18 (Petri nets)**.** Consider a Petri net N with place set P, and transition set $\{t_1, \dots t_k\}$. We consider markings m of the net where a place can contain any nonnegative integer, and where a transition can fire if all of its input places are positive. Firing is a vector operation which subtracts 1 from each input place and adds 1 to each output place. (This is again just for definiteness.) Let v_t be the vector representing the difference of these two vectors: v_t should be added to a marking m by transition t to achieve the new marking. Then for each transition t we describe a ϕ_c process Q_t which will manipulate markings as environments:

$$Q_t ::= [m_{in(t)} > 0 \rightarrow m \doteq m + v_t].Q_t$$

where $m_{in(t)}$ is the subvector of the marking m which is indexed by the input places for the transition t.

The whole Petri net is then just the composition

$$Q_{t_1} \parallel \ \dots \ \parallel Q_{t_k}$$

run in the environment of the initial marking.

Of course, for Petri nets, environmental actions are over a discrete dynamical system, not a continuous one.

5. On bisimulations

The notion of *strong bisimulation* is important in the π-calculus, as this relation between two processes ensures that one of them can be substituted for the other in any process context. Algebraically, the relation of strong bisimilarity is a congruence relation with respect to the process composition operators. We want a definition of strong bisimulation which leads to congruences with respect to the standard process composition operators. The following is an obvious candidate.

Definition 5.1. *A relation R over ϕ-expressions is a* strong simulation*if whenever $P \ R \ Q$ and for any action (either π or e) μ such that $P \xrightarrow{\mu} P'$, there is Q' such that $Q \xrightarrow{\mu} Q'$ and $P' \ R \ Q'$. We then can say We say that P and Q are* strongly bisimilar*, and write $P \sim Q$, if there is a relation R on process expressions, relating P with Q, such that both R and R^{-1} are strong simulations.*

We want this definition to preserve flow actions as well as π and e-actions. What does this mean? One answer is that we can consider environments as "surrounding contexts". Thus we have:

Proposition 5.2. *If P is strongly bisimilar to Q, then whenever $(E, P) \xrightarrow{t} (E', P)$ we have $(E, Q) \xrightarrow{t} (E', Q)$.*

[1]To use the operator $+$, we need also to assume that Q_1 is a sum.

To prove this proposition, it is easiest to begin a lemma, which extends (1) of Remark 4.15 to structured processes. In the statement, the notation $P \xrightarrow{e}$ means that for some P', $P \xrightarrow{e} P'$ (without reference to the environment).

Lemma 5.3. *For all environments $E = \{x, F, I\}$ with F autonomous, and for any process P, we have that $(E, P) \xrightarrow{t} (E', P)$ if and only if $E' = \{\phi(x,t), F, I\}$, the flow is in I, and no e with $P \xrightarrow{e}$ is enabled during time t.*

Proof. We proceed by induction on the structure of the process P. In the "base case", when P is a sum, then the assertion is just the Sum transition rule. For the inductive step, we consider only the case when P is $P_1 \parallel P_2$. Then $(E, P) \xrightarrow{t} (E', P)$ iff $(E, P_1) \xrightarrow{t} (E', P_1)$ and $(E, P_2) \xrightarrow{t} (E', P_2)$, by the Par flow transition rule. Assume $(E, P) \xrightarrow{t} (E', P)$. The inductive hypothesis applies to (E, P_1) and (E, P_2) so that $E' = \{\phi(x,t), F, I\}$, there is no enabled guard for any e such that $P_i \xrightarrow{e}$, and the flow is in I. Since the Lpar and Rpar rules ((7) in Definition 4.12) are the only rules allowing inference of an e-action for a parallel composition, the same holds for (E, P). Conversely, assume that $E' = \{\phi(x,t), F, I\}$, the flow is in I, and no e with $P \xrightarrow{e}$ has a true guard during time t. Then $(E, P_i) \xrightarrow{t} (E', P_i)$ by inductive hypothesis, and consequently $(E, P) \xrightarrow{t} (E', P)$ by the Par flow rule. This completes the inductive step for this case; the corresponding step for $P = \nu a P_1$ is easier. $\square$

Proof of Proposition 5.2. Assume P is strongly bisimilar to Q. If $(E, P) \xrightarrow{t} (E', P)$ then by the lemma $E' = \{\phi(x,t), F, I\}$, the flow is in I, and no e with $P \xrightarrow{e}$ has a true guard during time t. By bisimilarity the same statement is true for Q, because $P \xrightarrow{e}$ iff $Q \xrightarrow{e}$. By the lemma in the converse direction, we have $(E, Q) \xrightarrow{t} (E', Q)$. $\square$

This proposition shows that strong bisimilarity has the expected properties of bisimulations even when flow transitions are allowed. Furthermore, we have obvious congruences:

Proposition 5.4. *If $P \sim Q$ then*
- *$\mu.P + M \sim \mu.Q + M$, where M is any sum.*
- *$P \parallel R \sim Q \parallel R$.*
- *$\nu a P \sim \nu a Q$.*

The proof is straightforward. $\square$

Interestingly, the natural extension of weak bisimulation from CCS to ϕ_c fails to be an environmental congruence in the sense of Proposition 5.2. The problem is not with the definition of weak bisimulation itself, but with the Sum flow transition rule. This rule forces enabled environmental actions to be eager, but does not force reactions to be eager. To see what is happening, we start by defining the natural notion of weak bisimulation for ϕ_c:

Definition 5.5. *Let $\Rightarrow$ be the reflexive transitive closure of $\xrightarrow{\tau} \circ \equiv$, and let $\overset{\mu}{\Rightarrow}$ be the composition $\Rightarrow \circ \xrightarrow{\mu} \circ \Rightarrow$, where μ can be either a π-action α or an environmental action e. A relation R over ϕ-expressions is a* weak simulation *if whenever $P \; R \; Q$ and for any action μ such that $P \overset{\mu}{\Rightarrow} P'$, there is Q' such that $Q \xrightarrow{\mu} Q'$ and $P' \; R \; Q'$. We say that P and Q are* weakly bisimilar*, and write $P \approx Q$, if there is a relation R on process expressions, relating P with Q, such that both R and R^{-1} are weak simulations.*

We now show that Proposition 5.2 fails for weak bisimulations:

Example 5.6. Consider the processes

$$P = \nu a(\ a.[TRUE \to x \doteq 1] \parallel \overline{a}\ ) \text{ and } Q = [TRUE \to x \doteq 1].$$

These are weakly bisimilar, but the environment $E = \begin{pmatrix} x : 0 \\ \dot{x} : 1 \\ TRUE \end{pmatrix}$, in the context of process P, can flow for arbitrarily long periods before a reaction happens, while the same E in the context of the process Q is time-blocked because Q's environmental action is enabled.

It turns out that to deal with weak bisimulation, we need to revise the Sum and Par flow transition rules so that with respect to this new kind of bisimulation, weakly enabled environmental transitions, as well as weakly possible reactions, are made eager. We carry out this revision in Section 7, when we have introduced the full hybrid calculus.

6. The full ϕ-calculus

In this section we extend the full π-calculus to the full ϕ-calculus, which will therefore deal with name-passing and recursion (replication). We address first the status of environmental variables as names which can be passed from one process to another in a reaction. We recall the π-calculus rules for abstractions and concretions, and extend these to the π-calculus. Then we introduce *environmental restriction*, a way of localizing environmental variables. We then give the commitment rules for the ϕ-calculus. Lastly in the section, we introduce the rules for recursion.

6.1. Abstractions, concretions, and name-passing.

As above let $\mathcal{X}$ the set of environment variables. We define *output prefixes* to be of the form $\overline{a}\langle y_0, \ldots, y_{n-1}\rangle$, where $a \in \mathcal{N}$, and where the y's are in $\mathcal{N} \cup \mathcal{X}$, together with *input prefixes* of the form $a(y_0, \ldots, y_{n-1})$ with the same convention on the y's. We can think of variables in $\mathcal{X}$ as *links to the environment.* As such they can only have values in $\mathbb{R}$, and cannot be used as message channels. However, these variables can themselves be passed as messages between processes[2]. The syntax of ϕ-expressions is just as before, only now accommodating the new sorts of prefixes. (Replication will be added later in the section.)

We recall the π-calculus definitions of abstractions and concretions.

Definition 6.1 (Milner). *An* abstraction *of* arity $n \geq 0$ *takes the form* $(\vec{w}).P$*, where* $|w| = n$*. The letters* $G, H, \ldots,$ *stand for abstractions. Two abstractions are structurally congruent* ($\equiv$) *if, up to* α*-conversion, their bound names* $\vec{x}$ *are identical and their process parts are structurally congruent.*

A concretion *of* arity $n \geq 0$ *takes the form* $\nu\vec{z}\langle\,\vec{y}\,\rangle.P$ *where* $|\vec{y}| = n$ *and* $\vec{z} \subseteq \vec{y}$*. The letters* C, D *stand for concretions. Structural congruence for concretions is like that for abstractions, together with allowing restricted names to be reordered.*

An agent *is an abstraction or a concretion. The class of* π*-agents is denoted* A^{π}*, and the letters* A, B *stand for agents.*

We extend abstraction operators to the ϕ-calculus. Again, let $\mathcal{X}$ be the set of environmental variables. ϕ-abstractions simply allow $\mathcal{X}$-variables to occur as part of ordinary abstractions and concretions $(\vec{w}).P$ and $\langle\,\vec{y}\,\rangle.P$. Allowing these variables in $\vec{w}$ will allow for the passing of environmental variables as *links to the environment.* Substitutions must of course repect these two types of names. In particular, if x is an $\mathcal{X}$-variable in $\vec{w}$, substitutions for x are for occurrences of x in action prefixes and environmental actions inside the scope of $(\vec{w})$. As always we allow for α-conversion in the appropriate sense. See Example 6.4 below.

Definition 6.2. *Abstractions and concretions in the* ϕ*-calculus have the following forms:*

- ϕ*-abstractions are of the form* $(\vec{w}).P$ *for* $\vec{w}$ *over* $\mathcal{X} \cup \mathcal{N}$*.*

[2]We could even allow differential equation names to be passed, but omit this here.

- *ϕ-concretions are of the form $\nu\vec{z}\langle\,\vec{y}\,\rangle.P$, where $\vec{y}$ may include variables in $\mathcal{X}$ but $\vec{z} \subseteq \vec{y}$ may have no such variables.*

Again, an agent *is either an abstraction or a concretion.*

Abstractions apply to concretions more or less exactly as in the π-calculus.

Definition 6.3. *The* application $G@C$ *of a ϕ-abstraction and concretion is defined as follows:*

$$((\vec{w})P)@\nu\vec{z}\langle\,\vec{y}\,\rangle.Q =_{\mathrm{def}} \nu\vec{z}(\ \{\vec{y}/\vec{w}\}P \parallel Q\ )$$

where $\vec{y}$ and $\vec{w}$ have the same length, and where names in $\mathcal{N}$ must be substituted for names in $\mathcal{N}$, and names in $\mathcal{X}$ for names in $\mathcal{X}$.

Applications can of course be carried out in the context of an environment E.

Example 6.4. *For any E*

$$(E, (axb).a.[x \doteq 2].b@\langle\, cyd\,\rangle.[y \doteq 3]) = (E,\ c.[y \doteq 2].d \parallel [y \doteq 3]\ ).$$

We now introduce the *environmental restriction* operator $\mathsf{new}\ xP$ for $x \in \mathcal{X}$. The intent of this operator is to make these variables local to the process P. In the literature, this operator is called "variable hiding", and sometimes "abstraction". In ϕ-calculus usage, however, it is much more akin to the restriction operator νa. Whereas abstractions apply to concretions, this "abstraction" operator has no corresponding concretion. So we define it as an operator from processes to processes.

Definition 6.5. *For $x \in \mathcal{X}$ the operator $\mathsf{new}\ xP$ is called the* environmental restriction *operator. This operator restricts any e-action mentioning the variable x or $\dot{x}$ (we call such actions x-actions). We allow for α-conversions under the scope of $\mathsf{new}\ x$, replacing occurrences of x in x-actions by a new variable not occurring free in P.*

We turn to the rules for commitments in the full ϕ-calculus. For brevity we do not define reaction rules separately, as is done in Milner.

Definition 6.6 (ϕ-commitments)**.** *The commitments of embedded systems (aside from those involving the replication operator) are given in Figure 6.1. In the Sum-e rules, e is an environmental action of the form $\psi \to (c \doteq d; F \doteq G; I \doteq J)$ and E is the environment (c, F, I). Further, $\alpha \in \mathcal{N} \cup \overline{\mathcal{N}}$, and μ is either an α or an e.*

Notice the effect of the *Env*-commitment rule: it is simply to introduce a fresh new variable into P. This variable is local to the scope of $\mathsf{new}\ x$, as desired. Note: the substitution w/x includes occurrences of the variable x in e-actions under the scope of $\mathsf{new}\ x$.

The flow transition rules for the ϕ-calculus include those of the ϕ_c-calculus. Flow transitions are allowed for processes (and not abstractions) as in Definition 4.13. We need to add a flow transition rule for the environmental restriction operator; this works in analogy with the *Res* rule in Definition 4.13.

$$Res - x\text{:}\quad \frac{(E, P[w/x]) \xrightarrow{t} (E', P[w/x])}{(E, \mathsf{new}\ xP) \xrightarrow{t} (E', \mathsf{new}\ xP)}\ (w \text{ not mentioned in } E\text{, not free in } P).$$

The reason for the side conditions can be seen from an example:

Example 6.7. Let $P = \mathsf{new}\ x[x \doteq 1]$ and $Q = \mathsf{new}\ y[y \doteq 1]$. Then P and Q are structurally equivalent by α-conversion. Let E be the environment $\begin{pmatrix} x:0 \\ \dot{x}:1 \\ TRUE \end{pmatrix}$. If we did not have the

$$Sum - pi : (E,\ M + \alpha A + N) \xrightarrow{\alpha} (E, A) \qquad Sum - e : (E, M + e.P + N) \xrightarrow{e} ((d, G, J), P) \ \text{ if } \psi(c) \text{ is true;}$$

$$L - React : \frac{(E,P) \xrightarrow{w} (E,G) \qquad (E,Q) \xrightarrow{\overline{w}} (E,C)}{(E,\ P \parallel Q\ ) \xrightarrow{\tau} (E, G@C)} \qquad R - React : \frac{(E,P) \xrightarrow{\overline{w}} (E,C) \qquad (E,Q) \xrightarrow{w} (E,G)}{(E,\ P \parallel Q\ ) \xrightarrow{\tau} (E, G@C)}$$

$$L - par : \frac{(E,P) \xrightarrow{\mu} (E', A)}{(E,\ P \parallel Q\ ) \xrightarrow{\mu} (E',\ A \parallel Q\ )} \qquad R - par : \frac{(E,Q) \xrightarrow{\mu} (E', A)}{(E,\ P \parallel Q\ ) \xrightarrow{\mu} (E',\ P \parallel A\ )}$$

$$Res - pi : \frac{(E,P) \xrightarrow{\mu} (E', A)}{(E, \nu w P) \xrightarrow{\mu} (E', \nu w A)} \ \text{ if } \mu \notin \{w, \overline{w}\}$$

$$Res - x : \frac{(E,P) \xrightarrow{\mu} (E', Q)}{(E, \mathsf{new}\ xP) \xrightarrow{\mu} (E', \mathsf{new}\ xQ)} \ \text{ if } \mu \text{ is not an } x\text{-action}$$

$$Env : (E, \mathsf{new}\ xP) \xrightarrow{\tau} (E, P[w/x]) \ \ (w \in \mathcal{X},\ w \text{ not mentioned in } E \text{ and not free in } P.)$$

FIGURE 1. Commitment rules for the ϕ-calculus

side conditions, then $(E, P) \xrightarrow{t}$ for any t, but (E, Q) cannot flow because y is not defined in the environment. Once again, the new x declaration in P enforces that the local variable x is different from any variable in the environment of P.

6.2. **Recursion.** The π-calculus rule for recursion is

$$!P \equiv\ P \parallel !P$$

where $\equiv$ is structural equivalence. We adopt this for the ϕ-calculus, and we handle commitments just as in the π-calculus with provision for possible environmental changes. Thus when μ is either a π-action or an e-action

$$\frac{(E,\ P \parallel !P\ ) \xrightarrow{\mu} (E', Q)}{(E, !P) \xrightarrow{\mu} (E', Q)}.$$

The only other thing that we need to add is the flow transition rule. This is simply

$$\frac{(E,P) \xrightarrow{t} (E', P)}{(E, !P) \xrightarrow{t} (E', !P)}.$$

Notice that this rule dovetails well with the *Par* rule in Definition 4.13 together with the structural equivalence for recursion above.

Milner provides a translation of recursively defined processes into replicated parallel processes, which works exactly the same way in the ϕ-calculus. To review from [17]:

To encode the definition

$$A(\vec{x}) \overset{\text{def}}{=} Q_A, \text{where } Q_A = \cdots A(\langle\, \vec{u}\, \rangle \cdots A(\langle\, \vec{v}\, \rangle) \cdots$$

where the scope of the definition is some process

$$P = \cdots A\langle\, \vec{y}\, \rangle \cdots A\langle\, \vec{z}\, \rangle \cdots$$

(1) Invent a new name a to stand for A;
(2) For any process R, denote by $\widehat{R}$ the result of replacing every call $A\langle\, \vec{w}\, \rangle$ by $a\langle\, \vec{w}\, \rangle$;

(3) Replace P and the accompanying definition of A by

$$\widehat{\widehat{P}} \overset{\text{def}}{=} \nu a(\ \widehat{P} \parallel\ !a(\vec{x}).\widehat{Q_A})\ .$$

This translation has the expected effect of allowing us to unfold recursive definitions of the standard kind by replacing recursive calls of a process "textually" by the original definition.

Example 6.8 (Timeouts revisited.)**.** Consider again the definition

$$Q_1 \overset{d}{\triangleright} Q_2 \overset{\text{def}}{=} [x \doteq 0; F \doteq (\dot{x} = 1)].(Q_1 + [x = d].Q_2)$$

where x is not free in Q_1 or Q_2. Localizing the x variable is simple:

$$Q_1 \overset{d}{\triangleright} Q_2 \overset{\text{def}}{=} \mathsf{new}\ x([x \doteq 0; F \doteq (\dot{x} = 1)].(Q_1 + [x = d].Q_2)).$$

Using Milner's translation, the definition

$$TVM = cof.TVM \overset{1}{\triangleright} (tea.TVM \overset{1}{\triangleright} TVM)$$

now makes sense (provided of course that Q_1 is actually a sum).

Recursive constructions, commitments, and name-passing will be illustrated in Section 8 where we show how to represent a physical buffer consisting of robots moving back and forth in a "bucket brigade."

7. Extended bisimulations

In this section we extend the strong bisimulations of Section 5 to the full ϕ-calculus, introduce weak bisimulations and revise the flow transition rules so that we obtain similar behavior of weakly bisimilar processes in all environmental contexts.

7.1. **Strong bisimulation.**

As in Milner let $\mathcal{A}^\phi$ be the set of ϕ-calculus agents and let $\mathcal{P}^\phi$ be the set of processes. A binary relation S over $\mathcal{P}^\phi$ can be extended to a relation between abstactions of like arity and concretions of like arity using

$$F\ S\ G \text{ means that for all } \vec{y},\ F\langle\vec{y}\rangle\ S\ G\langle\vec{y}\rangle.$$
$$C\ S\ D \text{ means that } C \equiv \nu\vec{z}\langle\vec{y}\rangle.P \text{ and } D \equiv \nu\vec{z}\langle\vec{y}\rangle.Q \text{ such that } P\ S\ Q.$$

Definition 7.1. *A binary relation S over $\mathcal{P}^\phi$ is a strong simulation if whenever $P\ S\ Q$, and μ is either an α or e action,*

$$\textit{if } P \xrightarrow{\mu} A \textit{ then there exists } B \textit{ such that } Q \xrightarrow{\mu} B \textit{ and } A\ S\ B.$$

If both S and its converse are strong simulations then S is called a strong bisimulation. Two agents A and B are strongly bisimilar, written $A \sim B$, if there is a strong bisimulation relating them.

The extension of Proposition 5.2 to the full ϕ-calculus follows from Theorem 7.13 below and the fact that a strong bisimulation is a weak bisimulation.

7.2. **Weak bisimulation.**

We extend Milner's definition of input and output experiments to the ϕ-calculus. The definitions are mostly unchanged from the π-calculus, but we now must recognize that we have environmental actions e, that environmental variables can be passed as messages, and that there is a new kind of τ-transition corresponding to the *Env*-commitment.

We begin by defining some notions for process and agent expressions only, and then lifting them to embedded processes and agents. So, considering ϕ-process expressions only, define the relation $\Rightarrow$ to be the reflexive transitive closure of $\xrightarrow{\tau} \circ \equiv$, where $\equiv$ is structural equivalence.

Definition 7.2 (Experiment)**.** *The relation* $\overset{a\langle\vec{y}\rangle}{\rightarrow}$ *over* $\mathcal{P}^{\phi}$ *is defined as follows:*

$$P \overset{a\langle\vec{y}\rangle}{\rightarrow} P' \textit{ iff, for some } G,\ P \xrightarrow{a} G \textit{ and } G\langle\vec{y}\rangle \equiv P'.$$

An input experiment *is an instance of the relation* $\overset{a\langle\vec{y}\rangle}{\Rightarrow}$*, defined as*

$$P \overset{a\langle\vec{y}\rangle}{\Rightarrow} P' \textit{ iff } P \Rightarrow\overset{a\langle\vec{y}\rangle}{\rightarrow}\Rightarrow P'.$$

An output experiment *is an instance of the relation* $\overset{\overline{a}}{\Rightarrow}$*, where*

$$P \xrightarrow{\overline{a}} \nu\vec{z}\langle\vec{y}\rangle.P' \textit{ iff, for some } P'',\ P \Rightarrow\xrightarrow{\overline{a}} \nu\vec{z}\langle\vec{y}\rangle.P'' \textit{ and } P'' \Rightarrow P';$$

An e-experiment *is an instance of the relation* $\overset{e}{\Rightarrow}$*, where*

$$P \xrightarrow{e} P' \textit{ iff } P \Rightarrow\xrightarrow{e}\Rightarrow P'.$$

(The last action $\xrightarrow{e}$ *above just uses e as a* π*-prefix.)*

In what follows, we consider *embedded experiments*, defined as (the pair of endpoints of) sequences of commitments (where μ can be any of the actions in the above definition)

$$(E,P) \xrightarrow{\tau} (E,P_1) \xrightarrow{\tau} \dots \xrightarrow{\tau} (E,P_n) \xrightarrow{\mu} (E',P').$$

The final μ-commitment in the sequence is a commitment that is possible for (E,P_n), and may change the environment E.

Remark 7.3.

(1) Given an embedded experiment, we can pass by projection to the underlying process expressions in the sequence to obtain a non-embedded experiment. We just examine the commitments at each step, and notice that τ-commitments are accompanied by underlying τ-transitions in the process expressions at the underlying level. (Remember that we have extended τ-transitions at the underlying level to take account of *Env*-commitments.)

Conversely, if there is a non-embedded experiment connecting P to P', then there is a sequence

$$(*) \qquad P \xrightarrow{\tau} P_1 \xrightarrow{\tau} \dots \xrightarrow{\tau} P_n \xrightarrow{\mu} P'.$$

Given an environment E, we will construct a corresponding embedded experiment connecting (E,P) to a certain (E',Q'), whose projection will be another experiment starting with P and ending with a μ'-transition to Q'. The action μ' will be μ if $\mu \in \mathcal{N}$, and otherwise will be the environmental action e, some of whose free variables will have been substituted by others.

Given the sequence $(*)$, let E be an environment in which the set $V(E)$ of variables is mentioned. Proceeding from left to right along the sequence $(*)$, we construct a new embedded experiment

$$(E,P) \xrightarrow{\tau} (E,Q_1) \xrightarrow{\tau} \dots \xrightarrow{\tau} Q_n \xrightarrow{\mu'} Q',$$

with the property that each Q_i arises from P_i by making a substitution of new variables for free variables of P_i. Let Q_0 be P. Assume that (E,Q_i) has been constructed. If the transition $P_i \xrightarrow{\tau} P_{i+1}$ is not one of the new *Env*-transitions, then Q_{i+1} is just P_{i+1} with the same substitution of free variables that Q_i had. If $(*)$ has a transition $P_i = \mathsf{new}\ xP'_i \xrightarrow{\tau} P'_i[w/x]$, then Q_i will be of the form $\mathsf{new}\ xQ'_i$. Let z be a fresh variable not occurring in $V(E)$ and not free for x in Q'_i, and put $Q_{i+1} = P'_i[z/x]$. Having constructed (E,Q_n), suppose that e was the last action in $(*)$. Replace any variables in e by the ones which have been replaced into Q_n from P_n. This gives the new environmental action e'. Finally, let Q' be P' with the same updated variables. It is clear that the embedded sequence we constructed projects in the desired way.

(2) In the rest of the section, we will often consider *barbed* e-experiments, which are defined by sequences of $\tau \circ \equiv$-actions, followed by a last e-action. We will use the fact that, in either the embedded or non-embedded setting, a process admits some e-experiment if and only if it admits some barbed e-experiment.

Definition 7.4 (Weak simulation). *A binary relation R over $\mathcal{A}^\phi$ is a* weak simulation *if, whenever $P\ R\ Q$,*

$$\text{if } P \Rightarrow P' \text{ then there is } Q' \text{ such that } Q \Rightarrow Q' \text{ and } P'\ R\ Q';$$

$$\text{if } P \overset{a\langle\vec{y}\rangle}{\Rightarrow} P' \text{ then there is } Q' \text{ such that } Q \overset{a\langle\vec{y}\rangle}{\Rightarrow} Q' \text{ and } P'\ R\ Q';$$

$$\text{if } P \overset{\mu}{\Rightarrow} C, \text{ where } \mu \text{ is either } \overline{a} \text{ or } e, \text{ then there is } D \text{ such that } Q \overset{\mu}{\Rightarrow} D \text{ and } C\ R\ D.$$

If both R and its converse are weak simulations, R is a weak bisimulation. Two processes P and Q are weakly bisimilar, written $P \approx Q$, if there is some weak bisimulation relating them.

In this definition, of course, the experiments are not embedded.

Now we need to show that environments of weakly bisimilar processes flow the same way ; but as we showed in Example 5.6, this does not hold with the Sum flow transition rule in its current form. We revise this rule, and we also make reactions eager by revising the flow transition rule for Par.

Definition 7.5 (Revised flow rules). *We present all of the new flow rules in one display. In the following, E is an environment having an autonomous differential equation with flow $\phi(s,x)$ out of the state x, such that for a given time t, we have that for all $0 \le s < t$, $\phi(s,x)$ is defined and in the invariant set I.*

$$\textit{Sum: } (E, \Sigma_i \mu_i P_i) \xrightarrow{t} (E', \Sigma_i \mu_i P_i)$$

$$\textit{iff no } \mu_i \textit{ is an enabled } e\textit{-action in } E, \textit{ and for every } \tau \textit{ such that } \mu_i = \tau \textit{ we have } (E, P_i) \xrightarrow{t} (E', P_i)$$

$$\textit{Par: } \frac{(E,P) \xrightarrow{t} (E',P) \qquad (E,Q) \xrightarrow{t} (E',Q)}{(E,\ P \parallel Q\ ) \xrightarrow{t} (E',\ P \parallel Q\ )} \textit{ provided } (\forall S)(\ P \parallel Q\ \ \xrightarrow{\tau} S \textit{ implies } (E,S) \xrightarrow{t} (E',S)).$$

$$\textit{Res-}\pi\textit{: } \frac{(E,P) \xrightarrow{t} (E',P)}{(E, \nu a P) \xrightarrow{t} (E', \nu a P)}$$

$$\textit{Res-x: } \frac{(E, P[w/x]) \xrightarrow{t} (E', P[w/x])}{(E, \mathsf{new}\ xP) \xrightarrow{t} (E', \mathsf{new}\ xP)} \textit{ provided } w \textit{ is not free in } P \textit{ and not mentioned in } E.$$

$$\textit{Rep: } \frac{(E,P) \xrightarrow{t} (E',P)}{(E, !P) \xrightarrow{t} (E', !P)}.$$

It is understood in these rules that "P" refers to the structural equivalence class of P.

With this revision of the flow laws, we embark on the proof of our context theorem. We begin by passing to a non-embedded version of the flow laws. We define the *non-embedded* flow laws for ϕ-calculus expressions to be the projections of the flow laws for embedded expressions. In this setting, the relation "can flow for t seconds" just becomes a unary predicate of P, which we denote $Fl(P)$, suppressing t, which is fixed. Thus, for example, the non-embedded *Par* rule is

$$\text{Par: } \frac{Fl(P) \qquad Fl(Q)}{Fl(\ P \parallel Q\ )} \text{ provided } (\forall S)(\ P \parallel Q\ \xrightarrow{\tau} S \text{ implies } Fl(S)).$$

and

$$\text{Res-}x\text{: } \frac{Fl(P[w/x])}{Fl(\mathsf{new}\ xP)} \text{ provided } w \text{ is not free in } P.$$

This version of the laws reveals a basic structure; the laws are *τ-persistent*: for every τ-commitment possible for a process, the process committed to must be able to flow. This holds for the Sum, Par, and ($\mathsf{new}\ x$) rules, and has the effect in practice of making all of these (embedded) commitments eager: time cannot flow unless one of the τ-commitments available to a process in one of these forms actually takes place.

Lemma 7.6. *Let T be any τ-persistent property of ϕ-calculus processes, Then for all P, $T(P)$ holds if and only if for all P' with $P \xrightarrow{\tau^*} P'$ we have $T(P')$.*

Proof. The $\Leftarrow$ direction in the statement is trivial. The proof of $\Rightarrow$ is by structural induction on P, but we must be careful here, because we consider only P up to structural equivalence. The proof is therefore by induction on the (length of) the shortest representative of a structural equivalence class.

Basis: $P = 0$. This is clear, as 0 has no τ-commitments.

Induction: cases on the form of P.

(1) P is a sum. Assume $T(P)$. The only τ-commitments of a sum are to subprocesses, so the conclusion follows by τ-persistence of T.
(2) $P =\ P_1 \parallel P_2$. There are three different kinds of τ-transitions possible for P: those introduced by means of the Lpar and Rpar rules, and those introduced by the React rule. In each case, the process committed to is shorter than $P_1 \parallel P_2$, so the conclusion follows again by τ-persistence.
(3) The case of $\nu a P'$ is clear, as the τ-commitments of $\nu a P'$ are just those of P'.
(4) $P = \mathsf{new}\ xP'$. In this case the only τ-commitment of $\mathsf{new}\ xP'$ is to $P'[w/x]$, which is shorter than P.
(5) $P = !P'$. If P has a τ-commitment then since $!P' \equiv\ P' \parallel !P'$, we must have that P' has a τ-commitment. Therefore, $T(P')$ by inductive hypothesis. By the commitment law Rep, the conclusion again obtains.

□

Definition 7.7. *For any μ, write $P \downarrow_\mu$ if there is a Q such that $P \xrightarrow{\mu} Q$.*

Lemma 7.8. *For all P and e, if $Fl(p)$, then $\neg P \downarrow_e$.*

Proof. Again by induction on P. The conclusion is certainly true for 0. It holds for sums by virtue of the requirement that no μ_i prefix in a sum can be an e-action. For the case $P_1 \parallel P_2$, we have that $P \downarrow_e$ iff one of $P_i \downarrow_e$. The other cases are straightforward. □

Corollary 7.9. *If $Fl(P)$, then P has no barbed e-experiments.*

Proof. By the two previous lemmas. □

Lemma 7.10. *If P has no barbed e-experiments, then $Fl(P)$.*

Proof. Again this is an induction, but it is best presented as a proof by contradiction. Suppose that P has no barbed e-experiments, but that P cannot flow. We consider the smallest counterexample P(the structural equivalence class of process expressions whose smallest member is a counterexample). Clearly P cannot be 0, because 0 can flow.

(1) If P were a sum, then none of its non-τ arms can be an e. Hence if P cannot flow, then some P_i with $P \xrightarrow{\tau} P_i$ cannot flow. By induction, P_i has a barbed e-experiment, and therefore so does P.
(2) If P is $P_1 \parallel P_2$, then reasoning as in the Sum case shows that P cannot τ-commit to a non-flowing process. Hence if P cannot flow, then one of P_1 or P_2 must not be able to flow. Thus, by induction, one P_i, say P_1, has a barbed e-experiment, and then by the Lpar commitment rule, we construct an e-experiment for P.
(3) If P is $\nu a P_1$ and cannot flow, then P_1 cannot flow by the νa flow rule, so that P_1 has a barbed e-experiment, which passes through to P.
(4) If P is $\mathsf{new}\ xP_1$ and cannot flow, again $P_1[w/x]$ cannot flow for a w not free in P. This gives a barbed experiment for $P_1[w/x]$, and since $P \xrightarrow{\tau} P[w/x]$, we get a barbed experiment for P.
(5) Finally, if P is $!P'$, and cannot flow, we have by the Rep flow rule that P' cannot flow. This gives a barbed experiment for P', and by the structural equivalence $!P' \equiv P' \parallel !P'$ and the Lpar rule, we get a barbed experiment for P.

□

Lemma 7.11. *For any process P, P can flow if and only if P has no barbed e-experiments.*

Proof. By the previous lemma and Corollary 7.9. □

Lemma 7.12. *For any environments E, E' as in Definition 7.5, and any process expression P, $(E,P) \xrightarrow{t} (E',P)$ iff (E,P) has no (embedded) barbed e-experiments.*

Proof. This follows by observation of the proofs of the above lemmas. All of the steps go through in the embedded case; the only place where a τ-commitment might depend on the environment is in the $\mathsf{new}\ xP$ case. But there the argument can be made, since one only has to choose at that stage a fresh variable not free in P nor in the given environment E. □

Theorem 7.13. *If two weakly bisimilar processes are placed in the same environment, the environment fows in exactly the same way: if $P \approx Q$, then for any E, $(E,P) \xrightarrow{t} (E',P)$ if and only if $(E,Q) \xrightarrow{t} (E',Q)$.*

Proof. Assume $P \approx Q$. Suppose for some t, it is not the case that $(E,P) \xrightarrow{t} (E',P)$. We want to prove the same negative statement with P replaced by Q. Using the last lemma, there must be an embedded μ-experiment possible for (E,P), where μ is either an environmental action or an output action. This experiment projects onto a non-embedded μ-experiment for P. Since $P \approx Q$, the same non-embedded experiment is possible for Q. But using Remark 7.3, this gives us an embedded μ-experiment possible for (E,Q), so that the flow $(E,Q) \xrightarrow{t} (E',Q)$ is not possible, by the converse direction of the same lemma.

Finally, the converse implication of the theorem follows by a symmetric argument. □

Remark 7.14. *From this result, we can see that weakly bisimilar processes will evolve over time and over actions in the same way. Once a time-transition for the environment of two bisimilar process has completed, then the processes themselves can match each other's discrete transitions to again bisimilar processes, and this can be repeated.*

8. Representing threaded Petri nets in the ϕ-calculus

Threaded Petri nets [13](TPNs) merge hybrid system dynamics with Petri net concepts. They can be used to specify robotic assembly systems and to prove some useful properties such as deadlock-freeness and liveness. In this section we show an informal example of a TPN and its translation to ϕ-calculus, looking at the "bucket brigade" from Klavins and Koditschek. Formal details of the general translation are in Appendix A.

The bucket brigade consists of three robots arranged in a straight line. The first robot picks up a part from a parts feeder, moves right, passes the part to the second robot, and returns left for another part. The second robot moves right with the part, passes it to the third robot, and returns left for another exchange. The third robot moves right and drops the part in the output bin, then returns to meet the second robot. See Figure 2. In the threaded Petri net representation of this

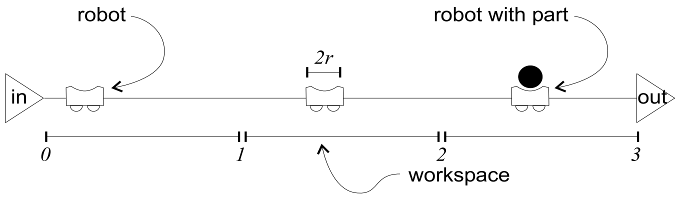


Figure 2. The bucket brigade.

scheme, tokens in a place are names of continuous variables. When a set of variables is in a place, the place evolves continuously according to a predefined controller, one controller for each place. The continuous variables are, in fact, partititioned among (some of) the places. Further, when each transition's input places are occupied by (evolving) variables, its output places are empty. A Liapunov-like firing rule applies: when a transition fires, it redistributes (according to a prespecified function) the variables in its input places bijectively to its output places, which then activate their respective controllers. The condition for firing is that for each output place q which receives a variable from an input place p, the goal predicate for p must entail the domain predicate for q. By referring to Figure 3 the reader can trace the action of the threaded Petri net for our bucket brigade. The gray directed lines in the graph represent the movement of tokens representing the robots and the parts in the brigade. The places (aside from the parts feeder and the output buffer) hold the tokens (continuous variables, links to the environment) we have described.

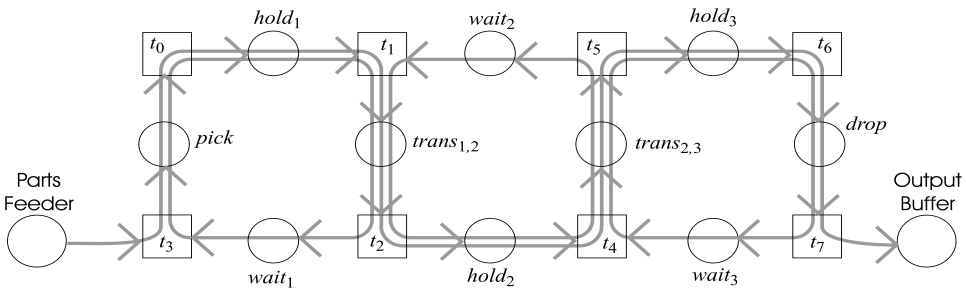


Figure 3. Action of the bucket brigade net.

We focus first on the brigade at the stage when robots 1 and 2 are approaching each other in preparation for the transfer of the part to robot 2. The variables in the places are given by

$$\mathit{pick} : \{\}, \mathit{hold}_1 : \{\}, \mathit{trans}_{12} : \{r_1, p, r_2\}, \mathit{wait}_3 : \{r_3\}, \mathit{hold}_2 : \{\}, \mathit{hold}_3 : \{\}, \mathit{drop} : \{\}.$$

The new configuration (omitting places holding no variables) would be

$$\mathit{wait}_1 : \{r_1\}, \mathit{hold}_2 : \{p, r_2\}, \mathit{wait}_3 : \{r_3\}.$$

We now consider the net firing rule for this stage. For this we look at the place $\mathit{trans}_{1,2}$, the input place to transition t_2. Associated with this place is a controller $f_{\mathit{trans}_{12}}$ which is a gradient system based on the Liapunov *navigation function* scheme of Rimon and Koditschek [18]. The purpose of the controller is to bring the robots into an appropriately "mated" configuration, requiring that they meet at almost zero velocity at the specified goal. The resulting closed loop dynamical system is thus endowed with a goal set $\mathcal{G}_{f_{\mathit{trans}_{12}}}$ which can be thought of as a predicate on the local state, i.e., the values of the continuous variables (p, r_1, r_2). All trajectories in the basin of attraction of this controller must enter the goal set, which is arranged to be a subset of the domain of attraction $\mathcal{D}_{f_{\mathit{wait}_1}}$ of the controller for the place wait_1 intersected with the domain $\mathcal{D}_{f_{\mathit{hold}_2}}$ of the controller for hold_2. Once a trajectory enters the goal set, the transition t_2 must fire, redistributing the variables as indicated in the previous paragraph.

Using the idea of representing transitions of a Petri net by individual ϕ-processes as in Example 4.18, we see immediately that the redistribution of the variables in the net can be modelled by environment variable name-passing between the processes. As an example we sketch the process T_2 representing the transition t_2. Let $\phi_2(p, r_1, r_2), \gamma_2(p, r_1, r_2)$ be the predicates

$$(p, r_1, r_2) \in \mathcal{D}_{f_{\mathit{trans}_{12}}} \setminus \mathcal{G}_{f_{\mathit{trans}_{12}}}, \quad (p, r_1, r_2) \in \mathcal{G}_{f_{\mathit{trans}_{12}}},$$

respectively.

We also introduce process names corresponding to the input and output places of transitions. In this example, they are trans_{12}, wait_1, and hold_2.

The process T_2 is then

$$\begin{aligned} T_2(p, r_1, r_2) \quad ::= \quad & \mathit{trans}_{12}(p, r_1, r_2) \text{ (receive variables from place } \mathit{trans}_{12}) \\ & .[I \doteq \phi_2(p, r_1, r_2); F \doteq ((\dot{p}, \dot{r_1}, \dot{r_2}) = f_{\mathit{trans}_{12}}(p, r_1, r_2))] \\ & \text{(force firing, hence activation later of } T_3 \text{ and } T_4; \\ & \text{activate controller for place } \mathit{trans}_{12}) \\ & .[\gamma_2(p, r_1, r_2)].\overline{\mathit{wait}_1}\langle\, r_1 \,\rangle.\overline{\mathit{hold}_2}\langle\, p, r_2 \,\rangle \text{ (wait until goal reached;} \\ & \text{send variables to } \mathit{hold}_2 \text{ and } \mathit{wait}_1) \\ & .T_2\langle\, p, r_1, r_2 \,\rangle \text{ (and repeat.)} \end{aligned}$$

The process T_1 representing t_1 has to be more complex since it receives variables from the two places wait_2 and hold_1, so has to execute two receive actions in parallel. First, though, for the invariant, we let $\phi_1(p, r_1, r_2)$ be the predicate

$$(p, r_1) \in (\mathcal{D}_{\mathit{hold}_1} \setminus \mathcal{G}_{\mathit{hold}_1}) \vee r_2 \in (\mathcal{D}_{\mathit{wait}_2} \setminus \mathcal{G}_{\mathit{wait}_2}).$$

This represents the fact that we wish to keep running the controllers for wait_2 and hold_1 as long as either one has not attained its goal set. (The Cartesian product of the two goal sets is predetermined to be in the domain of attraction of the controller $f_{\mathit{trans}_{12}}$.) We also let $\gamma_1(p, r_1, r_2)$ be the predicate

$$(p, r_1) \in \mathcal{G}_{\mathit{hold}_1} \wedge r_2 \in \mathcal{G}_{\mathit{wait}_2}.$$

We now need to deal with the problem of simulating the firing rule of transition t_1 in the net. The transition fires when all of its variable tokens are present in both of its input places, and the appropriate goal condition is satisfied. We simulate token-passing by name-passing during reaction, so we have to treat the problem of exactly when T_0 and T_5 send their variables to $hold_1$, and $wait_2$ and not to let T_1 fire when not all of its input variables have been received. This requires us to anticipate a sequence of input actions which may occur in any order. In the example of T_1, we need to anticipate that the action of receiving a pair of variables from $hold_1$ may or may not precede the action of getting a variable from $wait_2$.

One solution to this problem is to introduce Boolean variables corresponding to each place in the net. Such variables can be realized using continuous variables having zero derivative, and which are initialized and reset to 0 or 1. Let us call these variables $Bhold_1$, $Bwait_2$, and $Btrans_{12}$ respectively. To delay the firing of T_1 we introduce the test

$$(Bhold_1 = 1) \wedge (Bwait_2 = 1).$$

Officially, this is an environmental action with no body.

With this notation, we may describe the process corresponding to transition t_1.

$$\begin{array}{lll}
T_1(p, r_1, r_2) & ::= & [(Bhold_1 = 1) \wedge (Bwait_2 = 1)] \\
 & & .hold_1(p, r_1).wait_2(r_2) \\
 & & \quad \text{(receive variables from places } hold_1 \text{ and } wait_2\text{)} \\
 & .(& [I \doteq \phi_1(p, r_1, r_2); \\
 & & F \doteq ((\dot{p}, \dot{r_1}) = f_{hold_1}(p, r_1); \dot{r_2} = f_{wait_2}(r_2))] \\
 & & \text{(force firing, hence activation later of } T_2\text{;} \\
 & & \text{activate controllers for places } hold_1 \text{ and } wait_2\text{)} \\
 & & .[\gamma_1(p, r_1, r_2)].\overline{trans_{12}}\langle p, r_1, r_2 \rangle \ \ \text{(wait for goals; send variables to } trans_{12}\text{)} \\
 & & .[Bhold_1 := 0].[Bwait_2 := 0].[Btrans_{12} := 1] \ \ \text{(reset Booleans)} \\
 & & .T_1\langle p, r_1, r_2 \rangle \text{ (and repeat.)} \\
 &) &
\end{array}$$

The code for the process T_2 should now be rewritten in this same way.

Finally we model the parts feeder using recursion and the new operator. The feeder operates by proximity; every time robot 1 is within some small ϵ of 0, the feeder produces a new part.

We call this recursive process PF. It is convenient to think as well of the "parts feeder"" in Figure 3 as an actual place with name pf.

$$PF ::= \left[\ I \doteq (r_1 \geq \epsilon)\ \right].\mathsf{new}\ part(\left[\begin{array}{c} part \doteq 0 \\ \dot{part} \doteq 0 \end{array}\right].[Bpf := 1].\overline{pf}\langle part \rangle.PF).$$

The invariant I in this process is violated when the first robot gets within ϵ distance of the parts feeder. This triggers a call of the localizing operator new *part*. This in turn creates a fresh part name, using alpha-conversion to avoid clashes with part names in the environment when the "new" process "decays" at the *Env* commitment. Parts start with 0 velocity at position 0. The name of the part is passed to the place pf which is an input place to transition t_3.

The Petri net representing the whole bucket brigade is then represented by the parallel composition of all the processes representing the transitions, the parts feeder, and the parts drop.

9. Conclusion

We hope to have demonstrated in this report the feasibility of using process-algebraic techniques in the design of hybrid systems. However, much theoretical work remains to be done.

First, we need to have a deeper understanding of simulations and bisimulations. An important direction is studying the analogue of bisimulations for environments; in the literature these are called *topological conjugacies.* We would like to integrate these into a combined definition of bisimulation. It would be nice to give conditions on environments and environmental actions (guarantees that flows will eventually reach the boundary of an invariant, for example), so that a "time-abstract" version of a process could be proven live (deadlock-free) by passing to its discrete version using simulation. This (implicitly) is already the case in Klavins and Koditschek's threaded Petri nets. One can pass to the underlying net structure to prove liveness, because of the backchaining conditions assumed on the controllers in the net. (See Appendix A.)

Next, we would like to have a version of the ϕ-calculus which automatically is free from Zeno executions (infinite sequences of discrete transitions in bounded total time). Here we could study the *receptivity* property introduced in [15].

In this preliminary document we have said nothing about possible logics for reasoning about ϕ-calculus processes. A possibility is to extend the work by Mads Dam [4] which reasons about π-calculus in a μ-calculus (!). Such an extension would incorporate assertions about the continuous state.

On the control theory side, we would like to see what kinds of systems can be modeled with "superposition" environmental actions. Much work on coupled oscillators uses superposition to combine individual oscillators, and it is possible that one could smooth out hybrid jumps using this kind of reset. Also, we would like to investigate extensions of backchaining so that goal sets could be parts of limit cycles determined by Poincaré sections, for example. This work should be important in the further study of robotic assembly problems.

References

[1] R. Alur, R. Grosu, I. Lee, and O. Sokolsky. Compositional refinement for hierarchical hybrid systems. In Maria Domenica Di Benedetto and Alberto Sangiovanni-Vincentelli, editors, *Hybrid Systems, Computation, and Control*, pages 33–48. Springer-Verlag, 2001. Volume 2034 of Lecture Notes in Computer Science.

[2] R. Alur and. T. A. Henzinger. Reactive modules. *Formal methods in System Design*, 15:7–48, 1999.

[3] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *J. Assoc. Comput. Mach.*, 31(3):560–599, 1984.

[4] Mads Dam. Model-checking mobile processes. In *Springer LNCS 715*, pages 22–36, 1993.

[5] Rene David and Hassane Alla. On hybrid petri nets. *Discrete Event Dynamic Systems*, 11:9–40, 2001. Kluwer.

[6] Stephen J. Garland and Nancy A. Lynch. Using i/o automata for developing distributed systems. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, pages 285–312. Cambridge University Press, 2000.

[7] C. Ghezzi, P. Mandrioli, S. Morasca, and P. Mauro. A general way to put time into petri nets. In *Proceedings of 5th International Workshop on Software Specification*, pages 60–67, 1989.

[8] V. Gupta, R. Jagadeesan, and V. A. Saraswat. Computing with continuous change. *Science of Computer Programming*, 30:3–49, 1998.

[9] T. A. Henzinger. The theory of hybrid automata. In *Proc. 11th Annual Symposium on Logic in Computer Science*, pages 278–292, 1996.

[10] T. A. Henzinger, M. Minea, and V. Prabhu. Assume-guarantee reasoning for hierarchical hybrid systems. In Maria Domenica Di Benedetto and Alberto Sangiovanni-Vincentelli, editors, *Hybrid Systems, Computation, and Control*, pages 275–290. Springer-Verlag, 2001. Volume 2034 of Lecture Notes in Computer Science.

[11] He Jifeng. From csp to hybrid systems. In *A Classical Mind, Essays in Honour of C. A. R. Hoare*, pages 171–189. Prentice-Hall international, 1994.

[12] O. Khatib. Real time obstacle avoidance for manipulators and mobile robots. *International Journal of Robotics Research*, 5:90–99, 1986.

[13] Eric Klavins and Daniel Koditschek. A formalism for the composition of loosely coupled robot behaviors. Technical report, University of Michigan CSE Tech Report CSE-TR-12-99, 1999.

[14] Jana Kosecka. *A Framework for Modeling and Verifying Visually Guided Agents:Design, Analysis, and Experiments.* PhD thesis, University of Pennsylvania, 1996.

[15] Nancy Lynch, Roberto Segala, and Frits Vaandrager. Hybrid i/o automata revisited. In Maria Domenica Di Benedetto and Alberto Sangiovanni-Vincentelli, editors, *Hybrid Systems: Computation and Control. Fourth International Workshop*, pages 403–417. Springer-Verlag, 2001. Volume 2034 of Lecture Notes in Computer Science.

[16] Nancy Lynch and Mark Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of 6th Annual Symposium on Principles of Distributed Computing*, pages 137–151, 1987.

[17] Robin Milner. *Communicating and Mobile Systems: the π-calculus.* Cambridge University Press, 1999.

[18] Elon Rimon and D. E. Koditschek. Exact robot navigation using artificial potential fields. *IEEE Transactions on Robotics and Automation*, 8(5):501–518, Oct 1992.

[19] Steve Schneider. *Concurrent and real-time systems: the CSP approach.* Wiley, 2000.

[20] Roberto Segala. A process-algebraic view of i-o automata. Technical report, MIT Tech Memo MIT/LCS/TR557, 1992.

Appendix A. Formal translation of TPNs

For reference, we present in this appendix a summary of threaded Petri net definitions and a review of the control-theoretic ideas used in the constructions of goal sets and domains of attraction for the various controllers exemplified above. It will then be clear how the translation of TPNs to the ϕ-calculus can be carried out in general.

We begin with generalized marked graphs. These are Petri nets with finite sets P of places and T of transitions. We denote by ${}^\bullet t$ and $t^\bullet$ the set of input and output places respectively for transition t. In a marked graph, each place is an input place to at most one transition and an output place for at most one transition. Let $p^\bullet$ be the unique, if any, transition for which p is the input place. Similarly ${}^\bullet p$ is the unique transition feeding p.

Next, there is a finite set S of *slots* $\{s_1, \dots, s_n\}$; We have a function $\alpha : P \to \mathcal{P}(S)$, called the *slot distribution function*, which is such that (i) for each transition t, $\alpha(p) \cap \alpha(q) = \emptyset$ for all $p \neq q$ in ${}^\bullet t$; (ii) also for all $p \neq q$ in $t^\bullet$; and (iii) for each transition t,

$$\bigcup_{p \in {}^\bullet t} \alpha(p) = \bigcup_{p \in t^\bullet} \alpha(p).$$

In the bucket brigade, we have shown neither the distribution function nor the set S of slots. We might put $S = \{\mathit{fr}, \mathit{sr}, \mathit{tr}, \mathit{part}\}$, standing for first robot, second robot, third robot, and part respectively. Then, for example, $\alpha(\mathit{trans}_{12}) = \{fr, tr, part\}$. The slots can be thought of as the net equivalent of ϕ-calculus abstractions.

The distribution function has the property that if a slot s is in $\alpha(p)$ for p an input place of t, then there is a unique place q in $t^\bullet$ with s in $\alpha(q)$. Thus we have implicitly defined a (partial) function $\delta : S \times P \to P$, called the *redistribution function* of the net. (The transitions do not enter this definition, because any place is input to at most one transition; the function $\delta(p, s)$ is undefined if $p^\bullet$ is not defined.) By iterating the function $\delta(s, \cdot)$ for a fixed slot s, starting at a given place p, we get a finite or infinite sequence of places which is called a *thread* for the net starting at p. A *full thread* starts at a place p for which ${}^\bullet p$ is not defined, and is either infinite or terminates at a place p for which $p^\bullet$ is not defined. Following a gray line in Figure 3 and enumerating the places passed gives a thread.

We assume prespecified a set $\mathcal{V}$ of variables for the net. In the bucket brigade, $\mathcal{V} = \{p_1, p_2, \dots\} \cup \{r_1, r_2, r_3\}$. A *marking* for such a is a pair (m, ρ_m), where m is a set of places (the marked places) and a bijective map $\rho_m : \bigcup_{p \in m}(\{p\} \times \alpha(p)) \to \mathcal{V}$. For each slot s of a place p in m, $\rho(p, s)$ is the variable in that slot. The presence of a variable in a slot corresponds in the ϕ-translation to having carried out a commitment via a reaction which has passed environment variable names. In our example, the marking represented just before trans_{12} fires is $m = \{\mathit{trans}_{12}, \mathit{wait}_3\}$, and ρ_m is

$$\{(\mathit{trans}_{12}, fr) \mapsto r_1;\ (\mathit{trans}_{12}, sr) \mapsto r_2;\ (\mathit{trans}_{12}, part) \mapsto p;\ (\mathit{wait}_3, tr) \mapsto r_3\}.$$

The continuous dynamics of a net is given locally. For each place p there is assumed a differential equation $\dot{x} = F_p(x)$, $x \in \mathbb{R}^{\alpha(p)}$. Here we use the slot names as abstract variables, corresponding to the fact that the differential equation which is actually in force when a place is operating under a specific mode dynamics may vary according as to which actual variables occupy the slots in a place. We assume that the solution $f(x, t)$ of this equation has a stable fixed point (equilibrium point) x^*, where of course $F_p(x^*) = 0$. We call x^* the *goal*. The *goal set* of p is a (small) open set $\mathcal{G}_p$ containing the goal, and such that $(\forall t)(f(y, t) \in \mathcal{G}_p)$ for all $y \in \mathcal{G}_p$. The largest open set $\mathcal{D}_p$ with the same property is called the *domain of attraction* for p. The collection $\{F_p \mid p \in P\}$ is called a *palette of controllers* for the net.

The palette of controllers satisfies a *backchaining* condition, which we now explain. Roughly, the goal set of one controller in an input place to a transition is supposed to be a subset of the domain of a controller of an output place for that transition. Because a transition can have several input and output places, this condition is more complicated. Let t be any transition. Because we have a marked graph, the transition t is the unique one feeding any of its output places. If q is an output place of t, denote by $\delta^{-1}[\alpha(q)]$ the set of slots s in any input places p to t such that $\delta(s,p) \in \alpha(q)$. The backchaining condition for the net is then

$$(\forall q \in t^{\bullet}) : \pi_{\delta^{-1}[\alpha(q)]}(\mathcal{G}_{p_1} \times \cdots \times \mathcal{G}_{p_n}) \subseteq \mathcal{D}_q$$

where π_A is the projection of a set in $\mathbb{R}^S$ onto $\mathbb{R}^A$ for $A \subseteq S$, and $\{p_1, \ldots, p_n\} =^{\bullet} t$.

Let $x \in \mathbb{R}^{\mathcal{V}}$ be a "state" of the net. A transition t is *enabled* with respect to marking (m, ρ_m) and x, if

(1) ${}^{\bullet}t \subseteq m$ and $t^{\bullet} \cap m = \emptyset$;
(2) $(x_{\rho_m(p,s_1)}, \ldots, x_{\rho_m(p,s_n)}) \in \mathcal{G}_p$ for every input place p to t, where $\{s_1, \ldots, s_n\} = \alpha(p)$.

The *follower marking* for an enabled transition is the marking (n, ρ_n), where $n = (m \setminus^{\bullet} t) \cup t^{\bullet}$, and

$$\rho_n(r,s) = \begin{cases} \rho_m(r,s) \text{ for } r \notin t^{\bullet}; \\ \rho_n(r,s) = \rho_m(p,s) \text{ if } r \in t^{\bullet} \text{ and } \delta(p,s) = r. \end{cases}$$

With this review (and slight reformulation) of TPNs, we can present our formal translation of TPNs into the ϕ-calculus. We proceed on a transition-by-transition basis. Let t be a transition with ${}^{\bullet}t = \{p_1, \ldots, p_n\}$, $t^{\bullet} = \{q_1, \ldots, q_m\}$ and overall slot set $\{s_1, \ldots, s_l\} = \bigcup_{p \in {}^{\bullet}t} \alpha(p)$. Also by $t(\alpha(p))$ we mean an abstraction with name t over the environmental variables in $\alpha(p)$, using slot names as the bound variables of the abstraction. Let $\xi_t(s_1, \ldots, s_l), \gamma_t(s_1, \ldots, s_l)$ be the predicates

$$\bigvee_{p \in {}^{\bullet}t} (\alpha(p) \in \mathcal{D}_p \setminus \mathcal{G}_p), \quad \bigwedge_{p \in {}^{\bullet}t} \alpha(p) \in \mathcal{G}_p,$$

respectively.

As in the examples of the last section, we introduce formal place names $p_1, \ldots, p_n$, and the corresponding Boolean variables $Bp_1, \ldots, Bp_n$.

For the transition t we define a process $T_t(s_1, \ldots, s_l)$ as follows:

$T_t(s_1, \ldots, s_l) \quad ::=$

$[\bigwedge_{p \in {}^{\bullet}t} (Bp = 1)]$

$.p_1(\alpha(p_1)). \ldots .p_n(\alpha(p_n))$

(receive variables from all transitions feeding t)

$.($ $[I \doteq \xi(s_1, \ldots, s_l);\ F \doteq ((\dot{\alpha(p_1)} = f_{p_1}(\alpha(p_1); \ldots; \dot{\alpha(p_n)} = f_{p_n}(\alpha(p_n))]$

(force firing, hence activation later of controllers for output places of t;

activate controllers for input places)

$.[\gamma_t(s_1, \ldots, s_l)].\overline{q_1}\langle\, \alpha(q_1)\,\rangle. \ldots .\overline{q_m}\langle\, \alpha(q_m)\,\rangle$ (wait for goals; send variables to follower transitions)

$.[Bp_1 := 0]. \ldots .[Bp_n := 0].[Bq_1 := 1]. \ldots .[Bq_m := 1]$

$.T_t\langle\, s_1, \ldots, s_l\,\rangle$ (and repeat.)

$)$

As in the bucket brigade, we just put all the transitions T_t in parallel combination to get the ϕ-calculus process representing the net.

A word is in order about the adequacy of our translation. We should be concerned about preserving liveness and safety properties from the original TPN into the translated ϕ-calculus version. We notice that establishing these properties, in the work of Klavins and Koditschek, is reduced to consideration of the discrete version of the net, without the continuous dynamics. This discrete version is simply a Petri net which accumulates variable names in its places, and fires when the appropriate variable names for a given transition have been accumulated into the input places for that transition. The reason that this reduction is possible is the backchaining condition; a guarantee is provided that the goal set of a controller will be contained in the domain set of a follower controller.

It is easy to see that a corresponding translation of the discrete version of a TPN into ϕ-calculus will preserve liveness and safety properties of the original net. To do this, one can establish a kind of bisimulation property in which the firing of a transition in the net corresponds to an interaction step of the translated process in which the tokens on the input places are transmitted to the output places via name-passing.

There is, however, a minor problem with the TPN formalism itself. Once the goal predicate of a controller has been attained, there is nothing in the TPN formalism which forces a transition to actually fire. In contrast, this firing is forced in our translated version via the use of an appropriate invariant.

Finally, the parts feeder in Klavins and Koditschek's bucket brigade example is an anomalous kind of "place" in which tokens can magically be created. It does not actually conform to the TPN formalism. Our tratment of the parts feeder, by contrast, uses recursion and the new operator to generate this stream of part numbers.

CSE Division, EECS Department, University of Michigan, Ann Arbor, MI 49109, USA
E-mail address: {rounds, hosungs}@eecs.umich.edu