Computer Science at Kent

Mobile Processes in Unifying Theories

Xinbei Tang

Technical Report No. 1-04
January 2004

Published by the Computing Laboratory,
University of Kent, Canterbury, Kent, CT2 7NF, UK

Mobile Processes in Unifying Theories

Xinbei Tang

Computing Laboratory
University of Kent
Canterbury, Kent, CT2 7NF, UK

xt2@kent.ac.uk

January 2004

Abstract

This report presents the initial work in the development of a theory of mobile processes in *Circus*, a language for describing state-based reactive systems. The mathematical basis for the work is Hoare and He's *Unifying Theories of Programming (UTP)*, where the alphabetised relational calculus is used to provide a common framework for the semantics and refinement calculus of different programming paradigms.

As our first step, we study the denotational semantics of mobile processes in UTP. Process mobility is interpreted as the assignment or communication of higher-order variables, whose values are process constants or parameterised processes, in which the target variables update their values and the source variables lose their values. We then present a set of algebraic and refinement to be used for the development of mobile systems. The correctness of these laws can be ensured by the UTP semantics of mobile processes.

We illustrate our theory through a simple example that can be implemented in both a centralised and a distributed way. First, we present the π-calculus specification for both systems and demonstrate that they are observationally equivalent. Next, we show how the centralised system may be derived into the distributed one using our proposed laws.

By formalising mobile processes and studying their refinement laws in the same semantic framework of *Circus*, they can be included in *Circus*' refinement calculus to enhance *Circus* with the ability to develop networks of mobile processes.

Keywords. Mobile process, refinement, UTP, higher-order programming

Contents

1 Introduction

Mobile processes are becoming increasingly popular in software applications; they can roam from host to host in a network, carrying their own state and program code. A common example is a Java applet that is downloaded from a server across the Internet, and then executed in a client. Such processes bring new programming problems, and so require their own theoretical foundations for the correct and rigorous development of applications.

The most well-known formal theory for mobile processes is the π-calculus [18, 20, 27]. Extended from Milner's CCS [19], the π-calculus is a process algebra that provides a conceptual framework for understanding mobility, and mathematical tools for expressing mobile systems and reasoning about their behaviours [27]. In the π-calculus, it is links that move, whose simple idea is to allow processes to exchange channel names, thus change their interconnection structures dynamically. The higher-order π-calculus (HOπ in short) [26] treats mobility by process passing. In the π-calculus and the HOπ, the mobile and dynamic features of mobile systems can be modelled.

The semantics of the π-calculus and the HOπ are typically of an operational style, which illustrates how a program can be executed by a series of steps that change the program state. It provides an equivalence relation comparing processes using bisimulation, which is defined on the structure of the behavioural graph of the two processes, where the arrows of the graph are the transitions in which the process may participate. With the operational semantics, processes are close to implementation, thus their translation to a programming language is relatively easy. However, they cannot describe system behaviour abstractly and comprehend the inner meaning of a program. The operational semantics is difficult to be used for proof, refinement and system development. Moreover, the π-calculus cannot be used to model divergence, in that it principally adopts weak barbed congruence [27] as the behavioural equivalence for π-terms, in which the internal τ action of a system is ignored.

We propose a theory of mobile processes that is suitable for refinement, so that it can support stepwise development of mobile systems. Especially, starting with an abstract specification, we develop a distributed system by using mobile processes, in which each step of the derivation is provable in the theory. A denotational understanding of mobility is essential for this purpose. Based on the denotational semantics, a refinement ordering relation, which holds between an implementation and a specification, can be defined. A crucial result in this is that we are able to deal with the specification and the implementation in the same mathematical framework.

There are also some researches in which the denotational settings of the π-calculus and HOπ are studied [11, 6, 30]. These denotational approaches all involve a quite complicated type theory, domain theory or category theory, and no refinement relation based on these semantics is studied.

In this report, we present our initial results, which are denotational semantics and a set of algebraic laws for mobile processes. The mathematical basis for the work is Hoare and He's *Unifying Theories of Programming (UTP)* [13], which

uses a simple alphabetised form of Tarksi's relational calculus. The correctness of the laws can be guaranteed using the UTP semantics for mobile processes. We study a simple example in both π-calculus and our theory. The results show the suitability of our laws for the stepwise development of a distributed system from a centralised specification, rather than comparing processes by using bisimulation in the π-calculus.

Process mobility is exhibited in the higher-order variable assignment or communication, in which both the source and the target are variables, and have process behaviours as their values. When a higher-order mobile-variable assignment or communication takes place, the target variable is updated, more or less as one would expect; but at the same time, the source variable becomes undefined. In this way, processes are moved around in the system.

The presence of process-valued variables has an impact on monotonicity with respect to refinement, and we require that process assignment should be monotonic in the assigned value. This follows the treatment of higher-order programming in [13].

The remainder of this report is organised as follows. The next section gives a brief overview of unifying theories and explains the reasons why we select it as the theoretical basis. Section 3 gives a simple example implemented in both a centralised and a distributed way, and shows that they are observationally equivalent using the π-calculus. Thereafter, we present the syntax, the UTP semantics, and a set of laws for mobile processes in Section 4, Section 5 and Section 6 respectively. In Section 7, we illustrate our development method of distributed system through some refinement laws; thereafter, we apply the laws in the example in Section 8. We conclude the presented work in Section 9 and outline some future work.

2 The Unifying Theories of Programming

Hoare and He's Unifying Theories of Programming (UTP) [13] is a mathematical theory for programming in which the alphabetised relation calculus and the predicate calculus are adopted as the fundamental basis for formalising and linking various theories of programming across three dimensions: different computational paradigms ([13] section 0.1), different levels of abstraction ([13] section 0.2) and distinct mathematical representations ([13] section 0.3). For each programming paradigm, programs, designs and specifications are all interpreted as relations between an initial observation and a single subsequent (intermediate or final) observation of the behaviour of the execution of a program. Program correctness and refinement can be represented by inclusion of relations. All the laws of relational calculus are valid for reasoning about correctness in all theories and in all languages.

Within the framework of the general theory of relations, formal theories differentiate one from another by their alphabet, signature, and healthiness conditions.

The *alphabets* ([13] section 0.4) of a theory are just a set of variables to record

a range of external observations of program behaviour that are considered relevant. The variable of an initial observation before the program is started is undecorated, but the variable to record the similar intermediate or final observation taken subsequently is decorated with a dash. For instance, variable x may stand for the initial value of a global variable updated by the programs, and x' denotes the final value of that variable in an intermediate state or on termination. Except the global program variables, there are some other variables globally shared with the environment in which the program involves. The first example of these external variables is the boolean variable *ok* whose value *true* means that the system has started in a stable state. ok' takes the value *true* when the system has reached a stable and therefore observable state; this permits a description of programs that fail due to nonterminating loops or recursion. In a theory of reactive processes, *tr* records the cumulative interactions between a process and its environment; and the boolean variable *wait* distinguishes the intermediate observations of waiting states from final observations of termination. During a *wait*, the process can refuse to take part in certain events offered by the environment, which are specified by the variable *ref*.

The *signatures* ([13] section 0.5) of a theory are a set of operators and atomic components which provide syntax for denoting the objects of the theory. It gives the way to represent the elements of the theory by taking primitives directly as elements and using operations to construct them into larger expressions and programs. The signature of a programming theory at the lowest level of abstraction must obviously include all the notations and operations for the target programming language. Design successively removes unimplementable operators; all operators are monotonic. In the programming language, only implementable operations are left.

The *healthiness conditions* ([13] section 0.6) help designers select the required elements for a sub-theory from those of a more expressive theory in which it is embedded. By a suitable restriction of signature, design languages satisfy many healthiness conditions, and eventual target programming languages satisfy more. Thus in this elegant framework, in a top-down design process, programs form a subset of intermediate designs, and designs form a subset of specifications.

UTP has been successfully used as the theoretical foundations of the semantics and refinement calculi of various models in different areas of computing. Its power to construct a formal semantics and system development by design refinement are expressed thoroughly in these applications. In [8, 10], He, Liu and Li presented an observation-oriented semantics for an object-oriented language with a rich variety of features, including subtype, visibility, inheritance, dynamic binding, mutual recursive methods and recursions. In this model both class declarations and commands are identified as UTP designs. UTP relational model is also used for specifying and reasoning in activities of an object system development process in [9] where a refinement calculus for object systems is proposed. Li and He [15] provided a denotational semantics to a subset of Time RAISE [7] Specification Langauge (TRSL) using Duration Calculus [37] model, borrowing a lot of ideas from UTP. Based on this work, recently Ri and He [23] presented a technique for verifying the correctness of TRSL specification against

the real-time requirements described as predicates over the observables; a set of proof rules are developed and their soundness and completeness is proved. This work follows again the idea and methodology of UTP. In [32] Woodcock developed a formal semantics and refinement calculus for a shared-variable parallel programming language using the concepts and notations of UTP. In [33] Woodcock and Cavalcanti used UTP to formally define the semantics of a concurrent refinement langauge *Circus* that integrates CSP [12, 24], Z [36], specification statements [21] and guarded commands [5], for describing state-based reactive systems. With the same UTP semantics model, Sherif and He naturally extended *Circus* with discrete time in [29] and proposed in [28] a framework for the specification, validation and development of real time program using *Circus*. In *Circus* and timed *Circus*, UTP is demonstrated to be a powerful theory to unify different models. Similar work was done later by Qin, Dong and Chin in [22] by constructing a UTP semantics model for another integrated langauge — Timed Communicating Object Z (TCOZ) [17]. More recently, Jin and He proposed a hierarchy of four semantics models for specifying and reasoning programs with limited resources [14] following the design methodology in UTP. Each model in the hierarchy is a refinement of the one defined formerly.

As a powerful theoretical foundation for the denotational semantics and refinement calculi of so many different models, we believe UTP can also be used for our formalisms and development method for mobile processes. The further reasons why we select it as the theoretical basis are:

- UTP serves as the mathematical base of *Circus* [34]. We intend to include the refinement of mobile processes in the refinement calculus of *Circus*. It is adequate to use UTP — the same semantics framework of *Circus* — to study mobile processes, otherwise this inclusion will be very difficult or impossible.

- The approaches to communication and higher-order programming in UTP ([13] chapter 8 and 9) are applicable to mobile processes. Communication is one of the important features of mobile processes. Moreover, as processes themselves can be communicated through channels, we need to formalise mobile processes in higher-order programming.

3 A Simple Example in the π-Calculus

Suppose that there is a data centre that needs to analyse data based on information residing in different hosts on a network. For simplicity, we assume that this analysis amounts to getting the sum of the data in each host.

This small application can be implemented in two ways. In the first implementation (Figure 1), the data center directly communicates with each host, one by one, getting the local information and then updating the data. All the calculations for update are carried out in the data centre. This implementation is very simple, and so is obviously correct.

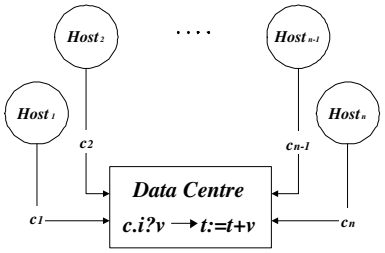

Figure 1: Centralised Implementation

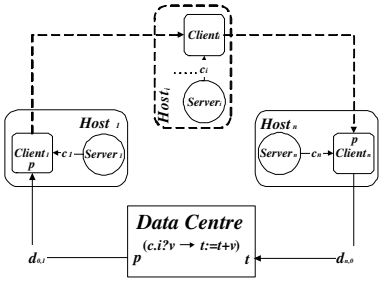

Figure 2: Distributed Implementation

In the second implementation (Figure 2), similar pieces of specification are abstracted and encapsulated in a parameterised process variable, which travels from the data centre, and roams the network to each host, taking its local state, and the operations used to update the local state. After its arrival at a host, it is plugged into a local channel c_i and activated, getting the local information and updating the local state. After visiting all hosts, it comes back to the data centre with the final result.

Both implementations can complete the simple task, but the latter one has a richer structure that might make it more useful. For example, if the nature of the analysis changes, then only the mobile process needs to be changed. Consider an international bank: it may change both data (exchange rates) and function (services offered to customers) frequently. If this is programmed as a mobile process, then the very latest upgrade arrives automatically in local branches. It may also be more efficient to move the process from the centre to each host, as local communication would replace remote communication.

Both systems can be specified in the π-calculus. As the π-calculus syntax does not have assignment, we use a suitable translation to convert assignment into π-terms as the composition of an input and an output action over a restricted (or bound) name. The π-calculus syntax that we adopt is from [20].

Definition 3.1 (π-calculus assignment)

$$[\![(t := e).P]\!] \;\widehat{=}\; \mathsf{new}\ h\ (\overline{h}\langle e\rangle.\mathsf{0} \mid h(t).[\![P]\!]) \qquad \{t \in \mathsf{fn}(P), h \notin \mathsf{fn}(P)\}$$

where the part enclosed in [ ] is the side condition that the definition should satisfy, and $\mathsf{fn}(P)$ *denotes the free names of* P. □

Observationally, the effect of the assignment of a value to variable t followed by executing P immediately is the same as substituting this value for all the free occurrences of the variable t in P. This is shown by the following simple theorem that states the bisimilarity between the two.

Theorem 3.1 (Assignment bisimulation)

$$(t := e).P \;\approx\; \{e/t\}P \qquad \{t \in \mathsf{fn}(P)\}$$

where $\{e/t\}P$ *denotes the systematic substitution of* e *for* t *in* P.

***Proof*:**

$$
\begin{aligned}
& LHS && \{\text{definition of assignment}\}\\
&= \mathsf{new}\ h\ (\overline{h}\langle e\rangle.\mathsf{0} \mid h(t).P) && \{\text{strong bisimilarity}\}\\
&\sim \tau.(\mathsf{0} \mid \{e/t\}P) && \{\text{weak bisimilarity}\}\\
&\approx \mathsf{0} \mid \{e/t\}P && \{\text{structural congruence}\}\\
&\equiv RHS
\end{aligned}
$$

The centralised system is the composition of a *Centre* process and n *Host*s over restricted channel names.

$$System \ \widehat{=}\ (\mathsf{new}\ c_1, c_2, \cdots, c_n)(Centre \mid Host_1 \mid Host_2 \mid \cdots \mid Host_n)$$

where the *Centre* and $Host_i$ are defined as:

$$
\begin{aligned}
Centre &\ \widehat{=}\ (t := 0).c_1(v).(t := t + v).\cdots.c_n(v).(t := t + v).result\langle t\rangle\\
Host_i &\ \widehat{=}\ \overline{c_i}\langle u_i\rangle
\end{aligned}
$$

where $c_i(v)$ is the input of the data that needs to be retrieved from the ith host, and u_i is the data that needs to be retrieved from the ith host. We sometimes abbreviate $result\langle t\rangle.\mathsf{0}$ as $result\langle t\rangle$.

In the assignment $t := t + v$ of *Centre*, the t in the left hand side is a restricted name while the t in the right hand side is a free name. We can better comprehend the scope of each t and v using renaming. The effect of *Centre* is the same as

$$
\begin{aligned}
&Centre\ \widehat{=}\\
&\quad (t_0 := 0).c_1(v_1).(t_1 := t_0 + v_1).\cdots.c_n(v_n).(t_n := t_{n-1} + v).result\langle t_n\rangle
\end{aligned}
$$

Observationally, *System* is equivalent to $result\langle u_1 + u_2 + \cdots + u_n\rangle$.

Theorem 3.2 (Centralised system bisimulation)

$$System \approx result\langle u_1 + u_2 + \cdots + u_n\rangle$$

***Proof*:**

$$
\begin{aligned}
& LHS && \{\text{definition}\}\\
&= (\mathsf{new}\ c_1, c_2, \cdots, c_n)\\
&\qquad \left(\begin{array}{l} ((t := 0).c_1(v_1).(t_1 := t + v_1).\cdots.c_n(v_n).(t_n := t_{n-1} + v_n).result\langle t_n\rangle)\\ \mid \overline{c_1}\langle u_1\rangle \mid \overline{c_2}\langle u_2\rangle \mid \cdots \mid \overline{c_n}\langle u_n\rangle \end{array}\right)\\
& && \{\text{structural congruence}\}\\
&\equiv (t := 0).(\mathsf{new}\ c_1)(c_1(v_1) \mid \overline{c_1}\langle u_1\rangle).(t_1 := t + v_1).\cdots.\\
&\qquad (\mathsf{new}\ c_n)(c_n(v_n) \mid \overline{c_n}\langle u_n\rangle).(t_n := t_{n-1} + v_n).result\langle t_n\rangle && \{\text{strong bisimilarity}\}\\
&\sim^* (t := 0).\tau.(t_1 := t + u_1).\cdots.\\
&\qquad (\mathsf{new}\ c_n)(c_n(v_n) \mid \overline{c_n}\langle u_n\rangle).(t_n := t_{n-1} + v_n).result\langle t_n\rangle
\end{aligned}
$$

$$
\begin{array}{ll}
 & \{\text{definition of } t := 0, \text{ Theorem 3.1, strong bisimilarity}\} \\
\sim \tau.\tau.(t_1 := u_1).\tau.(t_2 := t_1 + u_2).\cdots. & \\
\quad (\mathsf{new}\ c_n)(c_n(v_n) \mid \overline{c_n}\langle u_n\rangle).(t_n := t_{n-1} + v_n).\mathit{result}\langle t_n\rangle & \{\text{weak bisimilarity}\} \\
\approx t_1 := u_1.\tau.t_2 := t_1 + u_2.\cdots. & \\
\quad (\mathsf{new}\ c_n)(c_n(v_n) \mid \overline{c_n}\langle u_n\rangle).(t_n := t_{n-1} + v_n).\mathit{result}\langle t_n\rangle & \\
 & \{\text{induction over the indices as from step}^*\} \\
\approx \tau.\mathit{result}\langle u_1 + u_2 + \cdots + u_n\rangle & \\
\approx \mathit{RHS} & \{\text{weak bisimilarity}\}
\end{array}
$$

□

In the HOπ [26] and polyadic π-calculus [20], abstractions (parameterised processes) or multiple names can be transmitted directly along channel names. In the distributed implementation, an abstraction $(z, in, out, dn).P$ and the value of t are transmitted at the same time, where $(z, in, out, dn).P$ and its execution $((z, in, out, dn).P)\langle c_i, t_{i-1}, t_i, tg\rangle$ are defined as follows:

$$
\begin{array}{rl}
(z, in, out, dn).P & \widehat{=}\ z(v).(out := in + v).\overline{dn}\langle out\rangle.\mathsf{0} \\
((z, in, out, dn).P)\langle c_i, t_{i-1}, tmp, tg\rangle & \widehat{=}\ c_i(v).(tmp := t_{i-1} + v).\overline{tg}\langle tmp\rangle.\mathsf{0}
\end{array}
$$

Let $d_{0,1}$ be the name of the channel connecting *Centre* and $Host_1$, $d_{n,0}$ be the name of the channel connecting $Host_n$ and *Centre*, and $d_{i,i+1} (1 \leq i \leq n-1)$ be the name of the channel connecting $Host_i$ and $Host_{i+1}$. The specification for the distributed system is

$$
\begin{array}{l}
\mathit{MSystem}\ \widehat{=} \\
\quad (\mathsf{new}\ d_{0,1}, d_{1,2}, \cdots, d_{n,0})(\mathit{MCentre} \mid \mathit{MHost}_1 \mid \mathit{MHost}_2 \mid \cdots \mid \mathit{MHost}_n)
\end{array}
$$

The *Centre* now has the task of initialising its total, sending the mobile process and total on its way, and then waiting for the total to return home[1], before outputting the result.

$$
\mathit{MCentre}\ \widehat{=}\ (t := 0).\overline{d_{0,1}}\langle(z, in, out, dn).P, t\rangle.d_{n,0}(\mathit{final}).\mathit{result}\langle\mathit{final}\rangle.\mathsf{0}
$$

Each host now has an extra component: one that receives the mobile process, executes it, and then passes it on to the next host. This component is merely added to the previous behaviour of the host.

$$
\begin{array}{l}
\mathit{MHost}_i\ \widehat{=}\ (\mathsf{new}\ c_i)(\mathit{HostC}_i \mid \mathit{Host}_i) \\
\mathit{HostC}_i\ \widehat{=} \\
\quad \begin{cases}
d_{i-1,i}(p, t_{i-1}).(\mathsf{new}\ tg)(p\langle c_i, t_{i-1}, tmp, tg\rangle \mid tg(t_i).\overline{d_{i,i+1}}\langle p, t_i\rangle.\mathsf{0}) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{for } i = 1 \,..\, n-1 \\
d_{n-1,n}(p, t_{n-1}).(\mathsf{new}\ tg)(p\langle c_i, t_{n-1}, tmp, tg\rangle \mid tg(t_n).\overline{d_{n,0}}\langle t_n\rangle.\mathsf{0}) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{for } i = n
\end{cases}
\end{array}
$$

[1]The mobile process is discarded in the last-visited host after its mission in order to save network cost, but extra cost arises for specifying the last host separately.

In $HostC_i$, we use a restricted name tg to make the output via $d_{i,i+1}$ possible after p's execution. If we discard $\overline{dn}\langle out\rangle$ in the transmitted abstraction, then the value of the updated t would not be passed on, and $p\langle c_i, t_{i-1}, tmp\rangle.\overline{d_{i,i+1}}\langle p, t_i\rangle.\mathsf{0}$ would be syntactically wrong, because $p\langle c_i, t_{i-1}, tmp\rangle$ is a process rather than a prefix.

By using some rules in HOπ and polyadic π-calculus, we can get an equivalence to *MSystem*: it is also weakly bisimilar to $result\langle u_1 + u_2 + \cdots + u_n\rangle$.

Theorem 3.3 (Distributed system bisimulation)

$$MSystem \approx result\langle u_1 + u_2 + \cdots + u_n\rangle$$

***Proof*:** *Let* $A_i = (\mathit{new}\, tg)(p\langle c_i, t_{i-1}, tmp, tg\rangle \mid tg(t_i).\overline{d_{i,i+1}}\langle p, t_i\rangle.\mathit{0})$

$$
\begin{array}{l}
MSystem \qquad \{\text{def of } MCentre \text{ and } MHost\} \\
\equiv^* (\mathsf{new}\ d_{0,1}, d_{1,2}, \cdots, d_{n,0}) \\
\left(\begin{array}{l}
t := 0.\overline{d_{0,1}}\langle P, t\rangle.d_{n,0}(final).result\langle final\rangle \\
\mid (\mathsf{new}\ c_1)(d_{0,1}(p, t_0).A_1 \mid \overline{c_1}\langle u_1\rangle) \\
\mid (\mathsf{new}\ c_2)(d_{1,2}(p, t_1).A_2 \mid \overline{c_2}\langle u_2\rangle) \\
\mid \cdots \\
\mid (\mathsf{new}\ c_{n-1})(d_{n-2,n-1}(p, t_{n-2}).A_{n-1} \mid \overline{c_{n-1}}\langle u_{n-1}\rangle) \\
\mid (\mathsf{new}\ c_n) \\
\quad (d_{n-1,n}(p, t_{n-1}).(\mathsf{new}\ tg)(p\langle c_n, t_{n-1}, tmp, tg\rangle \mid tg(t_n).\overline{d_{n,0}}\langle t_n\rangle) \mid \overline{c_n}\langle u_n\rangle)
\end{array}\right) \\
\qquad \{\text{structural congruence, def of } t := 0\} \\
\sim (\mathsf{new}\ d_{1,2}, \cdots, d_{n,0}) \\
\left(\begin{array}{l}
\left(\begin{array}{l}
\tau.(\mathsf{new}\ d_{0,1})\left(\begin{array}{l}\overline{d_{0,1}}\langle P, 0\rangle \\ \mid (\mathsf{new}\ c_1)(d_{0,1}(p, t_0).A_1 \mid \overline{c_1}\langle u_1\rangle)\end{array}\right) \\
.d_{n,0}(final).result\langle final\rangle
\end{array}\right) \\
\mid (\mathsf{new}\ c_2)(d_{1,2}(p, t_1).A_2 \mid \overline{c_2}\langle u_2\rangle) \\
\mid \cdots \\
\mid (\mathsf{new}\ c_{n-1})(d_{n-2,n-1}(p, t_{n-2}).A_{n-1} \mid \overline{c_{n-1}}\langle u_{n-1}\rangle) \\
\mid (\mathsf{new}\ c_n) \\
\quad (d_{n-1,n}(p, t_{n-1}).(\mathsf{new}\ tg)(p\langle c_n, t_{n-1}, tmp, tg\rangle \mid tg(t_n).\overline{d_{n,0}}\langle t_n\rangle) \mid \overline{c_n}\langle u_n\rangle)
\end{array}\right) \\
\qquad \{\text{synchronisation over } d_{0,1}, \text{ def of } p\langle c_1, t_0, tmp, tg\rangle\} \\
\sim (\mathsf{new}\ d_{1,2}, \cdots, d_{n,0}) \\
\left(\begin{array}{l}
\left(\begin{array}{l}
\tau.\tau.(\mathsf{new}\ c_1) \\
\quad ((\mathsf{new}\ tg)(c_1(v).tmp := 0 + v.\overline{tg}\langle tmp\rangle \mid tg(t_1).\overline{d_{1,2}}\langle P, t_1\rangle) \mid \overline{c_1}\langle u_1\rangle) \\
.d_{n,0}(final).result\langle final\rangle
\end{array}\right) \\
\mid (\mathsf{new}\ c_2)(d_{1,2}(p, t_1).A_2 \mid \overline{c_2}\langle u_2\rangle) \\
\mid \cdots \\
\mid (\mathsf{new}\ c_{n-1})(d_{n-2,n-1}(p, t_{n-2}).A_{n-1} \mid \overline{c_{n-1}}\langle u_{n-1}\rangle) \\
\mid (\mathsf{new}\ c_n) \\
\quad (d_{n-1,n}(p, t_{n-1}).(\mathsf{new}\ tg)(p\langle c_n, t_{n-1}, tmp, tg\rangle \mid tg(t_n).\overline{d_{n,0}}\langle t_n\rangle) \mid \overline{c_n}\langle u_n\rangle)
\end{array}\right) \\
\qquad \{\text{synchronisation over } tg, \text{ def of } tmp := v\} \\
\sim (\mathsf{new}\ d_{1,2}, \cdots, d_{n,0})
\end{array}
$$

$$
\left(\begin{array}{l}
\left(\begin{array}{l}
\tau.\tau.(\mathsf{new}\ c_1)(c_1(v).\tau.\tau.\overline{d_{1,2}}\langle P, v\rangle) \mid \overline{c_1}\langle u_1\rangle) \\
.d_{n,0}(\mathit{final}).\mathit{result}\langle \mathit{final}\rangle
\end{array}\right) \\
\mid (\mathsf{new}\ c_2)(d_{1,2}(p, t_1).A_2 \mid \overline{c_2}\langle u_2\rangle) \\
\mid \cdots \\
\mid (\mathsf{new}\ c_{n-1})(d_{n-2,n-1}(p, t_{n-2}).A_{n-1} \mid \overline{c_{n-1}}\langle u_{n-1}\rangle) \\
\mid (\mathsf{new}\ c_n) \\
\quad (d_{n-1,n}(p, t_{n-1}).(\mathsf{new}\ tg)(p\langle c_n, t_{n-1}, tmp, tg\rangle \mid tg(t_n).\overline{d_{n,0}}\langle t_n\rangle) \mid \overline{c_n}\langle u_n\rangle)
\end{array}\right)
$$

{synchronisation over c_1}

$$
\sim (\mathsf{new}\ d_{1,2}, \cdots, d_{n,0})
\left(\begin{array}{l}
\left(\begin{array}{l}
\tau.\tau.\tau.(\tau.\overline{d_{1,2}}\langle P, u_1\rangle) \\
.d_{n,0}(\mathit{final}).\mathit{result}\langle \mathit{final}\rangle
\end{array}\right) \\
\mid (\mathsf{new}\ c_2)(d_{1,2}(p, t_1).A_2 \mid \overline{c_2}\langle u_2\rangle) \\
\mid \cdots \\
\mid (\mathsf{new}\ c_{n-1})(d_{n-2,n-1}(p, t_{n-2}).A_{n-1} \mid \overline{c_{n-1}}\langle u_{n-1}\rangle) \\
\mid (\mathsf{new}\ c_n) \\
\quad (d_{n-1,n}(p, t_{n-1}).(\mathsf{new}\ tg)(p\langle c_n, t_{n-1}, tmp, tg\rangle \mid tg(t_n).\overline{d_{n,0}}\langle t_n\rangle) \mid \overline{c_n}\langle u_n\rangle)
\end{array}\right)
$$

{weak bisimilarity}

$$
\approx (\mathsf{new}\ d_{1,2}, \cdots, d_{n,0})
\left(\begin{array}{l}
\overline{d_{1,2}}\langle P, u_1\rangle.d_{n,0}(\mathit{final}).\mathit{result}\langle \mathit{final}\rangle \\
\mid (\mathsf{new}\ c_2)(d_{1,2}(p, t_1).A_2 \mid \overline{c_2}\langle u_2\rangle) \\
\mid \cdots \\
\mid (\mathsf{new}\ c_{n-1})(d_{n-2,n-1}(p, t_{n-2}).A_{n-1} \mid \overline{c_{n-1}}\langle u_{n-1}\rangle) \\
\mid (\mathsf{new}\ c_n) \\
\quad (d_{n-1,n}(p, t_{n-1}).(\mathsf{new}\ tg)(p\langle c_n, t_{n-1}, tmp, tg\rangle \mid tg(t_n).\overline{d_{n,0}}\langle t_n\rangle) \mid \overline{c_n}\langle u_n\rangle)
\end{array}\right)
$$

{induction over the indices as from step*}

$$
\approx \mathit{result}\langle u_1 + u_2 + \cdots + u_n\rangle
$$

□

Theorem 3.2 and Theorem 3.3 show that the π-calculus can be used for verification. If we have got a specification in the π-calculus, it is possible for us to use the structural congruence rules and strong/weak bisimulation rules to deduce that this specification satisfies some requirement. Both *System* and *MSystem* are observationally equivalent to $\mathit{result}\langle u_1 + u_2 + \cdots + u_n\rangle$, so they both satisfy the requirement of getting the sum of the data in each host.

We are interested in a step-wise development process in the spirit of Morgan's refinement calculus for sequential programs [21]. We would like to start from an abstract, centralised specification, and develop a concrete, distributed implementation, proceeding in small steps that are easy to justify and that explain design decisions. It is not easy to follow this discipline in the π-calculus. Instead, we would like to base our language of mobile processes firmly on the notion of refinement, and develop sets of laws that encourage piece-wise development. In later sections, after we present the denotational semantics and refinement laws for mobile processes, we show this step-wise development in our proposed approach.

4 Syntax

The syntax of our language is a subset of occam [16] and CSP [12, 24], but enriched with primitives for process variables, mobile and clone assignment and communication, and (parameterised) process variable activation. These mobility constructs are inspired by occam$_\mathsf{M}$ [1].

In discussing the semantics, we make use of the following conventions for meta-variables. p and q range over all program variables; t ranges over data variables; e ranges over data; h ranges over process variables; E ranges over data or process values; x, y and z range over formal name, value, and result parameters; ne, ve, and re range over actual name, value, and result parameters; b ranges over boolean values; X ranges over sets of events.

The basic elements in our model are processes, which are constructed from the following rules:

$$
\begin{array}{lll}
P & ::= & SKIP \mid STOP \mid CHAOS \mid \textbf{\textit{vid}}\ p : T \mid \textbf{\textit{end}}\ p \\
& \mid & h := \{\!|P|\!\} \mid h := \{\!|\lambda\, x : var(T_1), y : val(T_2), z : res(T_3) \bullet P|\!\} \\
& \mid & t := e \mid p :=_m q \mid p := q \mid {<}h{>} \mid h(ne, ve, re) \\
& \mid & Comm \rightarrow P \mid P \lhd b \rhd Q \mid P\ ;\ Q \mid b * P \\
& \mid & P \parallel Q \mid P \,\Box\, Q \mid P \sqcap Q \mid P \setminus X \\
\textbf{\textit{vid}} & ::= & \textbf{\textit{var}} \mid \textbf{\textit{proc}} \\
Comm & ::= & ch?p \mid ch!q \mid ch!!q \mid ch.E
\end{array}
$$

$SKIP$ does nothing and terminates immediately; $STOP$ represents deadlock; $CHAOS$ is the worst process and the bottom element of the complete lattice of mobile process: its behaviour is arbitrary.

The variable declaration $\textbf{\textit{vid}}\ p : T$ introduces a new variable p of type T; correspondingly $\textbf{\textit{end}}\ p$ terminates the scope of p. When it is clear from the context that p is a data variable or a process variable, we use ***var*** or ***proc*** for its declaration. The type T determines the range of values that p can have. When p is a process variable, its type determines the alphabet and interface[2] of the process that may be assigned to p. For convenience, we often omit types when they are irrelevant.

Higher-order programming treats program as data, and higher-order assignment or communication assigns process constants to higher-order variables. Process constants are enclosed in brackets, which have no semantic importance and are ignored when the process is activated. We distinguish simple process constants from parameterised ones. The higher-order constant assignment $h := \{\!|P|\!\}$ assigns a simple process constant P to h, and $h := \{\!|\lambda\, x : var(T_1), y : val(T_2), z : res(T_3) \bullet P|\!\}$ assigns h a parameterised process constant, which has a body P and a formal name parameter x, value parameter y, and result parameter z.

The first-order constant assignment $t := e$ assigns a value e to the data variable t. The clone variable assignment $p := q$ is similar to constant assignment, except that the term in its right hand side is a variable rather than a

[2]The interface is defined as parameters and input/output channels through which the process can interact with its environment.

value. After this clone assignment, the value of p is updated according to q's value, and q gets a value that is better than its original one (in Section 5.3, we explain what a better value is and why the value should be better). The notation $p :=_m q$ denotes mobile variable assignment. On its termination, the value of the target variable p is updated and the source variable q is undefined, thus the result of any attempt of using q is unpredictable.

The notation $ch.E$ stands for a communication that takes place as soon as both participants are ready. By executing the input prefix $ch?p \rightarrow P$, the variable p accepts a message from the channel, and then the program behaves like P. The mobile output prefix $ch!!q \rightarrow P$ transfers the value of variable q through channel ch and then executes P. As in mobile assignment, any attempt to use q after output is unpredictable. The clone output prefix $ch!q \rightarrow P$ outputs the value of q, but retains q's value.

Once initialised, a process variable h may be activated by executing it, denoted by $<h>$. A parameterised process variable can be activated by providing parameters.

A conditional $(P \lhd b \rhd Q)$ behaves as P if b is *true*, otherwise as Q. The sequential composition of two processes $(P \ ; \ Q)$ results in a process that behaves as P, and, on termination of P, behaves as Q. An iteration $(b * P)$ repeatedly executes P until b is *false*. A parallel composition $(P \parallel Q)$ executes P and Q concurrently, such that events in the alphabet of both parties require their simultaneous participation, whereas the remaining events occur independently. An external choice $(P \Box Q)$ allows the environment to choose between P and Q, whereas the internal choice $(P \sqcap Q)$ selects one of the two processes nondeterministically. $P \setminus X$ hides the events in the set X, so that they happen invisibly, without the participation of the environment.

5 Semantics

This section present the denotational semantics for mobile processes. There are two main differences in our approach of higher-order programming with conventional first-order programming. In first-order programming, we only discuss the refinement ordering between processes. In our approach, however, as mobile processes are modelled as the values of higher-order variables, there are two new refinement orderings: refinement between variables and refinement between higher-order variables and processes. These three refinement orderings are defined in Section 5.1. After giving the UTP model that we use for the semantics of mobile processes in Section 5.2, we present the other difference which is in the semantics of higher-order assignment and communication in Section 5.3. Finally, the denotational semantics is given in the Section 5.4.

5.1 Refinement orderings

In first-order programming, we say a process P is refined by Q, denoted by $P \sqsubseteq Q$, if for all variables in the alphabet, the behaviour of Q is more predictable

and controllable than that of P.

Definition 5.1 (Process refinement)

$$P \sqsubseteq Q \ \widehat{=}\ \forall\, \boldsymbol{obs}, \boldsymbol{obs}', v, v' \bullet Q \Rightarrow P$$

where $\boldsymbol{obs}, v$ *are observable and program variables (see Section 5.2.1 for details) respectively. The universal closure over the alphabet is written more concisely:* $[\, Q \Rightarrow P \,]$. □

As higher-order variables hold processes as their values, the ordering between program variables can be defined in the light of the refinement ordering between their values. Two process variables can be compared only when they have the same types. We say a process variable h is a refinement of g, if the activation behaviour of h is more controllable and predictable than that of g. For first-order data variables, two variables are comparable only when they are equal. More specifically, we define the refinement ordering between variables as follows.

Definition 5.2 (Variable refinement) *Let* p *and* q *be two program variables of the same type*

$$p \sqsubseteq q \ \widehat{=}$$
$$\begin{cases} p = q & \text{if } p, q \text{ are data variables} \\ [\, q(ne, ve, re) \Rightarrow p(ne, ve, re) \,] & \text{if } p, q \text{ are parameterised process variables} \\ [\, <q> \Rightarrow <p> \,] & \text{otherwise} \end{cases}$$

where ne*,* ve*, and* re *are the actual name parameter, value parameter and result parameter for the activations of* p *and* q. □

A refinement ordering between process variable and process can be defined similarly.

Definition 5.3 (Variable process refinement) *Suppose* h *is a process variable and* Q *is a process,* h *and* Q *have the same type.*

$$h \sqsupseteq Q \ \widehat{=}$$
$$\begin{cases} [\, h(ne, ve, re) \Rightarrow Q(ne, ve, re) \,] & \text{if } h \text{ and } Q \text{ are parameterised} \\ [\, <h> \ \Rightarrow \ Q \,] & \text{otherwise} \end{cases}$$

where $\boldsymbol{obs}, v$ *and* ne, ve, re *are same as that in Definition 5.2.* □

5.2 UTP semantics model

In this section we present the UTP semantics model to be used to formalise mobile processes.

5.2.1 Alphabet

In the UTP, a process P is described in terms of an implicit description of its entire behaviour using the alphabet. The alphabet of our model consists of the following external observable variables and program variables:

- $\mathcal{A}$: the set of events in which P can engage
- ok, ok': $\mathbb{B}$, indicates divergence. $ok = true$ represents that P has been properly started in a stable state. $ok' = true$ indicates subsequent stabilisation in an observational state, either in an intermediate state or a final one, and $ok' = false$ indicates the program is divergent.
- $wait, wait'$: $\mathbb{B}$, indicates termination. $wait = true$ means that P has been started in an intermediate state, and $wait' = true$ indicates that the program has not terminated in which case all the other dashed variables stand for intermediate observations rather than final ones.
- tr, tr': $\mathrm{seq}\,\mathcal{A}$. They represent the cumulative records of all interactions that P has made so far. tr records the sequence of events which took place before P started, and tr' refers to the sequence of all events at the moment when the observation takes place.
- ref, ref': $\mathbb{P}\mathcal{A}$. ref represents the set of events refused by P before it has started, and ref' stands for the set of events refused by P at the time of observation.
- v, v': all program variables including process variables and data variables

We use ***obs*** to represent all observable variables $ok, wait, tr, ref$ in short. P's behaviour is described by a relation between undashed and dashed variables. By using this model, the refusal can be indicated by ref, and the divergence is captured by the ok variable. The boolean $wait$ variable describes the state in which the process is waiting for interaction with the environment. Therefore, we are able to reason about the refusal and divergence behaviours of mobile processes.

We use αP to denote the alphabet of process P, and

$$\alpha P \;=\; in\alpha P \cup out\alpha P$$

where $in\alpha P$ is a set of undashed variables recording initial values, and $out\alpha P$ is a set of dashed variables recording final values.

Usually, we use subscript to indicate the alphabet of a process. For instance, P_A denotes a process P of alphabet A. The alphabet of a process can be extended with a new variable by conjunction with a predicate that indicates the final value of this variable is a refinement of its initial value. The reason why we adopt refinement rather than equivalence will be explained in Section 5.3.

Definition 5.4 (Alphabet extension) *Let* $p, p' \notin \alpha P$

$$\begin{array}{rcl} P_{+p} & \widehat{=} & P \wedge (p' \sqsupseteq p) \\ \alpha(P_{+p}) & \widehat{=} & \alpha P \cup \{p, p'\} \end{array} \qquad \square$$

Alphabet extension is often needed to make sure sequential composition works properly. Further discussion will be shown in a later section.

5.2.2 Healthiness conditions

Healthiness conditions distinguish feasible specifications or designs from infeasible ones, where infeasibility means impossible implementation in the target programming language. There are five healthiness conditions for mobile processes as follows.

$$\begin{array}{ll} \boldsymbol{M1} & P = P \wedge (tr \leq tr') \\ \boldsymbol{M2} & P = \sqcap\{P[s, s \frown (tr' - tr)/tr, tr'] \mid s \in \text{seq } \mathcal{A}\} \\ \boldsymbol{M3} & P = I\!I \lhd wait \rhd P \\ & \quad \textbf{\textit{where}} \\ & \quad I\!I \ \widehat{=} \ I \lhd ok \rhd (tr \leq tr') \\ & \quad I \ \widehat{=} \ (ok' = ok) \wedge (tr' = tr) \wedge (wait' = wait) \wedge (ref' = ref) \wedge (v' \sqsupseteq v) \\ \boldsymbol{M4} & P = P \lhd ok \rhd (tr \leq tr') \\ \boldsymbol{M5} & P = P \,;\, J \\ & \quad \textbf{\textit{where}} \\ & \quad J \ \widehat{=} \ (ok \Rightarrow ok') \wedge (tr' = tr) \wedge (wait' = wait) \wedge (ref' = ref) \wedge (v' \sqsupseteq v) \end{array}$$

Let

$$\begin{array}{rcl} \boldsymbol{M1}(P) & = & P \wedge (tr \leq tr') \\ \boldsymbol{M2}(P) & = & \sqcap\{P[s, s \frown (tr' - tr)/tr, tr'] \mid s \in \text{seq } \mathcal{A}\} \\ \boldsymbol{M3}(P) & = & I\!I \lhd wait \rhd P \\ \boldsymbol{M4}(P) & = & P \lhd ok \rhd (tr \leq tr') \\ \boldsymbol{M5}(P) & = & P \,;\, J \end{array}$$

thus every healthiness condition $\boldsymbol{Mi}$ can be written as

$$P = \boldsymbol{Mi}(P)$$

If P satisfies it, we say 'P is $\boldsymbol{Mi}$' or 'P is $\boldsymbol{Mi}$ healthy'. In later discussion, we will use abbreviations to represent function composition and healthiness conditions for easier presentation. For instance, $\boldsymbol{M23}(P)$ is used to refer to $\boldsymbol{M2} \diamond \boldsymbol{M3}(P)$, '$P$ is $\boldsymbol{M12}$' refers to P satisfies '$\boldsymbol{M1}$ and $\boldsymbol{M2}$'.

The sequential composition $(P \,;\, Q)$ behaves as P until P terminates successfully, at which point it passes control to Q.

Definition 5.5 (Sequential composition) *Provided* $out\alpha P = in\alpha' Q = \{\alpha'\}$, *then*

$$[\![P;\ Q]\!] \ \widehat{=} \ \exists \alpha_0 \bullet [\![P]\!][\alpha_0/\alpha'] \wedge [\![Q]\!][\alpha_0/\alpha] \qquad \square$$

In the sequential composition $(P\ ;\ Q)$, the final state of P that is recorded by α' is passed on as the initial state of Q that is recorded by α, but this state is an intermediate state of $(P\ ;\ Q)$, and cannot be observed from the environment. The definition uses existential quantification to hide the intermediate observation, and to remove the variables that record it from the alphabet. Thus a new set of variables α_0 is introduced to record the hidden observation.

In reactive programming, the sequential composition has three possible states. The first state is that if the first process diverges, then the sequential composition does as well. The second state is that if the first process is in a waiting state, then the following process cannot start. The third is that if the first process terminates and does not diverge, then the following process starts immediately after the first one terminates. Alternatively, the sequential composition in reactive programming is the same as

$$\begin{array}{rcl} [\![P\ ;\ Q]\!] & = & [\![P]\!][\mathit{false}/\mathit{ok}'] \vee [\![P]\!][\mathit{true}/\mathit{wait}'] \\ & & \vee\ ([\![P]\!][\mathit{true}, \mathit{false}/\mathit{ok}', \mathit{wait}'] \diamond [\![Q]\!]) \end{array}$$

where the operator $\diamond$ is to denote the concatenation of two sequentially made observations. For two observation predicates $A(o'), B(o)$, where o, o' represent the initial and final value of all the observation variables respectively, the concatenation of them is defined as

$$A(o') \diamond B(o) \ \widehat{=}\ \exists\, o_0 \bullet A(o_0) \wedge B(o_0)$$

Because the alphabet of a process can always be extended as defined in Definition 5.4, we can ensure that any sequential composition is always meaningful. In later discussion, when there is no confusion or we are not interested in the alphabet, we often omit the alphabet extension when we sequentially composite two processes which have different alphabets.

The conditional $P \lhd\ b\ \rhd Q$ defines a choice on two alternatives P and Q in accordance with the initial value of b, where b is a program expression, containing only initial variables and always producing a Boolean result. If b is true, it then acts like P, otherwise it behaves like Q instead.

Definition 5.6 (conditional) *Provided* $\alpha P = \alpha Q$*, then*

$$[\![P \lhd\ b\ \rhd Q]\!] \ \widehat{=}\ b \wedge [\![P]\!] \vee \neg\ b \wedge [\![Q]\!] \qquad \Box$$

Please note that to make this definition meaningful, there is a constraint that the two conditional branches have the same alphabet. This is because the alphabet of a process should be fixed, and cannot depend on the value of b.

The sequential composition and conditional enjoy some algebraic properties.

Law 5.1 (; associative)

$$(P;\ Q);\ R\ =\ P;\ (Q;\ R) \qquad \Box$$

Law 5.2 (; conditional left distributive)

$$(P \lhd\ b\ \rhd Q);\ R\ =\ (P;\ R) \lhd\ b\ \rhd (Q;\ R) \qquad \Box$$

The first healthiness condition (***M1***) says that a process can never undo its execution. In other words, for any valid observation, the final value of the trace tr' is always an expansion of the initial one tr.

The second healthiness condition (***M2***) indicates that the initial value of trace tr has no influence on a process. In other words, tr may be replaced by an arbitrary trace s and the events which the process itself involves keep the same as $(tr' - tr)$.

The third healthiness condition (***M3***) ensures that the sequential composition works as supposed. Let us consider the sequential composition of two processes $X;P$. Control can pass from X to P only when X terminates, indicated by the fact $wait'_X = wait_P = false$. If the predecessor X is in a waiting state, indicated by the fact $wait'_X = wait_P = true$, the process P will leave the state unchanged.

The fourth one (***M4***) states that if a process has not started, then we cannot predict its behaviour except that its final trace tr' is an expansion of its initial one tr.

The fifth healthiness condition (***M5***) shows the fact that divergence is something that is never wanted. This is characterised by the monotonicity of ok': if P is not divergent, indicated by $ok = true$, then we can say $P;J$ is not divergent as well, indicated by $ok' = true$; but if P is divergent, indicated by $ok = false$, we cannot insist on the divergence of $(P\ ;\ J)$, indicated by ok' can be either *true* or *false*.

Refinement ordering between program variables is adopted in the above healthiness conditions. Identity I does not change any value of data and observation variables but allows process variables to be improved. It is the left unit of sequential composition of any predicate.

Law 5.3 (Left unit for predicate) *Any predicate Pred satisfies*

$$I;\ Pred\ =\ Pred$$

***Proof*:**

(1) $I\ ;\ Pred \sqsupseteq Pred$

$I\ ;\ Pred$ {def of I and ;}

$= \exists\, v_0, \boldsymbol{obs}_0 \bullet v_0 \sqsupseteq v \wedge \boldsymbol{obs}_0 = \boldsymbol{obs} \wedge Pred(\boldsymbol{obs}_0, v_0, \boldsymbol{obs}', v')$ {one-point rule}

$= \exists\, v_0 \bullet v_0 \sqsupseteq v \wedge Pred(v_0, v')$ {initial state monotonicity, $v_0 \sqsupseteq v$}

$\sqsupseteq (\exists\, v_0 \bullet v_0 \sqsupseteq v) \wedge Pred(v, v')$ {let $v_0 = v$}

$= true \wedge Pred$

$= Pred$

(2) $I\ ;\ Pred \sqsubseteq Pred$

$I\ ;\ Pred$ {def of I and ;}

$= \exists\, v_0 \bullet v_0 \sqsupseteq v \wedge Pred(v_0, v')$ {$(v_0 \sqsupseteq v) \sqsubseteq (v_0 = v)$}

$\sqsubseteq \exists\, v_0 \bullet v_0 = v \wedge Pred(v_0, v')$ {one-point rule}

$= Pred$ □

Law 5.4 (Left zero for J)

$$(tr \leq tr' \ ; J) \ = \ tr \leq tr'$$

***Proof*:**

LHS {def of J, let x contains $ref, wait, v$}
$= tr \leq tr' \ ; ok \Rightarrow ok' \wedge tr' = tr \wedge x' = x$ {def of ;}
$= \exists\, tr_0, ok_0, x_0 \bullet tr \leq tr_0 \wedge ok_0 \Rightarrow ok' \wedge tr' = tr_0 \wedge x' = x_0$ {predicate calculus}
$= (\exists\, tr_0 \bullet tr \leq tr_0 \wedge tr' = tr_0) \wedge (\exists\, ok_0 \bullet ok_0 \Rightarrow ok') \wedge (\exists\, x_0 \bullet x' = x_0)$
{one-point rule, let $ok_0 = ok'$, $x_0 = x'$}
$= (tr \leq tr') \wedge true \wedge true$
$= tr \leq tr'$ □

Healthiness conditions satisfies some algebraic properties.

Theorem 5.1 (Property of M)

(1) *If P is **M14**, then P satisfies*
$P = (I\!I;\ P)$ and $(tr \leq tr';\ P) = tr \leq tr'$
(2) *If P satisfies $P = (I\!I;\ P)$ and $(tr \leq tr';\ P) = tr \leq tr'$,*
*then P is **M4**.*

***Proof*:**

(1.1)
$tr \leq tr';\ P$ {assumption: P is ***M14***}
$= tr \leq tr';\ (P \wedge tr \leq tr') \lhd ok \rhd tr \leq tr'$ {def of ;}
$= \exists\, ok_0, tr_0, \cdots \bullet (tr \leq tr_0 \wedge ((P \wedge tr_0 \leq tr') \lhd ok_0 \rhd tr_0 \leq tr'))$
{$\wedge$-distr-over-$\lhd\rhd$, predicate calculus}
$= \exists\, ok_0, \cdots \bullet (P \wedge tr \leq tr') \lhd ok_0 \rhd (tr \leq tr')$ {ok_0 case analysis}
$= ((P \wedge tr \leq tr') \lhd ok_0 \rhd (tr \leq tr'))[true/ok_0] \vee$
$((P \wedge tr \leq tr') \lhd ok_0 \rhd (tr \leq tr'))[false/ok_0]$ {substitution}
$= (P \wedge tr \leq tr') \vee (tr \leq tr')$ {absorption}
$= tr \leq tr'$
(1.2)
$I\!I;\ P$ {def of $I\!I$}
$= (I \lhd ok \rhd tr \leq tr');\ P$ {;-left-distr-over-$\lhd\rhd$}
$= (I;\ P) \lhd ok \rhd (tr \leq tr';\ P)$ {Law 5.3, $(tr \leq tr';\ P) = tr \leq tr'$}
$= P \lhd ok \rhd tr \leq tr'$ {assumption: P is ***M4***}
$= P$
(2)

$P \lhd ok \rhd tr \leq tr'$ {Law 5.3, assumption: $(tr \leq tr';\ P) = tr \leq tr'$}
$= (I;\ P) \lhd ok \rhd (tr \leq tr';\ P)$ {;-left-distr-over-$\lhd\rhd$}
$= (I \lhd ok \rhd tr \leq tr');\ P$ {def of $I\!I$, assumption: $P = I\!I;\ P$}
$= P$

□

Definition 5.7 (*MP* process) *A predicate P is called an **MP** process or a healthy process if it satisfies $P = \boldsymbol{M12345}(P)$.* □

In a design calculus, for any operator to be useful, the combination of two healthy processes by this operator should be healthy as well. The fact is captured by the closure of ***MP*** processes under this operator, therefore we have the obligation to discuss and prove this issue when a new operator is defined.

Theorem 5.2 (*MP* process closure) *The set of **MP** processes form a complete lattice, which is a $\{\wedge, \vee,\ ;\ \}$-closure.*

***Proof*:** *Let α denote all alphabets except the mentioned one, we only show ; -closure: **M12345**(P); **M12345**(Q)=**M12345**$(P;\ Q)$*

M1$.\ P;\ Q$ {P and Q are ***M1***}
$= P \wedge tr \leq tr';\ Q \wedge tr \leq tr'$ {def of ; }
$= \exists\, tr_0 \bullet (P;\ Q) \wedge tr \leq tr_0 \wedge tr_0 \leq tr'$ {predicate calculus}
$= (P;\ Q) \wedge (tr \leq tr')$
$= \boldsymbol{M1}(P;\ Q)$

$$
\begin{array}{ll}
\boldsymbol{M2}.\ P;\ Q & \{\text{def of ; }\} \\
= \exists\, \alpha_0, tr_0 \bullet (P(\alpha, tr, \alpha_0, tr_0) \wedge Q(\alpha_0, tr_0, \alpha', tr')) & \{P \text{ and } Q \text{ are } \boldsymbol{M2}\} \\
= \exists\, \alpha_0, tr_0 \bullet \left(\begin{array}{l} \prod_{s_1} P(\alpha, s_1, \alpha_0, s_1 \frown (tr_0 - tr)) \wedge \\ Q(\alpha_0, s_1 \frown (tr_0 - tr), \alpha', s_1 \frown (tr_0 - tr) \frown (tr' - tr_0)) \end{array} \right) & \{\text{def of ;}\} \\
= \exists\, tr_0 \bullet \prod_{s_1} (P;\ Q)(s_1, s_1 \frown (tr_0 - tr) \frown (tr' - tr_0))) & \{\text{sequence calculation}\} \\
= \prod_{s_1} (P;\ Q)(s_1, s_1 \frown (tr' - tr))) & \\
= \ \boldsymbol{M2}(P;\ Q) & \\
\boldsymbol{M3}.\ P;\ Q & \{P \text{ is } \boldsymbol{M3}\} \\
= (I\!I \lhd \mathit{wait} \rhd P);\ Q & \{\text{;-left-distr-over-}\lhd\rhd\} \\
= (I\!I;\ Q) \lhd \mathit{wait} \rhd (P;\ Q) & \{Q \text{ is } \boldsymbol{M14}, \text{ Theorem 5.1}\} \\
= Q \lhd \mathit{wait} \rhd (P;\ Q) & \{Q \text{ is } \boldsymbol{M3}\} \\
= (I\!I \lhd \mathit{wait} \rhd Q) \lhd \mathit{wait} \rhd (P;\ Q) & \{\text{def of conditional}\} \\
= I\!I \lhd \mathit{wait} \rhd (P;\ Q) & \\
= \boldsymbol{M3}(P;\ Q) & \\
\boldsymbol{M4}.\ P;\ Q & \{P \text{ is } \boldsymbol{M4}\} \\
= (P \lhd \mathit{ok} \rhd \mathit{tr} \leq \mathit{tr}');\ Q & \{\text{;-left-distr-over-}\lhd\rhd\} \\
= (P;\ Q) \lhd \mathit{ok} \rhd (\mathit{tr} \leq \mathit{tr}';\ Q) & \{Q \text{ is } \boldsymbol{M14}, \text{ Theorem 5.1}\} \\
= (P;\ Q) \lhd \mathit{ok} \rhd \mathit{tr} \leq \mathit{tr}' & \\
= \boldsymbol{M4}(P;\ Q) & \\
\boldsymbol{M5}.\ P;\ Q & \{Q \text{ is } \boldsymbol{M5}\} \\
= P;\ (Q;\ J) & \{\text{; -assoc}\} \\
= (P;\ Q);\ J & \\
= \boldsymbol{M5}(P;\ Q) &
\end{array}
$$

□

5.3 Higher-order assignment

Besides the refinement ordering between program variables, another difference with first-order programming in our approach is in the semantics of higher-order assignment and communication.

First-order assignment $t := e$ has no interaction with the environment. It always terminates and never diverges. On termination, it equates the final value of t with value e, but does not change the other variables.

Definition 5.8 (First-order assignment)

$$[\![t := e]\!] \ \widehat{=}\ \boldsymbol{M34}(\mathit{ok}' \wedge \neg\, \mathit{wait}' \wedge \mathit{tr}' = \mathit{tr} \wedge t' = e \wedge v' = v)$$

where the function application of $\boldsymbol{M34}$ *to the predicate is to ensure that assignment is healthy. The satisfaction of the other healthiness conditions can be derived from the definition (see Section 5.4).* □

In higher-order programming, we need change the semantics of higher-order assignment to allow an implementation of replacing the assigned value by anyone that refines it.

Definition 5.9 (Higher-order constant assignment) *Let* $\alpha h = \alpha P$, $h, h' \notin \alpha P$

$$\begin{array}{rl} [\![h := \{\!|P|\!\}]\!] & \widehat{=}\ \boldsymbol{M34}(ok' \wedge \neg\ wait' \wedge tr' = tr \wedge h' \sqsupseteq P \wedge v' \sqsupseteq v) \\ \alpha(h := \{\!|P|\!\}) & = \{\boldsymbol{obs}, \boldsymbol{obs}', h, h', v, v'\} \end{array}$$

where v *are all program variables except* h. □

The reason is to guarantee that assignment is monotonic with respect to refinement in the assigned value. This can be seen in the following theorem.

Theorem 5.3 (Assignment monotonicity) *Suppose that* h *is a higher-order variable.*

$$(P \sqsubseteq Q)\ \Rightarrow\ (h := \{\!|P|\!\} \sqsubseteq h := \{\!|Q|\!\})$$

***Proof*:**

$$\begin{array}{ll} (P \sqsubseteq Q) \Rightarrow (h := \{\!|P|\!\} \sqsubseteq h := \{\!|Q|\!\}) & \{\text{Def. 5.1 and 5.9}\} \\ \equiv (P \sqsubseteq Q) \Rightarrow & \\ \quad \left[\begin{array}{l} \boldsymbol{M34}(ok' \wedge \neg\ wait' \wedge tr' = tr \wedge h' \sqsupseteq Q \wedge v' \sqsupseteq v) \\ \quad \Rightarrow \boldsymbol{M34}(ok' \wedge \neg\ wait' \wedge tr' = tr \wedge h' \sqsupseteq P \wedge v' \sqsupseteq v) \end{array} \right] & \\ \qquad \{\boldsymbol{M3} \text{ and } \boldsymbol{M4}, \text{ one-point, propositional calculus}\} & \\ \equiv (P \sqsubseteq Q) \Rightarrow [(ok' \wedge \neg\ wait' \wedge h' \sqsupseteq Q \wedge v' \sqsupseteq v) \Rightarrow (h' \sqsupseteq P)] & \\ \qquad \{\text{universal closure, case analysis}\} & \\ \equiv (P \sqsubseteq Q) \Rightarrow [(h' \sqsupseteq Q) \Rightarrow (h' \sqsupseteq P)] & \{\text{universal closure, } h' \notin \alpha P, h' \notin \alpha Q\} \\ \equiv \forall\, h' \bullet (P \sqsubseteq Q) \Rightarrow (h' \sqsupseteq Q \Rightarrow h' \sqsupseteq P) & \{\text{propositional calculus}\} \\ \equiv \forall\, h' \bullet (P \sqsubseteq Q \wedge h' \sqsupseteq Q) \Rightarrow h' \sqsupseteq P) & \{\text{transitivity of refinement}\} \\ \equiv \forall\, h' \bullet (P \sqsubseteq Q \wedge h' \sqsupseteq Q \wedge h' \sqsupseteq P) \Rightarrow (h' \sqsupseteq P) & \{\text{propositional calculus}\} \\ \equiv \forall\, h' \bullet true & \{\text{predicate calculus}\} \\ \equiv true & \end{array}$$

□

Adoption of equation in the definition of assignment will lead to a contradiction that $(P \sqsubseteq Q)$ implies $(P = Q)$:

$$
\begin{array}{ll}
h := \{\!|P|\!\} \sqsubseteq h := \{\!|Q|\!\} & \{\text{refinement (definition 5.1)}\} \\
= [\, h := \{\!|Q|\!\} \Rightarrow h := \{\!|P|\!\}\,] & \{\text{assumption (definition of assignment)}\} \\
= [\, \textbf{\textit{M34}}(ok' \wedge \neg\, wait' \wedge tr' = tr \wedge h' = \{\!|Q|\!\} \wedge v' = v) \Rightarrow & \\
\qquad \textbf{\textit{M34}}(ok' \wedge \neg\, wait' \wedge tr' = tr \wedge h' = \{\!|P|\!\} \wedge v' = v)\,] & \\
& \{\text{definition of } \textbf{\textit{M3}} \text{ and } \textbf{\textit{M4}}\} \\
= [\, ok' \wedge \neg\, wait' \wedge tr' = tr \wedge h' = \{\!|Q|\!\} \wedge v' = v \Rightarrow & \\
\qquad ok' \wedge \neg\, wait' \wedge tr' = tr \wedge h' = \{\!|P|\!\} \wedge v' = v\,] & \\
& \{\text{one-point rule, three times}\} \\
= [\, ok' \wedge \neg\, wait' \Rightarrow ok' \wedge \neg\, wait' \wedge \{\!|Q|\!\} = \{\!|P|\!\}\,] & \{\text{propositional calculus}\} \\
= [\, ok' \wedge \neg\, wait' \Rightarrow \{\!|Q|\!\} = \{\!|P|\!\}\,] & \{\text{universal closure, case analysis}\} \\
= (P = Q) &
\end{array}
$$

For the same reason, we also adopt the inequation in the definitions of higher-order communication and I, $I\!I$ and $SKIP$. For first-order data, two values are comparable if and only if they are equal. Therefore higher-order assignment or communication and $SKIP$ are consistent with their counterpart in first-order programming. We extend the language, but we are interested in a conservative extension in which the new semantics is not only suitable for the extension part, but also for the original part.

5.4 Denotational semantics of mobile processes

5.4.1 Primitive processes

We first discuss some simple processes, which serve as primitives and normally need to be used in combination to exhibit more complex behaviours.

SKIP The process $SKIP$ refuses to engage in any communication event, but terminates immediately. It allows improvement of all program variables.

Definition 5.10 (SKIP) *Let* $\alpha SKIP = A = \{\textbf{\textit{obs}}, \textbf{\textit{obs}}', v, v'\}$

$$[\![SKIP]\!] \;\widehat{=}\; \textbf{\textit{M123}}\,(\textbf{\textit{true}}_{\{ref, ref'\}}\;;\; I\!I)$$

where $\textbf{\textit{true}}_{\{ref, ref'\}}$ *is a predicate whose alphabet only contains* ref *and* ref' *that have arbitrary values.* □

The definition indicates that for process $SKIP$, the initial and final values of ref are entirely irrelevant. We adopt different notation $\textbf{\textit{true}}_{\{ref, ref'\}}$ to represent it from the existential quantification over ref in the definition of $SKIP$ appeared in [13] chapter 9. The reason is to avoid the inconsistency in existential quantification with that in the definitions of sequential composition and variable declaration. In the definition of sequential composition, the existential quantification hides the intermediate observation and removes the variables that record it from the alphabet. In a later section, we will also see that in the definition of variable declaration, the existential quantification over a variable

removes this variable from the alphabet. In the definition of *SKIP*, however, *ref* should be in the alphabet.

Process *SKIP* is an ***MP*** process, but we only apply ***M123*** in its definition as ***M4*** and ***M5*** healthiness can be implied or proved by expanding its definition.

$$
\begin{array}{ll}
[\![SKIP]\!] & \{\text{def of } I\!I, \text{ alphabet extension}\} \\
= \boldsymbol{M123}\left(\begin{array}{l} \boldsymbol{true}_{\{ref,ref'\}} \wedge \\ \quad (ok' = ok \wedge wait' = wait \wedge tr' = tr \wedge v' \sqsupseteq v); \\ (\boldsymbol{obs}' = \boldsymbol{obs} \wedge v' \sqsupseteq v) \lhd ok \rhd tr \leq tr' \end{array}\right) & \{\text{def of } ;\} \\
= \boldsymbol{M123}\left(\begin{array}{l} \exists\, \boldsymbol{obs}_0, v_0 \bullet \\ \quad \boldsymbol{true}_{\{ref,ref_0\}} \wedge \\ \quad\quad (ok_0 = ok \wedge wait_0 = wait \wedge tr_0 = tr \wedge v_0 \sqsupseteq v) \wedge \\ \quad (\boldsymbol{obs}' = \boldsymbol{obs}_0 \wedge v' \sqsupseteq v_0) \lhd ok_0 \rhd tr_0 \leq tr' \end{array}\right) & \\
 & \{ref, ref' \text{ are arbitrary, predicate calculus}\} \\
= \boldsymbol{M123}\left(\begin{array}{l} (ok' \wedge wait' = wait \wedge tr' = tr \wedge v' \sqsupseteq v) \\ \lhd ok \rhd \\ \left(\begin{array}{l} tr \leq tr' \wedge (\exists\, ok_0 \bullet ok_0 = ok) \wedge \\ (\exists\, wait_0 \bullet wait_0 = wait) \wedge (\exists\, v_0 \bullet v_0 \sqsupseteq v) \end{array}\right) \end{array}\right) & \\
 & \{\text{let } ok_0 = ok, wait_0 = wait, v_0 = v\} \\
= \boldsymbol{M123}((ok' \wedge wait' = wait \wedge tr' = tr \wedge v' \sqsupseteq v) \lhd ok \rhd tr \leq tr') & \\
 & \{\text{def of } \boldsymbol{M4}\} \\
= \boldsymbol{M1234}(ok' \wedge wait' = wait \wedge tr' = tr \wedge v' \sqsupseteq v) & \{\text{def of } \boldsymbol{M1}\} \\
= \boldsymbol{M234}(ok' \wedge wait' = wait \wedge tr' = tr \wedge v' \sqsupseteq v) & \{\text{def of } \boldsymbol{M2}, \boldsymbol{M3}\} \\
= \bigsqcap_s \boldsymbol{M4}(I\!I \lhd wait \rhd (ok' \wedge wait' = wait \wedge s = s \frown (tr' - tr) \wedge v' \sqsupseteq v)) & \\
 & \{\text{calculation}\} \\
= \boldsymbol{M4}(I\!I \lhd wait \rhd (ok' \wedge \neg\, wait' \wedge tr' - tr = \langle\rangle \wedge v' \sqsupseteq v)) & \{\text{def of } \boldsymbol{M3}\} \\
= \boldsymbol{M34}(ok' \wedge \neg\, wait' \wedge tr' = tr \wedge v' \sqsupseteq v) &
\end{array}
$$

It is clear that *SKIP* satisfies ***M4***.

Theorem 5.4 (SKIP healthy) *SKIP is an* ***MP*** *process.*

Proof*:** *We only need to prove SKIP is* ***M5*. For easier presentation, let*
$A = (ok' \wedge \neg\, wait' \wedge tr' = tr \wedge v' \sqsupseteq v)$
$B = (tr' = tr \wedge ref' = ref \wedge wait' = wait \wedge v' \sqsupseteq v)$

$$
\begin{array}{ll}
SKIP;\ J & \{\text{def of } SKIP, I \text{ and } \boldsymbol{M34}\} \\
= (I \lhd wait \rhd A) \lhd ok \rhd tr \leq tr';\ J & \{;\text{-left-distr-over-}\lhd\rhd, \text{ def of } I\} \\
= \left(\begin{array}{l} ok' \wedge B;\ J \\ \lhd wait \rhd \\ ok' \wedge \neg\, wait' \wedge tr' = tr \wedge v' \sqsupseteq v;\ J \end{array}\right) \lhd ok \rhd (tr \leq tr';\ J) & \\
 & \{\text{def of } ; \text{ and Law 5.4}\} \\
= ((ok' \wedge B) \lhd wait \rhd A) \lhd ok \rhd tr \leq tr' & \{\text{def of conditional}\} \\
= ((ok' = ok \wedge B) \lhd wait \rhd A) \lhd ok \rhd tr \leq tr' & \{\text{def of } \boldsymbol{M34} \text{ and } SKIP\}
\end{array}
$$

$= SKIP$

□

SKIP satisfies the following law.

Law 5.5 (Zero for $I\!I$) $I\!I;\ SKIP = SKIP;\ I\!I = SKIP$

***Proof*:**

$I\!I;\ SKIP$ {def of $I\!I$, ;-distr-left-over-$\lhd\rhd$}
$= (I;\ SKIP) \lhd ok \rhd (tr \leq tr';\ SKIP)$ {Law 5.3, *SKIP* is ***M14***, Theorem 5.1}
$= SKIP \lhd ok \rhd tr \leq tr'$ {*SKIP* is ***M4***}
$= SKIP$
$SKIP;\ I\!I$ {def of *SKIP*, ;-distr-left-over-$\lhd\rhd$}
$= (\boldsymbol{M3}(ok' \wedge \neg\ wait' \wedge tr' = tr \wedge v' \sqsupseteq v);\ I\!I) \lhd ok \rhd (tr \leq tr';\ I\!I)$
{def of ***M3***, ;-distr-left-over-$\lhd\rhd$, $I\!I$ is ***M14***, Theorem 5.1}}
$= ((I\!I;\ I\!I) \lhd wait \rhd (ok' \wedge \neg\ wait' \wedge tr' = tr \wedge v' \sqsupseteq v;\ I\!I))$
$\lhd ok \rhd tr \leq tr'$ {def of ; and $I\!I$}
$= (I\!I \lhd wait \rhd (ok' \wedge \neg\ wait' \wedge tr' = tr \wedge v' \sqsupseteq v;\ I)) \lhd ok \rhd tr \leq tr'$
{def of ;}
$= (I\!I \lhd wait \rhd (ok' \wedge \neg\ wait' \wedge tr' = tr \wedge v' \sqsupseteq v)) \lhd ok \rhd tr \leq tr'$
{def of *SKIP*}
$= SKIP$

□

In standard theory of CSP, *SKIP* is the unit of sequential composition, but this feature cannot be proved from the semantics of *SKIP* and sequential composition. Therefore we specify it by two healthiness conditions ***M6*** and ***M7***.

$$\boldsymbol{M6}(P) \ =\ SKIP\ ;\ P$$
$$\boldsymbol{M7}(P) \ =\ P\ ;\ SKIP$$

Theorem 5.5 (Property of *M6*) ***MP*** *process P satisfies* ***M6 iff***

$$\neg\ wait \Rightarrow (P = \boldsymbol{true}_{\{ref,ref'\}};\ P)$$

***Proof*:**

$P = SKIP;\ P$ {def of *SKIP*, ; closure, P is ***M123***}
$\Leftrightarrow \boldsymbol{M123}(P) = \boldsymbol{M123}(\boldsymbol{true}_{\{ref,ref'\}};\ I\!I;\ P)$ {P is ***M14***, Theorem 5.1}
$\Leftrightarrow \boldsymbol{M123}(P) = \boldsymbol{M123}(\boldsymbol{true}_{\{ref,ref'\}};\ P)$ {def of ***M3***}
$\Leftrightarrow \boldsymbol{M12}(I\!I \lhd wait \rhd P) =$
$\boldsymbol{M12}(I\!I \lhd wait \rhd (\boldsymbol{true}_{\{ref,ref'\}};\ P))$ {def of ***M12***}
$\Leftrightarrow wait \vee (P = \boldsymbol{true}_{\{ref,ref'\}};\ P)$ {propositional law}

$$\Leftrightarrow \neg\ wait \Rightarrow (P = \textbf{\textit{true}}_{\{ref, ref'\}};\ P)$$

□

There is a similar theorem for ***M7***-healthy process.

Theorem 5.6 (Property of *M7*) ***MP*** *process P satisfies* ***M7*** ***iff***

$$\neg\ wait \Rightarrow (P = P;\ \textbf{\textit{true}}_{\{ref, ref'\}})$$

***Proof*:** *Similar to the proof of Theorem 5.5.* □

Theorem 5.5 says that an ***MP*** process does not depend on the initial value of *ref* when *wait* is false. Of course, it must behave as required by ***M3*** when *wait* is *true*. Theorem 5.6 indicates that the value of ref' is irrelevant after the termination of the process.

STOP The deadlock process *STOP* is unable to communicate with its environment and always stays in a waiting state.

Definition 5.11 (STOP)

$$[\![STOP]\!] \ \widehat{=}\ \textbf{\textit{M4}}\left(\begin{array}{l} ok' \wedge wait' \wedge tr' = tr \wedge \\ wait \Rightarrow (ref' = ref \wedge v' \sqsupseteq v) \end{array}\right)$$

□

If *STOP* is started in a divergent state ($ok = true$), then by ***M4***, trace extension ($tr \leq tr'$) is all that we can gurantee about the resulting behaviour. Otherwise, if *STOP* is started in a stable state ($ok = false$), then *STOP* is stable as well ($ok' = true$), it does not terminate ($wait' = true$), and it does not change tr. If *STOP* is started in a state in which its sequential predecssor has not terminated ($wait = true$), then nothing changes including *ref*, but except the allowance of v's improvement.

When *STOP* is started in a state in which its predecessor has terminated ($wait = false$), we cannot affirm any restriction on v and *ref*. This is because the values of v are not observable in a deadlock, and *ref* is arbitrary as *STOP* cannot perform any communication event.

Theorem 5.7 (STOP healthy) *STOP is an* ***MP*** *process.*

Proof*:** *We need to prove STOP is* ***M1235*. For easier presentation, let*
$A = (ok' \wedge wait' \wedge tr' = tr \wedge wait \Rightarrow (ref' = ref \wedge v' \sqsupseteq v))$
$B = (tr' = tr \wedge wait' = wait \wedge ref' = ref \wedge v' \sqsupseteq v)$

M1. Directly from the definition of *STOP*.

M2. Directly from the definition and $(s \leq s \frown (tr' - tr)) \equiv (tr' \leq tr)$.

M3. $I\!I \lhd wait \rhd STOP$ {def of $I\!I$ and *STOP*}

$= (I \lhd ok \rhd tr \leq tr') \lhd wait \rhd (A \lhd ok \rhd tr \leq tr')$ {def of conditional}

$= (I \lhd wait \rhd A) \lhd ok \rhd tr \leq tr'$ {expand I and A}

$$= \left(\begin{array}{l} \left(\begin{array}{l} (ok' \land wait' \land tr' = tr \land ref' = ref \land v' \sqsupseteq v) \\ \lhd wait \rhd (ok' \land wait' \land tr' = tr) \end{array} \right) \\ \lhd ok \rhd tr \leq tr' \end{array} \right)$$

{propositional law}

$= (ok' \land wait' \land tr' = tr \land wait \Rightarrow (ref' = ref \land v' \sqsupseteq v)) \lhd ok \rhd tr \leq tr'$

$= STOP$

M5. $STOP;\ J$ {definition of $STOP$}

$= (A \lhd ok \rhd tr \leq tr');\ J$ {;-distr-left-over-$\lhd\rhd$}

$= (A;\ J) \lhd ok \rhd (tr \leq tr';\ J)$ {def of ; and J, Law 5.4}

$$= \left(\begin{array}{l} \exists\, \boldsymbol{obs}_0, v_0 \bullet \\ \quad ok_0 \land wait_0 \land tr_0 = tr \land wait \Rightarrow (ref_0 = ref \land v_0 \sqsupseteq v) \land \\ \quad ok_0 \Rightarrow ok' \land tr' = tr_0 \land wait' = wait_0 \land ref' = ref_0 \land v' \sqsupseteq v_0 \end{array} \right)$$

$\lhd ok \rhd tr \leq tr'$ {predicate calculus}

$$= \left(\begin{array}{l} \exists\, ref_0, v_0 \bullet ok' \land wait' \land tr' = tr \land \\ \quad ref' = ref_0 \land v' \sqsupseteq v_0 \land wait \Rightarrow (ref_0 = ref \land v_0 \sqsupseteq v) \end{array} \right)$$

$\lhd ok \rhd tr \leq tr'$ {propositional laws, predicate calculus}

$$= \left(\begin{array}{l} ok' \land wait' \land tr' = tr \land \\ \left(\begin{array}{l} \neg\, wait \land (\exists\, ref_0 \bullet ref' = ref_0) \land (\exists\, v_0 \bullet v' \sqsupseteq v_0) \\ \lor\ ref' = ref \land v' \sqsupseteq v) \end{array} \right) \end{array} \right)$$

$\lhd ok \rhd tr \leq tr'$ {let $ref_0 = ref', v_0 = v'$}

$$= \left(\begin{array}{l} ok' \land wait' \land tr' = tr \land (\neg\, wait \lor ref' = ref \land v' \sqsupseteq v) \\ \lhd ok \rhd tr \leq tr' \end{array} \right)$$

{propositional laws, def of $STOP$}

$= STOP$

□

Following ***M3***, as $STOP$ is never terminated ($wait' = true$), its sequential successor is irrelevant and leaves the state unchanged. This feature is captured by a left zero law of sequential composition.

Law 5.6 (STOP left zero) $STOP;\ P = STOP$

***Proof*:**

$STOP;\ P$ {def of ; and $STOP$, P is ***M34***, ;-closure}

$= \boldsymbol{M4}(ok' \land wait' \land tr' = tr \land wait \Rightarrow (ref' = ref \land v' \sqsupseteq v);\ \boldsymbol{M3}(P))$ {def of ***M3***}

$= \boldsymbol{M4}(ok' \land wait' \land tr' = tr \land wait \Rightarrow (ref' = ref \land v' \sqsupseteq v);\ I)$ {def of ;}

$= \boldsymbol{M4}(ok' \land wait' \land tr' = tr \land wait \Rightarrow (ref' = ref \land v' \sqsupseteq v))$ {def of $STOP$}

$= STOP$

□

CHAOS *CHAOS* is the worst ***MP*** process. Except that it satisfies healthiness conditions, we cannot say anything else about its behaviour.

Definition 5.12 (CHAOS)

$$[\![CHAOS]\!] \mathrel{\widehat{=}} \boldsymbol{M123}(\boldsymbol{true})$$ □

We can expand the definition to get an alternative one.

$$\begin{array}{ll}
CHAOS & \{\text{def of } CHAOS \text{ and } \boldsymbol{M3}\} \\
= I\!I \lhd wait \rhd \boldsymbol{M12}(\boldsymbol{true}) & \{\boldsymbol{M1}\} \\
= I\!I \lhd wait \rhd \boldsymbol{M2}(\boldsymbol{true} \land tr \leq tr') & \{\boldsymbol{M2}\} \\
= I\!I \lhd wait \rhd \bigsqcap_s (s \leq s \frown (tr' - tr)) & \\
= I\!I \lhd wait \rhd tr \leq tr' &
\end{array}$$

If *CHAOS* is started in a state in which its predecessor does not terminate ($wait = true$), then nothing changes. For example, $STOP;\ CHAOS = STOP$. Otherwise, it can only guarantee the trace is extended, but it cannot undo the interactions that the predecessor has already been involved. If its predecessor ***MP*** process P terminates but does not change tr, for example, *SKIP* and assignment, then the effect of $P;\ CHAOS$ is the same as *CHAOS*.

Law 5.7 (CHAOS right zero)

(1) $\boldsymbol{M3}(\neg\ wait' \land tr' = tr);\ CHAOS = CHAOS$
(2) *Let Pred be a predicate that does not contain tr, tr' and $wait'$, then*
$\boldsymbol{M3}(Pred \land \neg\ wait' \land tr' = tr);\ CHAOS = CHAOS$

***Proof*:**

$$\begin{array}{ll}
\text{(1) } LHS & \{\text{def of } \boldsymbol{M3}, \text{ ;-distr-left-over-}\lhd\rhd\} \\
= (I\!I;\ CHAOS) \lhd wait \rhd (\neg\ wait' \land tr' = tr;\ CHAOS) & \\
& \{\text{def of } I\!I, \text{ ;-distr-left-over-}\lhd\rhd, \text{ def of } CHAOS\} \\
= \left(\begin{array}{l} (I;\ CHAOS) \lhd ok \rhd (tr \leq tr';\ CHAOS) \\ \quad \lhd wait \rhd \\ (\neg\ wait' \land tr' = tr;\ I\!I \lhd wait \rhd tr \leq tr') \end{array} \right) & \\
& \{\text{Law 5.3, Theorem 5.1, def of ;}\} \\
= (CHAOS \lhd ok \rhd tr \leq tr') \lhd wait \rhd & \\
\quad (\exists\, wait_0, tr_0, \cdots \bullet \neg\ wait_0 \land tr_0 = tr \land I\!I \lhd wait_0 \rhd tr_0 \leq tr') & \\
& \{CHAOS \text{ is } \boldsymbol{M4},\ wait_0 \text{ case analysis, one-point rule}\} \\
= CHAOS \lhd wait \rhd tr \leq tr' & \{\text{def of } CHAOS\} \\
= (I\!I \lhd wait \rhd tr \leq tr') \lhd wait \rhd tr \leq tr' & \{\text{def of conditional}\} \\
= I\!I \lhd wait \rhd tr \leq tr' & \{\text{def of } CHAOS\} \\
= CHAOS &
\end{array}$$

(2) Let $Pred = Pred(\boldsymbol{obs}, v, ok', ref', v')$

LHS {def of ***M3***, ;-distr-left-over-$\lhd\rhd$}

$= (I\!I;\ CHAOS) \lhd wait \rhd (Pred \wedge \neg\ wait' \wedge tr' = tr;\ CHAOS)$

{same as that in (1), def of $CHAOS$}

$= CHAOS \lhd wait \rhd (Pred \wedge \neg\ wait' \wedge tr' = tr;\ I\!I \lhd wait \rhd tr \leq tr')$

{def of $CHAOS$ and conditional, def of ;, one-point rule}

$= I\!I \lhd wait \rhd (\exists\, ok_0, ref_0, v_0 \bullet Pred[ok_0, ref_0, v_0/ok', ref', ok'] \wedge tr \leq tr')$

{substitution}

$= I\!I \lhd wait \rhd ((\exists\, ok', ref', v' \bullet Pred) \wedge tr \leq tr')$ {predicate calculus}

$= I\!I \lhd wait \rhd tr \leq tr'$ {def of $CHAOS$}

$= CHAOS$

□

Theorem 5.8 (CHAOS healthy) *$CHAOS$ is an **MP** process.*

***Proof*:** *Let* $A = (tr' = tr \wedge ref' = ref \wedge wait' = wait \wedge v' \sqsupseteq v)$

M4. $CHAOS \lhd ok \rhd tr \leq tr'$ {def of $CHAOS$}

$= (I\!I \lhd wait \rhd tr \leq tr') \lhd ok \rhd tr \leq tr'$ {expand $I\!I$ when $ok = true$}

$= (I \lhd wait \rhd tr \leq tr') \lhd ok \rhd tr \leq tr'$

$= I \lhd (ok \wedge wait) \rhd tr \leq tr'$

$= (I \lhd ok \rhd tr \leq tr') \lhd wait \rhd tr \leq tr'$

$= I\!I \lhd wait \rhd tr \leq tr'$

$= CHAOS$

M5. $CHAOS;\ J$ {def of $CHAOS$}

$= ((I \lhd ok \rhd tr \leq tr') \lhd wait \rhd tr \leq tr');\ J$ {;-distr-left-over-$\lhd\rhd$}

$= ((I;\ J) \lhd ok \rhd (tr \leq tr';\ J)) \lhd wait \rhd (tr \leq tr';\ J)$

{expand I when $ok = true$}

$= ((ok' \wedge A;\ J) \lhd ok \rhd (tr \leq tr';\ J)) \lhd wait \rhd (tr \leq tr';\ J)$

{def of ; and J, Law 5.4 and Theorem 5.1}

$= ((ok' \wedge A) \lhd ok \rhd tr \leq tr') \lhd wait \rhd tr \leq tr'\ = I\!I \lhd wait \rhd tr \leq tr'$

$= CHAOS$

□

Variable Declaration To introduce a new variable p, we use the form of declaration ***vid*** p: T to permit the variable p of type T to be used in the portion of the program that follows it. Correspondingly, undeclaration ***end*** p terminates the region of permitted use of p. The portion of program P in which the variable p may be used is called its *scope*. In this case, p is called a *local variable* or *bound variable* of P.

The variables presented in the alphabet are *global variables* or *free variables*, and their values can be recorded at the initial state, the intermediate state, and on termination. Local variables represent the internal state, which is inaccessible from outside.

The variable declaration ***vid*** p behaves like *SKIP* except that after the declaration, the variable p becomes an invisible local variable, and is removed from the alphabet.

Definition 5.13 (Variable declaration/undeclaration) *Let* $p, p' \in A$,

$$\begin{array}{rcl} [\![\textbf{\textit{vid}}\ p]\!] & \widehat{=} & \exists\, p \bullet [\![SKIP_A]\!] \\ [\![\textbf{\textit{end}}\ p]\!] & \widehat{=} & \exists\, p' \bullet [\![SKIP_A]\!] \\ \alpha(\textbf{\textit{vid}}\ p) & \widehat{=} & A \setminus \{p\} \\ \alpha(\textbf{\textit{end}}\ p) & \widehat{=} & A \setminus \{p'\} \end{array}$$

where $A \setminus \{p\}$ *is a subset of* A *that contains all elements of* A *except* p. □

In the above definition, existential quantifier removes the related variable from alphabet A. This has already been shown in the definition of sequential composition, where the intermediate value of the variables are hidden by an existential quantifier over the intermediate variables. More generally, we have

$$\exists\, p \bullet Q_A \ =\ Q_{A\setminus\{p\}}$$

Alternately, variable declaration and undeclaration can be defined as

$$\begin{array}{rcl} [\![\textbf{\textit{vid}}\ p]\!] & \widehat{=} & [\![SKIP_{A\setminus\{p\}}]\!] \\ [\![\textbf{\textit{end}}\ p]\!] & \widehat{=} & [\![SKIP_{A\setminus\{p'\}}]\!] \end{array}$$

As variable declaration and undeclaration are not homogenous, we need be care when use it in the conditional. They are not allowed to be used in the conditional separately except that the same variable declaration or undeclaration appears in both choice branches, or they appear as a pair in one branch. For instance, we do not allow $(\textbf{\textit{vid}}\ x;\ P) \lhd b \rhd Q$ if ***vid*** x does not occur in Q and ***end*** x does not occur in P.

For more convenient presentation, we allow variables to be declared together

$$\begin{array}{lll} \textbf{\textit{vid}}\ p, q, .., r & \text{rather than} & \textbf{\textit{vid}}\ p;\ \textbf{\textit{vid}}\ q;\ \cdots;\ \textbf{\textit{vid}}\ r \\ \textbf{\textit{end}}\ p, q, .., r & \text{rather than} & \textbf{\textit{end}}\ p;\ \textbf{\textit{end}}\ q;\ \cdots;\ \textbf{\textit{end}}\ r \end{array}$$

and we enclose a variable by a block. We abbreviate $(\textbf{\textit{proc}}\ h;\ Q;\ \textbf{\textit{end}}\ h)$ by $(\textbf{\textit{proc}}\ h \bullet Q)$ to represent that the scope of h is valid in Q.

Theorem 5.9 (Variable declaration healthy) *Variable declaration* ***vid*** p *and undeclaration* ***end*** p *are* ***MP*** *processes.*

***Proof*:** *Direct from Definition 5.13 and Theorem 5.4.* □

Assignment After the declaration of a variable, it can be initialised by being the target of an assignment or a communication. The assignment terminates immediately, with the value of its target variable updated.

Higher-order constant assignment $h := \{\!|P|\!\}$ is defined as Definition 5.9. It is also suitable for first-order constant assignment. The alphabet of $(h := \{\!|P|\!\})$ does not include αP because P is a constant in the assignment, thus the αP is not relevant unless h is activated.

Mobile variable assignment makes a value, which is a data or a process, move from the source variable to the target one. On termination of the assignment $p :=_m q$, the source variable q is undefined, and has an arbitrary process value. However, it is still in the scope, and can be referred again if reinitialised.

Definition 5.14 (Mobile variable assignment) *Let* $\alpha p = \alpha q,\ p, p' \notin \alpha q$

$$\begin{aligned} [\![p :=_m q]\!] &\mathrel{\widehat{=}} \textbf{\textit{M34}}\left(\begin{array}{l} ok' \wedge \neg\ wait' \wedge (tr' = tr) \\ \wedge\ (p' \sqsupseteq q) \wedge (v' \sqsupseteq v) \end{array}\right) \\ \alpha(p :=_m q) &\mathrel{\widehat{=}} \{\textbf{\textit{obs}}, \textbf{\textit{obs}}', p, p', q, q', v, v'\} \end{aligned}$$

where v *is the set of program variables except* p *and* q. □

In the above definition, q' is in the alphabet but not mentioned in the predicate so that we can view its value is arbitrary. Alternatively, we can specify this fact by $q' \sqsupseteq CHAOS$ which is always true.

Similar to the constant assignment, the clone variable assignment terminates immediately and updates the target variable in the light of the value of the source variable. Different from mobile assignment, the source variable is still defined on the termination of clone assignment.

Definition 5.15 (Clone variable assignment) *Let* $\alpha p = \alpha q,\ p, p' \notin \alpha q$

$$\begin{aligned} [\![p := q]\!] &\mathrel{\widehat{=}} \textbf{\textit{M34}}\left(\begin{array}{l} ok' \wedge \neg\ wait' \wedge (tr' = tr) \\ \wedge\ (p' \sqsupseteq q) \wedge (q' \sqsupseteq q) \wedge (v' \sqsupseteq v) \end{array}\right) \\ \alpha(p := q) &= \{\textbf{\textit{obs}}, \textbf{\textit{obs}}', p, p', q, q', v, v'\} \end{aligned}$$

where v *is the set of program variables except* p *and* q. □

Theorem 5.10 (Assignment healthy) *Assignment are* ***MP*** *processes.*

***Proof*:** *Similar to the proof of Theorem 5.4* □

We allow assignment to be combined with declaration. Usually we write ***proc*** $h := \{\!|Q|\!\}$ instead of ***proc*** $h;\ h := \{\!|Q|\!\}$.

Variable activation Once initialised, a process variable h may be activated by executing its process constant value; therefore, a process can be replaced by the activation of a variable which has been assigned by this process constant before.

Law 5.8 (Copy-rule-1) *Let* $h, h' \notin \alpha Q$, *then*

$$(h := \{\!|Q|\!\};\ \langle h \rangle) = (h := \{\!|Q|\!\};\ Q)$$

***Proof*:**

$$
\begin{array}{ll}
RHS & \{\text{meaning of } \langle h \rangle \text{ and Def. 5.9}\} \\
= h := \{\!|Q|\!\};\ \sqcap\{R \mid R \sqsupseteq Q\} & \{\text{Def. 5.22}\} \\
= h := \{\!|Q|\!\};\ R_1 \vee \cdots \vee R_n \vee Q & \\
 & \{R_i \sqsupseteq Q, \text{ Def. 5.1, propositional calculus}\} \\
= LHS &
\end{array}
$$

where process R_i refines Q and n can be infinite. □

Even though we adopt inequality in higher-order assingment, this rule is an equality rather than a refinement from the proof.

If a parameterised process variable h has the value of

$$\lambda\, x : var(T_1), y : val(T_2), z : res(T_2) \bullet P$$

then the effect of the activation $h(ne, ve, re)$ is calculated by

$$h(ne, ve, re) \mathrel{\widehat{=}} \left(\begin{array}{l} \textbf{\textit{vid}}\ x := ne, y := ve, z; \\ P; \\ ne := x, re := z; \\ \textbf{\textit{end}}\ x, y, z \end{array} \right)$$

It initially assigns the values of actual parameters ne and ve to the formal name and value parameters of P, and executes P. On the termination of P, the values of the name and result parameters are passed back to the actual parameters ne and re.

Communication Same as higher-order assignment, higher-order communication is monotonic in the communicated value. If one communicated process Q is better than another one P, then the communication system with Q should be better than that with P.

The communication $ch.E$ is stable while waiting for synchronisation of its two communicating parties ($ch!$ or $ch!!$ and $ch?$). As soon as both of them are ready, the communication takes place to perform any communication event $ch.M$, with its communicated value M better than E, and then terminates, allowing the program variables to improve.

Definition 5.16 (Higher order communication)

$$
\begin{array}{l}
[\![ch.E]\!] \mathrel{\widehat{=}} \\
\sqcap_M \left\{ \textbf{\textit{M34}} \left(\begin{array}{l} wait' \wedge ch \notin ref' \wedge tr' = tr \wedge ok' \wedge v' \sqsupseteq v \\ \vee \\ \neg\ wait' \wedge tr' = tr \frown \langle ch.M \rangle \wedge ok' \wedge v' \sqsupseteq v \end{array} \right) \middle| M \sqsupseteq E \right\}
\end{array}
$$

where $\sqcap_M$ represents an non-deterministic choice over the process M, which will be defined in a later section, and $ch \notin ref'$ represents that all the communication events over ch are not in the refusal set. □

This definition is also suitable for first-order communication $ch.e$. When the communicated value is first-order data e, the internal choice will result in e itself, as M can only equate e for first-order data.

Theorem 5.11 (Communication monotonicity) *Communication is monotonic in the communicated value.*

$$(P \sqsubseteq Q) \;\Rightarrow\; (ch.P \sqsubseteq ch.Q)$$

***Proof*:** *Similar to the proof of Theorem 5.3.* □

By the input prefix $ch?p \rightarrow P$, variable p accepts a message from the channel and then the program behaves like P.

Definition 5.17 (Input prefix)

$$ch?p \rightarrow P \;\widehat{=}\; \Box_{E \in Mes} \bullet (ch.E;\; p := E;\; P)$$

where Mes is the set of all messages (data or processes) that can be transferred on the channel ch. External choice $\Box$ is defined in a later subsection. □

The clone output prefix $ch!q \rightarrow P$ represents a program which is willing to transfer the value of process variable q through channel ch and then behaves as P. Mobile output prefix $ch!!q \rightarrow P$ has the same meaning of $ch!q \rightarrow P$ except that output variable q is undefined after the transfer.

Definition 5.18 (Clone output prefix)

$$ch!q \rightarrow P \;\widehat{=}\; ch.q;\; P$$ □

Definition 5.19 (Mobile output prefix)

$$ch!!q \rightarrow P \;\widehat{=}\; \sqcap_M \left\{ \begin{array}{l} \textbf{\textit{M34}} \left(\begin{array}{l} wait' \wedge ch \notin ref' \wedge tr' = tr \wedge ok' \wedge v' \sqsupseteq v \wedge q' \sqsupseteq q \\ \vee\, (\neg\, wait' \wedge tr' = tr \frown \langle ch.M \rangle \wedge ok' \wedge v' \sqsupseteq v) \end{array} \right) \\ \mid M \sqsupseteq q \end{array} \right\} ;\; P$$

where v are all program variables except q. □

Theorem 5.12 (Communication healthy) *Communication, input prefix and output prefix are **MP** processes.*

***Proof*:** *Similar to the proof of Theorem 5.4.* □

5.4.2 Parallel composition

The parallel composition $P \parallel Q$ executes P and Q concurrently such that the events in the alphabet of both parties require their simultaneous participation, whereas the remaining events of the system occur independently. Event synchronisation is the only way in which one parallel process can affect its partner. The system terminates after both P and Q have terminated successfully, and it becomes divergent after either one of its components does so. An event can be refused by a parallel composition if either parallel party can refuse it.

We only take the observation variables $\boldsymbol{obs}$ to be the shared write variables between two parallel processes. Same as the parallel constraint in occam [2], shared write program variables are not considered in our language — a variable that is written in one component of the parallel may not appear in any other component of the parallel. For data variables, their values can be updated as the input of a communication or the target of an assignment, or as the output of a mobile communication or the source of an mobile assignment. For process variables, all of them are write variables. This is because the assignment to a variable p allows the improvement of all process variables v, and other primitive processes allow the improvement of all process variables. Therefore the read variables of a process are only those first-order variables that are not the source of an mobile assignment/communication or not the target of any assignment/communication. There is another constraint for process variables. When a process variable is in activation, it cannot be activated in parallel, or assigned to/from another variable, or communicated as input or output.

Let $Var(P)$ denote all program variables employed by process P, and $RdVar(P)$ denote a subset of $Var(P)$ whose values are never modified by P. The parallel composition is defined as:

Definition 5.20 (Parallel composition) *Suppose that* $Var(P) \cap Var(Q) = RdVar(P) \cap RdVar(Q)$*, then*

$$
\begin{array}{rcl}
\mathcal{A}(P \parallel Q) & \widehat{=} & \mathcal{A}P \cup \mathcal{A}Q \\
[\![P \parallel Q]\!] & \widehat{=} & [\![(P(\boldsymbol{obs}, 1.\boldsymbol{obs}')]\!] \wedge [\![Q(\boldsymbol{obs}, 2.\boldsymbol{obs}')]\!];\ M(1.\boldsymbol{obs}, 2.\boldsymbol{obs}, \boldsymbol{obs}') \\
M & \widehat{=} & \left(\begin{array}{l} ok' = (1.ok \wedge 2.ok) \wedge \\ wait' = (1.wait \vee 2.wait) \wedge \\ ref' = (1.ref \cup 2.ref) \wedge \\ (tr' - tr) = (1.tr - tr) \parallel (2.tr - tr) \end{array} \right);\ [\![SKIP]\!]
\end{array}
$$

□

The trace merge function $s \parallel t$ is defined the same as that in [24], in which $\{a, b\} \notin E(s) \cap E(t), \{c, d\} \in E(s) \cap E(t), a \neq b, c \neq d$, where $E(s)$ is the set

of events in s.

$$
\begin{array}{rcl}
s \parallel t & \widehat{=} & t \parallel s \\
\langle\rangle \parallel \langle\rangle & \widehat{=} & \{\langle\rangle\} \\
\langle\rangle \parallel \langle c\rangle & \widehat{=} & \{\ \} \\
\langle\rangle \parallel \langle a\rangle & \widehat{=} & \{\langle a\rangle\} \\
\langle c\rangle \frown x \parallel \langle c\rangle \frown y & \widehat{=} & \{\langle c\rangle \frown u \mid u \in x \parallel y\} \\
\langle a\rangle \frown x \parallel \langle c\rangle \frown y & \widehat{=} & \{\langle a\rangle \frown u \mid u \in x \parallel \langle c\rangle \frown y\} \\
\langle c\rangle \frown x \parallel \langle d\rangle \frown y & \widehat{=} & \{\ \} \\
\langle a\rangle \frown x \parallel \langle b\rangle \frown y & \widehat{=} & \{\langle a\rangle \frown u \mid u \in x \parallel \langle b\rangle \frown y\} \\
 & & \cup \{\langle b\rangle \frown u \mid u \in \langle a\rangle \frown x \parallel y\}
\end{array}
$$

In the definition of parallel composition, $Var(P) \cap Var(Q) = RdVar(P) \cap RdVar(Q)$ excludes the possibility of shared write program variables between two parallel components. M is a predicate that merges the final values of observable variables $1.\boldsymbol{obs}$ and $2.\boldsymbol{obs}$ produced by the two processes respectively; the sequential composition with *SKIP* deals with the situation when either parallel process is divergent or waiting.

Let us consider $(A;SKIP)$, where A only merges the final values of observable variables produced by the two parallel processes.

If either parallel party is divergent, indicated by $ok'_A = false$, then according to the definitions of sequential composition and *SKIP* (***M4*** healthy), all that we can guarantee about the behaviour of *SKIP* is trace extension and we cannot predict the final value of any other observation variables. Therefore the parallelism may not keep the good behaviours of the other process which is not divergent. Moreover, as *SKIP* is ***M5*** healthy and it is the unit of sequential composition, sequentially composing *SKIP* makes the parallelism ***M5*** healthy as well.

If either parallel party is waiting, indicated by $wait'_A = true$, then *SKIP* behaves as required by ***M3*** and keep the behaviours unchanged, such that the parallel composition does not terminate as well. If both parallel processes terminate, then $wait'_A = false$. According to the definition of *SKIP*, this indicates that the $A;SKIP$ terminates and the final value of *ref* is irrelevant after termination, making the parallel composition ***M7*** healthy.

Theorem 5.13 ($\parallel$-closure) *The set of healthy processes is closed under the parallel composition* $\parallel$. □

5.4.3 Iteration

If b is a condition, the notation $b * P$ repeats the process P as long as b is *true* before each iteration. More formally, it can be defined as the recursion.

Definition 5.21 (iteration)

$$
\begin{array}{rcl}
[\![b * P]\!] & \widehat{=} & \mu\, X \bullet (([\![P]\!];\ X) \lhd b \rhd [\![SKIP]\!]) \\
\alpha(b * P) & \widehat{=} & \alpha P
\end{array}
$$

where $\mu\, X \bullet F(X)$ *stands for the weakest fixed point [31] of the recursive equation* $X = F(X)$. □

5.4.4 Internal choice

The internal choice $P \sqcap Q$ stands for a program which is executed by performing either P or Q, but the choice between them is made arbitrarily and nondeterministically.

Definition 5.22 (Internal choice) *Provided* $\alpha P = \alpha Q$, *then*

$$\begin{array}{rcl} [\![P \sqcap Q]\!] & \widehat{=} & [\![P]\!] \vee [\![Q]\!] \\ \alpha(P \sqcap Q) & \widehat{=} & \alpha P \end{array}$$ □

Alphabet restriction is required that the alphabets of two processes should be the same, otherwise the internal choice has no meaning. The reason is the same as the alphabet restriction of conditional choice.

5.4.5 External choice

The external choice $P \,\Box\, Q$ behaves as either one of the two processes, where the choice is made by the environment when it performs one of the first interactions or the first terminations without event offered by P and Q. If the first interaction or termination is the one in which Q has performed, then the rest of the behaviours of $P \,\Box\, Q$ is described by Q. Similarly, P can be selected by occurrence of an interaction possible for P but not for Q. Finally, if the first event is possible for both P and Q, then the choice between them is nondeterministic.

Definition 5.23 (External choice) *Provided* $\mathcal{A}P = \mathcal{A}Q$, *then*

$$[\![P \,\Box\, Q]\!] \;\widehat{=}\; \boldsymbol{M5}(([\![P]\!] \wedge [\![Q]\!]) \lhd [\![STOP]\!] \rhd ([\![P]\!] \vee [\![Q]\!]))$$ □

The definition also says that before P and Q communicate with the environment and the choice has not been made, which is indicated by $STOP = true$, the observation is agreed by both P and Q, and only internal interaction can take place, and an event can be refused by $P \,\Box\, Q$ just if it can be refused by both of them.

The purpose of applying healthiness condition ***M5*** is to deal with divergent behaviour. When $\neg\, STOP$ is selected, according to the definition of $STOP$ and conditional, [3]

$$\neg\, STOP = \left(\begin{array}{l} ok \wedge (\neg\, ok' \vee \neg\, wait' \vee tr' \neq tr \vee \neg\, (wait \Rightarrow ref' = ref \wedge v' \sqsupseteq v)) \\ \vee \\ \neg\, ok \wedge \neg\, tr \leq tr' \\ \vee \\ \neg\, tr \leq tr' \wedge \left(\begin{array}{l} \neg\, ok' \vee \neg\, wait' \vee tr' \neq tr \\ \vee \neg\, (wait \Rightarrow ref' = ref \wedge v' \sqsupseteq v) \end{array} \right) \end{array} \right)$$

[3] $\neg\, (P \lhd b \rhd Q) = (b \wedge \neg\, P) \vee (\neg\, b \wedge \neg\, Q) \vee (\neg\, P \wedge \neg\, Q)$

By applying ***M5***, we cannot insist that $P \Box Q$ be divergent even when ok' is *false*.

The reason of the alphabet restriction is the same as that for conditional and internal choice.

Theorem 5.14 ($\Box$-healthy) *The set of healthy processes is closed under the external choice $\Box$.* □

5.4.6 Hiding

Let X be a set of events that are regarded as internal events of a process P. We use $P \setminus X$ to denote the process in which the events of X occur silently without participation or even the knowledge of the environment of P. $P \setminus X$ reaches a stable state only when P is stable and cannot perform any event of X. It reaches divergent if P does and it terminates when P has terminated.

Definition 5.24 (Hiding)

$$\begin{array}{rcl} \mathcal{A}(P \setminus X) & \widehat{=} & \mathcal{A}P - X \\ [\![P \setminus X]\!] & \widehat{=} & \exists\, tra, refa \bullet \left(\begin{array}{l} [\![P]\!][tra/tr', refa/ref'] \wedge \\ tr' = tra \downarrow (\mathcal{A}P - X) \wedge \\ refa = ref' \cup X \end{array} \right) ;\ SKIP \end{array}$$

□

Similar to that in the definition of parallel composition, the sequential composition with *SKIP* makes the definition of hiding healthy.

When we want to hide some communication events, instead of writing the communication events in X, we simply use a channel set CH to denote hiding all the communication on the channels in CH.

Theorem 5.15 (Hiding closure) *The set of healthy processes is closed under the hiding $\setminus X$.* □

6 Algebraic Properties and Laws

Besides some algebraic laws for *SKIP*, *STOP* and *CHAOS* in the previous section, we will present more algebraic properties and laws for other ***MP*** processes in this section. The proofs of these laws are based on the UTP semantics of mobile processes, some of which are largely straightforward and are omitted.

6.1 Variable declaration and undeclaration

As we have seen in the previous chapter, higher-order variables play an important role in our work of mobile processes, because process mobility is exhibited in mobile variable assignment and communication. In this subsection, we will present and prove a series of laws of variable declaration and undeclaration.

In spite of the changed definition of identity and some other operators, these laws are very similar to their counterparts in conventional non-mobile first-order programming.

Law 6.1 (Sequential declaration commutative) *The sequential composition of two variable declarations or undeclarations are commutative.*

(1) $(\textbf{\textit{vid}}\ p;\ \textbf{\textit{vid}}\ q) = (\textbf{\textit{vid}}\ q;\ \textbf{\textit{vid}}\ p) = (\textbf{\textit{vid}}\ p, q)$
(2) $(\textbf{\textit{end}}\ p;\ \textbf{\textit{end}}\ q) = (\textbf{\textit{end}}\ q;\ \textbf{\textit{end}}\ p) = (\textbf{\textit{end}}\ p, q)$
(3) $(\textbf{\textit{vid}}\ p;\ \textbf{\textit{end}}\ q) = (\textbf{\textit{end}}\ q;\ \textbf{\textit{vid}}\ p)$ $\{p$ and q are distinct$\}$

□

Law 6.2 (Variable elimination/intorduction) *If a declared program variable is never used or initialised, its declaration has no effect.*

$$(\textbf{\textit{vid}}\ p\ ;\ Q_{+p}\ ;\ \textbf{\textit{end}}\ p)\ =\ Q \qquad \{p, p' \notin \alpha Q\}$$

Alphabet extension is needed in order to balance alphabets.

***Proof*:**

LHS	{Definition 5.13}
$= \exists\, p \bullet SKIP_A;\ Q;\ \exists\, p' \bullet SKIP_A$	{predicate calculus}
$= \exists\, p \bullet (SKIP_A;\ Q);\ \exists\, p' \bullet SKIP_A$	{***M6***}
$= \exists\, p \bullet Q;\ \exists\, p' \bullet SKIP_A$	{$p \notin \alpha Q$, predicate calculus}
$= Q;\ \exists\, p' \bullet SKIP_A$	{predicate calculus, $p' \notin \alpha Q$}
$= \exists\, p' \bullet (Q;\ SKIP_A)$	{***M7***}
$= \exists\, p' \bullet Q$	{$p' \notin \alpha Q$}
$= RHS$	

□

Law 6.3 $(\textbf{\textit{vid}}\ p;\ \textbf{\textit{end}}\ p) = SKIP$ □

Similarly, if an initialised program variable is never used, its initialisation has no effect.

Law 6.4 $(\textbf{\textit{proc}}\ h;\ h := \{\!|Q|\!\};\ \textbf{\textit{end}}\ h) = SKIP$ □

But if the program variable is initialised by a mobile assignment or a mobile communication, its initialisation changes the value of the source variable to arbitrary, therefore the whole program is refined by $SKIP$.

Law 6.5 $(\textbf{\textit{vid}}\ p;\ p :=_m q;\ \textbf{\textit{end}}\ p) \sqsubseteq SKIP$ □

By applying the above three laws, we can eliminate a non-useful variable in the derication process, or we can include a process in the scope of a new variable, assuming this variable is not free in the process.

The following law states that, upon declaration, a variable may take any value within its type, the choice being arbitrarily non-deterministic, and this variable remains unchanged until it is assigned or input to.

Law 6.6 (Initial value arbitrary) $(\textbf{\textit{proc}}\ h) \ = \ \textbf{\textit{proc}}\ h := \{\!|CHAOS|\!\}$

***Proof*:**

$$
\begin{array}{ll}
RHS & \{\text{syntax}\} \\
= \textbf{\textit{proc}}\ h;\ h := \{\!|CHAOS|\!\} & \{\text{Definition 5.13 and 5.9}\} \\
= (\textbf{\textit{M34}}(ok' \wedge \neg\ wait' \wedge tr' = tr \wedge v' \sqsupseteq v))_{A\setminus\{h\}}; & \\
\quad (\textbf{\textit{M34}}(ok' \wedge \neg\ wait' \wedge tr' = tr \wedge h' \sqsupseteq CHAOS \wedge v' \sqsupseteq v))_{A} & \\
\qquad \{h' \text{ in the alphabet but not mentioned in the first predicate}\} & \\
= (\textbf{\textit{M34}}(ok' \wedge \neg\ wait' \wedge tr' = tr \wedge v' \sqsupseteq v \wedge \textbf{\textit{true}}_{h'}))_{A\setminus h}; & \\
\quad (\textbf{\textit{M34}}(ok' \wedge \neg\ wait' \wedge tr' = tr \wedge \textbf{\textit{true}}_{h'} \wedge v' \sqsupseteq v))_{A} & \{\text{;-closure}\} \\
= \left(\textbf{\textit{M34}}\left(\begin{array}{l} ok' \wedge \neg\ wait' \wedge tr' = tr \wedge v' \sqsupseteq v \wedge \textbf{\textit{true}}_{h'}; \\ ok' \wedge \neg\ wait' \wedge tr' = tr \wedge v' \sqsupseteq v \wedge \textbf{\textit{true}}_{h'} \end{array}\right)\right)_{A\setminus\{h\}} & \{\text{def of ;}\} \\
= (\textbf{\textit{M34}}(ok' \wedge \neg\ wait' \wedge tr' = tr \wedge v' \sqsupseteq v \wedge \textbf{\textit{true}}_{h'}))_{A\setminus\{h\}} & \{\text{def of } SKIP\} \\
= SKIP_{A\setminus h} & \\
= LHS &
\end{array}
$$

□

As variable declaration ***proc*** h assigns an arbitrary value to this variable, any assignment to this variable can make it more deterministic.

Law 6.7 (Initial value refinement) $(\textbf{\textit{proc}}\ h) \sqsubseteq (\textbf{\textit{proc}}\ h := \{\!|Q|\!\})$ □

The next law states that the sequential composition of ***end*** p with ***vid*** p has no effect whenever it is followed by an update of p that does not depend on the previous value of p.

Law 6.8 (Vacuous declaration) *If the type of q and p are same, and p does not occur in Q and q, then*

(1) $(\textbf{\textit{end}}\ p;\ \textbf{\textit{proc}}\ p := \{\!|Q|\!\}) = (p := \{\!|Q|\!\})$
(2) $(\textbf{\textit{end}}\ p;\ \textbf{\textit{vid}}\ p := q) = (p := q)$ □

As a special case of the above law, the sequential composition of variable undeclaration with the same variable declaration changes the value of this variable to arbitrary.

Lemma 6.1 $(\textbf{\textit{end}}\ p;\ \textbf{\textit{proc}}\ p) = (p := \{\!|CHAOS|\!\})$ □

Obviously from the above lemma, the sequential composition of ***end*** p with ***proc*** p is refined by $SKIP$.

Law 6.9 $(\textbf{\textit{end}}\ p;\ \textbf{\textit{vid}}\ p) \sqsubseteq SKIP$ □

When the same variable is declared in two sequential programs, the two declarations may be replaced by a single one.

Law 6.10 $(\textbf{\textit{proc}}\ p;\ P;\ \textbf{\textit{end}}\ p;\ \textbf{\textit{proc}}\ p;\ Q;\ \textbf{\textit{end}}\ p) \sqsubseteq (\textbf{\textit{proc}}\ p;\ P;\ Q;\ \textbf{\textit{end}}\ p)$

Proof*:** *direct from Law 6.9 and* ***M6*.* □

The above law may reduce non-determinism. Suppose Q is $q :=_m p$. On the left hand side, the final value of q is arbitrary. On the right hand side, however, p may be assigned a certain value in P, thus the final value of q would be a refinement of p. The right hand side is at least as deterministic as the left one.

Law 6.11 (Variable scope conditional distributivity) *Variable scope is distributed over conditional, if no interference occurs with the condition.*

$$(\textbf{\textit{vid}}\ p;\ P \lhd b \rhd Q;\ \textbf{\textit{end}}\ p)$$
$$= (\textbf{\textit{vid}}\ p;\ P;\ \textbf{\textit{end}}\ p) \lhd b \rhd (\textbf{\textit{vid}}\ p;\ Q;\ \textbf{\textit{end}}\ p)\ \{p \text{ not occur in } b\}$$

□

Law 6.12 (Variable scope extension/shrinkage) *The scope of a variable may be extended by moving variable declaration in front of a process that contains no free occurrences of it, or moving variable undeclaration after this process.*

$$(1)\ (P;\ \textbf{\textit{vid}}\ p) = (\textbf{\textit{vid}}\ p;\ P_{+p})$$
$$(2)\ (\textbf{\textit{end}}\ p;\ P) = (P_{+p};\ \textbf{\textit{end}}\ p) \qquad \{p, p' \notin \alpha P\}$$

***Proof*:** *We only show the proof of (1). (2) can be proved in the same way.*

$$\begin{aligned}
&LHS && \{\text{Definition 5.13}\}\\
&= P;\ \exists\, p \bullet SKIP_A && \{p \notin \alpha P\}\\
&= \exists\, p \bullet (P;\ SKIP_A) && \{\textbf{\textit{M67}}\}\\
&= \exists\, p \bullet (SKIP_A;\ P) && \{p \notin \alpha P\}\\
&= (\exists\, p \bullet SKIP_A);\ P && \{\text{Definition 5.13}\}\\
&= RHS
\end{aligned}$$

□

If we look at the above laws from the other way around, we can see the scope of the variable p is shrunk.

A variable q can be replaced by a new one p, by introducing the new variable p, which is not declared in the block of q, and assigning q to p.

Law 6.13 (Local variable renaming) *If Q contains occurrences of q but no occurrence of p, then*

$$(Q;\ \textbf{\textit{end}}\ q) = (\textbf{\textit{vid}}\ p;\ p :=_m q;\ Q[p/q];\ \textbf{\textit{end}}\ p, q)$$

***Proof*:**

$$\begin{array}{ll} & \\ RHS & \{\text{Definition 5.14}\} \\ = \textbf{\textit{vid}}\ p;\ p := q;\ Q;\ \textbf{\textit{end}}\ p, q & \{\text{Law 6.12}\} \\ = \textbf{\textit{vid}}\ p;\ p := q;\ \textbf{\textit{end}}\ p;\ Q;\ \textbf{\textit{end}}\ q & \{\text{Law 6.4}\} \\ = SKIP;\ Q;\ \textbf{\textit{end}}\ q & \{\textbf{\textit{M6}}\} \\ = LHS & \end{array}$$

□

If a declared process variable does not occur in one of the partners of its successive parallel composition, its declaration does not affect this process.

Law 6.14 (Variable scope parallel distributivity) *Suppose Q contains no occurrence of p, and P contains no occurrence of q, then*

$$\textbf{\textit{vid}}\ p, q;\ (P \parallel Q);\ \textbf{\textit{end}}\ p, q\ =\ (\textbf{\textit{vid}}\ p;\ P;\ \textbf{\textit{end}}\ p) \parallel (\textbf{\textit{vid}}\ q;\ Q;\ \textbf{\textit{end}}\ q)$$

□

In the procedure to derive a distributed system from a centralised one, we can apply this law to narrow the scope of a variable to its relevant process, therefore the variable declaration can be bounded together with this process when the centralised specification is split.

6.2 Assignment

The value of a process variable is viewed as a process constant, and is not subject to substitutions for the variables it contains.

Law 6.15 (Process constant)

$$(P(x)_{+h}\ ;\ h := \{\!|Q(x)|\!\}) \ =\ (h := \{\!|Q(x)|\!\}\ ;\ P(x)_{+h}) \qquad \{h, h' \notin \alpha P,\ h, h' \notin \alpha Q\}$$

where $P(x)$ and $Q(x)$ are processes in which x occurs as a variable. □

The assignment of a constant to a process variable at the end of its scope is vacuous. In the other words, if the value of a variable is updated but never used, this update is irrelevant.

Law 6.16 (Vacuous constant assignment)

$$(h := \{\!|Q|\!\} \ ; \ \textbf{\textit{end}}\ h) \ = \ (\textbf{\textit{end}}\ h) \qquad \{h, h' \notin \alpha Q\}$$

□

The mobile assignment $(h :=_m g \ ; \ \textbf{\textit{end}}\ h)$ would not have been vacuous, since the effect on g (making its value undefined) would persist beyond the end of h's scope. The corresponding law for mobile variable assignment is stronger in that it requires both variable scopes to end.

Law 6.17 (Vacuous mobile variable assignment)

$$(p :=_m q \ ; \ \textbf{\textit{end}}\ p \ ; \ \textbf{\textit{end}}\ q) \ = \ (\textbf{\textit{end}}\ p \ ; \ \textbf{\textit{end}}\ q) \qquad \square$$

The effect of the assignment of a variable to itself is the same as *SKIP*.

Law 6.18 (Vacuous identity assignment)

$$(p := p) \ = \ (p :=_m p) \ = \ SKIP \qquad \square$$

When the mobile assignment terminates, on the right hand side, p's value becomes arbitrary; however, on the left hand side p gets a value that refines the original value of right-hand-side p. Therefore, the whole conjunctive effect is that p refines its original value, which is the same as *SKIP*.

Any non-identity mobile assignment can be converted to clone assignment.

Law 6.19 (Mobile assignment to clone assignment)

$$(p :=_m q) = (p := q; \ \textbf{\textit{end}}\ q; \ \textbf{\textit{vid}}\ q) \qquad \{p \text{ and } q \text{ are distinct}\}$$

□

In spite of the introduction of mobile assignment, any kind of assignment can be reduced to assignment normal form – total non-mobile assignment – in which all the variables of the program appear on the left hand side in some standard order, and their related values are shown on the right hand side,

$$h, g, ...r := \{\!|P|\!\}, \{\!|Q|\!\}, ...\{\!|R|\!\}$$

by transforming mobile assignment to non-mobile assignment and adding identity clone assignment.

Law 6.20 (Assignment normal form)

$$(h :=_m g) = (h, g, ..., r := g, \{\!|CHAOS|\!\}, ..., r)$$
$$\{g \text{ and } h \text{ are distinct process variables}\}$$
$$(p := q) = (p, q, ..., r := q, q, ..., r) \qquad \{p \text{ and } q \text{ are distinct variables}\}$$

□

Successive assignments to the same variable can be reduced to one.

Law 6.21 (Assignment mergence) *If g and h are process variables, and Q and R are process constants, then*

$$(1)\ (h := \{\!|Q|\!};\ h := F(h)) = (h := F(Q))$$
$$(2)\ (h := g;\ h := F(h)) = (h := F(g))$$
$$(3)\ (h := \{\!|Q|\!};\ h := \{\!|R|\!}) = (h := \{\!|R|\!}) \qquad \{R \text{ not refer to } h\}$$

□

Law 6.22 (Assignment sequence)

$$(q := p;\ p := f(q)) = (q := p;\ p := f(p))$$ □

CHAOS is a right zero of sequential composition of assignment.

Law 6.23 (Assignment right zero) *If P is an assignment defined by Definition 5.9, 5.14 or 5.15, then*

$$P;\ CHAOS = CHAOS$$

***Proof*:** *Directly from the definition of assignment and Law 5.7.* □

6.3 Process variable activation

Any intent to activate an uninitialised process variable leads to chaos.

Law 6.24 (Uninitialised activation)

$$(\textbf{\textit{proc}}\ h;\ <h>) = CHAOS$$

***Proof*:**

$$\begin{aligned}
&LHS && \{\text{Law 6.6}\}\\
&= \textbf{\textit{proc}}\ h := \{\!|CHAOS|\!};\ <h> && \{\text{Definition 5.13 and 5.9}\}\\
&= SKIP_{A\setminus h};\ h := \{\!|CHAOS|\!};\ \sqcap_R\{R \mid R \sqsupseteq CHAOS\} && \\
& && \{\text{set theory, Def. 5.22}\}\\
&= SKIP_{A\setminus h};\ (h := \{\!|CHAOS|\!};\ CHAOS) && \{\text{Law 6.23}\}\\
&= SKIP_{A\setminus h};\ CHAOS && \{\textbf{\textit{M6}}\}\\
&= RHS &&
\end{aligned}$$

□

The mobility of processes is expressed in the following law.

Law 6.25 (Undefined activation) *For distinct process variables g and h.*

$$(g :=_m h \,;\, \langle h \rangle) \;=\; CHAOS \qquad \{\text{Law 6.25.A}\}$$
$$(ch!!h \rightarrow \langle h \rangle) \;=\; ch.h \,;\, CHAOS \qquad \{\text{Law 6.25.B}\}$$

where ch is a channel name. □

Law 6.25 captures the fact that a mobile process has moved after assignment or communication, since its value has been passed to a new location (g, or the other end of the channel ch), and none of its behaviours is available at its old location (the higher-order variable h). In Law 6.25.A, as there is no communication with the environment, the update of g can not be observed, therefore the whole effect is the same as $CHAOS$; however, in Law 6.25.B, the execution of h still leads to $CHAOS$, but this does not undo the communication that has already happened.

The proofs of the above two laws can be easily derived from Law 6.19, the definition of output, Law 6.24 and Law 6.23.

6.4 Prefix

The mobile output can be replaced by clone output.

Law 6.26 (Mobile output to clone output)

$$(ch!!q \rightarrow P) = (ch!q \rightarrow \textbf{\textit{end}}\ q;\ \textbf{\textit{vid}}\ q;\ P)$$

***Proof*:** *Directly from the definitions.* □

6.5 Iteration

If a boolean condition b holds, then the iteration $b*P$ will execute P and iterate. Otherwise, it terminates immediately. In first-order programming, there is a law

$$(t := e;\ b * P) \;=\; t := e;\ (P;\ b * P) \lhd b[e/t] \rhd SKIP$$

in which the initial value of b can be evaluated by substituting the occurrences of t in b with its value e. However, a similar one in our theory may not hold

$$(h := \{\!|E|\!\};\ b * P) \;=?\; h := \{\!|E|\!\};\ (P;\ b * P) \lhd b[E/h] \rhd SKIP$$

This is because we cannot evaluate the initial value of b by substituting h with E, as the value of h is non-deterministic.

The following two laws of conventional imperative language still hold.

Law 6.27 $true * SKIP = CHAOS$ □

Law 6.28 $false * P = SKIP$ □

6.6 Other processes

The algebraic laws for some other processes are largely same as their counterpart in non-mobile conventional imperative programming. We briefly list these laws in this section, without detailed explanation.

Property 6.1 (Parallel composition)

(1) $P \parallel Q \;=\; Q \parallel P$
(2) $P \parallel (Q \parallel R) \;=\; (P \parallel Q) \parallel R$
(3) $SKIP \parallel P \;=\; P$
(4) $CHAOS \parallel P \;=\; CHAOS$
(5) $(P \lhd b \rhd Q) \parallel R \;=\; (P \parallel R) \lhd b \rhd (Q \parallel R)$ □

Property 6.2 (External choice)

(1) $P \,\Box\, P = P$
(2) $P \,\Box\, Q = Q \,\Box\, P$
(3) $(P \,\Box\, Q) \,\Box\, R = P \,\Box\, (Q \,\Box\, R)$
(4) $P \,\Box\, CHAOS = CHAOS$
(5) $P \,\Box\, STOP = P$
(6) $P \,\Box\, SKIP \sqsubseteq SKIP$
(7) $P \,\Box\, (Q \sqcap R) = (P \,\Box\, Q) \sqcap (P \,\Box\, R)$
(8) $P \sqcap (Q \,\Box\, R) = (P \sqcap Q) \,\Box\, (P \sqcap R)$
(9) $P;\ (Q \,\Box\, R) = (P;\ R) \,\Box\, (P;\ R)$
(10) $(P \,\Box\, Q);\ R = (P;\ R) \,\Box\, (Q;\ R)$
(11) $P \lhd b \rhd (Q \,\Box\, R) = (P \lhd b \rhd Q) \,\Box\, (P \lhd b \rhd R)$
(12) $P \,\Box\, (Q \lhd b \rhd R) = (P \,\Box\, Q) \lhd b \rhd (P \,\Box\, R)$ □

Property 6.3 (Internal choice)

(1) $P \sqcap P = P$
(2) $P \sqcap Q = Q \sqcap P$
(3) $(P \sqcap Q) \sqcap R = P \sqcap (Q \sqcap R)$
(4) $CHAOS \sqcap P = CHAOS$
(5) $P;\ (Q \sqcap R) = (P;\ Q) \sqcap (P;\ R)$
(6) $(P \sqcap Q);\ R = (P;\ R) \sqcap (Q;\ R)$
(7) $P \lhd b \rhd (Q \sqcap R) = (P \lhd b \rhd Q) \sqcap (P \lhd b \rhd R)$
(8) $P \sqcap (Q \lhd b \rhd R) = (P \sqcap Q) \lhd b \rhd (P \sqcap R)$ □

Property 6.4 (Hiding)

(1) $P \setminus \{\} = P$
(2) $P \setminus X \setminus Y = P \setminus (X \cup Y)$
(3) $(a \to P) \setminus X = \begin{cases} a \to (P \setminus X), & \text{if } a \notin X; \\ P \setminus X, & \text{if } a \in X \end{cases}$
(4) $(P;\ Q) \setminus X = (P \setminus X);\ (Q \setminus X)$

(5) $(P \parallel Q) \setminus X = (P \setminus X) \parallel Q$, $\{X \notin \mathcal{A}Q\}$

(6) $ch.E \setminus CH = SKIP$, $\{ch \in CH\}$

(7) $P \setminus CH = P$ $\{P$ not communicate through any channel in $CH\}$

□

7 Development Method for Mobile Processes

Our development procedure involves two main steps. In the first step we group similar pieces of specification as the value of a parameterised process variable (see Section 7.1). In the second step, by converting assignment into communication, we make the variable mobile: consequently the activations of the variable can be completed in different hosts over a network (see Section 7.2). We present the derivation procedure through a set of laws for ***MP*** processes.

7.1 Abstracting process variable

The first step of the derivation is to replace a process by assigning this process constant to a new declared process variable and following by its activation.

Law 7.1 (Copy-rule-2)

$$Q \;=\; (\textbf{\textit{proc}}\ h := \{\!|Q|\!\} \bullet \langle h \rangle) \qquad \{h, h' \notin \alpha Q\}$$

Proof.

LHS	{variable introduction}
$= \textbf{\textit{proc}}\ h;\ Q;\ \textbf{\textit{end}}\ h$	{constant assignment vacuous}
$= \textbf{\textit{proc}}\ h;\ Q;\ h := \{\!|Q|\!\};\ \textbf{\textit{end}}\ h$	{process-constant}
$= \textbf{\textit{proc}}\ h;\ h := \{\!|Q|\!\};\ Q;\ \textbf{\textit{end}}\ h$	{copy-rule-1}
$= \textbf{\textit{proc}}\ h;\ h := \{\!|Q|\!\};\ \langle h \rangle;\ \textbf{\textit{end}}\ h$	{syntax}
$= RHS$	

□

The law for parameterised processes differs in that we activate them by providing actual parameters.

Law 7.2 (Parameterised copy-rule)

$$Q(i,j,k) \;=\; \textbf{\textit{proc}}\ h := \{\!|\lambda\, x : var(T_1), y : val(T_2), z : res(T_3) \bullet Q(x,y,z)|\!\} \bullet h(i,j,k) \qquad \{h, h' \notin \alpha Q\}$$

where $Q(i,j,k)$ is a parameterised process with actual name, value and result parameters i, j and k of type T_1, T_2 and T_3 respectively. □

We introduce two notations to represent a series of sequential compositions. An indexed sequential composition is the sequential composition of a series of processes in order.

Definition 7.1 (Indexed sequential composition)

$$(\,{}_{;}^{\circ}\, i : 1 \,..\, n \bullet P_i\,) \;\widehat{=}\; \begin{cases} \mathit{SKIP} & n = 0 \\ (\,{}_{;}^{\circ}\, i : 1 \,..\, n-1 \bullet P_i\,)\;;\, P_n & n \geq 1 \end{cases} \qquad \Box$$

Sequential compositions may be iterated over a sequence s.

Definition 7.2 (Iterated sequential composition) $\forall\, s \in \mathrm{seq}\, T$

$$(\,{}_{;}^{\circ}\, i : s \bullet P(i)\,) \;\widehat{=}\; \begin{cases} \mathit{SKIP} & s = \langle\rangle \\ P(\mathit{head}(s));\ (\,{}_{;}^{\circ}\, i : \mathit{tail}(s) \bullet P(i)\,) & s \neq \langle\rangle \end{cases}$$

where i is one of the elements in the sequence of parameters, and P is a parameterised process, $\mathit{head}(s)$ is the first element of s, $\mathit{tail}(s)$ is a subsequence of s after removing its first element. $\Box$

For example, we denote the program $(t := t + 2;\ t := t + 7;\ t := t + 5)$ as

$${}_{;}^{\circ}\, i : \langle 2, 7, 5\rangle \bullet \{\!|\lambda\, j : \mathit{val}(\mathbb{N}) \bullet t := t + j|\!\}(i)$$

For a series of similar pieces of program, we may be able to assign the parameterised process to a newly-introduced process variable, and activate it in series with proper arguments.

Law 7.3 (Iterated parameterised copy-rule) *Suppose Q is a parameterised process with a value parameter of type I, then, for any sequence s,*

$$(\,{}_{;}^{\circ}\, i : s \bullet \{\!|\lambda\, j : \mathit{val}(I) \bullet Q(j)|\!\}(i)\,) = \left(\begin{array}{c} \textbf{\textit{proc}}\ h := \{\!|\lambda\, j : \mathit{val}(I) \bullet Q(j)|\!\} \bullet \\ (\,{}_{;}^{\circ}\, i : s \bullet h(i)\,) \end{array}\right) \qquad \{h, h' \notin \alpha Q\}$$

$\Box$

For instance, by using this law, we have the following derivation

$$(t := t + 2;\ t := t + 7;\ t := t + 5) = \left(\begin{array}{c} \textbf{\textit{proc}}\ h := \{\!|\lambda\, j : \mathit{val}(\mathbb{N}) \bullet t := t + j|\!\} \bullet \\ (h(2);\ h(7);\ h(5)) \end{array}\right)$$

As a special case of the above law, when $s = \langle 1, 2, \ldots, n\rangle$,

$$(\,{}_{;}^{\circ}\, i : 1 \,..\, n \bullet P(i)\,) = \left(\begin{array}{c} \textbf{\textit{proc}}\ h := \{\!|\lambda\, j : \mathit{val}(\mathbb{N}) \bullet P(j)|\!\} \bullet \\ (\,{}_{;}^{\circ}\, i : 1 \,..\, n \bullet h(i)\,) \end{array}\right)$$

In the same way, we have similar laws for an iterated parameterised process which has name or result parameters.

7.2 Moving process variable

Even though we group similar pieces of specification as the value of a newly introduced parameterised process variable and activate it at necessary occurrences (Law 7.3), the whole specification is still centralised. In order to achieve a distributed system, we may consider putting many activations of this variable in different distributed components. To make sure that the variables activated in different components have the same process values or similar structures, we initialise the variable at one component but make it mobile, transmitted from one distributed component to another one. It is necessary to introduce communication in this step. Actually, the assignment and the communication are semantically equivalent.

Law 7.4 (Assignment-communication equivalence)

$$\begin{array}{rcl}(p := q) & = & ((ch?p \rightarrow SKIP) \parallel (ch!q \rightarrow SKIP)) \setminus \{ch\} \\ (p :=_m q) & = & ((ch?p \rightarrow SKIP) \parallel (ch!!q \rightarrow SKIP)) \setminus \{ch\}\end{array}$$ □

We borrow the concepts of pipes and chaining in CSP [12, 24]. Pipes are special processes which have only two channels, namely an input channel *left* and an output channel *right*. For example, a pipe that recursively accepts a number from *left*, and output its double to *right* can be represented by:

$$\mu\, X \bullet left?p \rightarrow right!(p + p) \rightarrow X$$

Chaining links two pipes together as a new pipe.

Definition 7.3 (Chaining)

$$P \gg Q \;\widehat{=}\; (P[mid/right] \parallel Q[mid/left]) \setminus \{mid\}$$ □

It is clear that chaining operator is associative.

Law 7.5 (Chaining associative)

$$P \gg (Q \gg R) \;=\; (P \gg Q) \gg R$$ □

We may define the indexed chaining which connects a series of processes in order as a long pipe.

Definition 7.4 (Indexed-chaining)

$$(\, \gg i : 1 \,..\, n \bullet P_i\,) \;\widehat{=}\; \begin{cases} P_1 & n = 1 \\ (\, \gg i : 1 \,..\, n-1 \bullet P_i\,) \gg P_n & n > 1 \end{cases}$$

□

We introduce a new notation: double chaining links two pipes as a ring. All the communications between pipes are hidden from the environment.

Definition 7.5 (Double-chaining)

$$P \lll\!\ggg Q \mathrel{\widehat{=}} (P[mid_1, mid_2/right, left] \parallel Q[mid_1, mid_2/left, right]) \setminus \{mid_1, mid_2\}$$

□

The double chaining operator is commutative.

Law 7.6 (Double-chaining commutative)

$$P \lll\!\ggg Q = Q \lll\!\ggg P$$ □

A ring of processes can be viewed as a long chain with the two chain ends connected. The order of processes in the ring is important, but the connecting point of the chain can be arbitrary. In other words, the chain can be started from any process and ended at one of its backwards adjacent process. This feature is captured by the following law.

Law 7.7 (Exchange)

$$\begin{array}{ll} & P_1 \lll\!\ggg (\gg i : 2 \mathinner{..} n \bullet P_i) \\ = & P_k \lll\!\ggg ((\gg i : k+1 \mathinner{..} n \bullet P_i) \gg (\gg i : 1 \mathinner{..} k-1 \bullet P_i)) \quad 1 < k < n \\ = & P_n \lll\!\ggg (\gg i : 1 \mathinner{..} n-1 \bullet P_i) \end{array}$$

Proof. *We show the proof of* $P \lll\!\ggg (Q \gg R) = Q \lll\!\ggg (R \gg P)$

$$\begin{array}{ll} & P \lll\!\ggg (Q \gg R) & \{\text{def of } \lll\!\ggg\} \\ = & (P[m_1, m_2/r, l] \parallel (Q \gg R)[m_1, m_2/l, r]) \setminus \{m_1, m_2\} & \{\text{def of } \gg\} \\ = & (P[m_1, m_2/r, l] \parallel (Q[m/r] \parallel R[m/l])[m_1, m_2/l, r]) \setminus \{m_1, m_2, m\} & \{\text{def of pipe}\} \\ = & (P[m_1, m_2/r, l] \parallel Q[m_1, m/l, r] \parallel R[m, m_2/l, r]) \setminus \{m_1, m_2, m\} & \{\parallel\text{-commutative}\} \\ = & (Q[m_1, m/l, r] \parallel (R[m, m_2/l, r] \parallel P[m_1, m_2/r, l])) \setminus \{m_1, m_2, m\} & \{\text{def of } \gg\} \\ = & (Q[m_1, m/l, r] \parallel (R \gg P)[m, m_1/l, r]) \setminus \{m_1, m\} & \{\text{def of } \lll\!\ggg\} \\ = & Q \lll\!\ggg (R \gg P) \end{array}$$

□

The update of a variable p by an expression of p can be implemented by double chaining two pipes, where the first pipe mobile outputs p, while the second pipe receives the value of p, assigns it to r and then outputs the value of the updated variable immediately to the first pipe.

Law 7.8 (Delegation with double-chaining)

$$(p := f(p)) = \left(\begin{array}{l} (right!!p \rightarrow left?p \rightarrow SKIP) \\ \lll\!\ggg (\boldsymbol{vid}\, r;\ left?r \rightarrow right!f(r) \rightarrow \boldsymbol{end}\, r) \end{array} \right)$$

where $f(p)$ *is an expression of* p*.*

Proof.

$$
\begin{array}{ll}
RHS & \{\text{Definition 7.5}\} \\
= ((mid_1!!p \rightarrow mid_2?p \rightarrow SKIP) \parallel (\textbf{\textit{vid}}\ r;\ mid_1?r \rightarrow mid_2!f(r) \rightarrow \textbf{\textit{end}}\ r)) & \\
\quad \setminus \{mid_1, mid_2\} & \{\text{Law 6.14}\} \\
= \textbf{\textit{vid}}\ r;\ ((mid_1!!p \rightarrow mid_2?p \rightarrow SKIP) \parallel (mid_1?r \rightarrow mid_2!f(r) \rightarrow SKIP)) & \\
\quad \setminus \{mid_1, mid_2\};\ \textbf{\textit{end}}\ r & \{\text{Law 7.4, twice}\} \\
= \textbf{\textit{vid}}\ r;\ r := p;\ p := f(r);\ \textbf{\textit{end}}\ r & \{\text{Law 6.22}\} \\
= \textbf{\textit{vid}}\ r;\ r := p;\ p := f(p);\ \textbf{\textit{end}}\ r & \{\text{Law 6.12}\} \\
= \textbf{\textit{vid}}\ r;\ r := p;\ \textbf{\textit{end}}\ r;\ p := f(p) & \{\text{Law 6.4}\} \\
= SKIP;\ p := f(p) & \{\textbf{\textit{M6}}\} \\
= LHS &
\end{array}
$$

□

As the update is executed in the second pipe, the value of p in the first pipe is irrelevant after its output and before its coming back. Therefore, we adopt mobile output for p. In the second pipe, $f(r)$ is not a variable but a value based on variable r, so that we use normal output for $f(r)$.

Similarly, the serial update of a variable can also be implemented by a ring of pipes, in which different updates are executed in different pipes.

Law 7.9 (Serial delegation with chaining)

$$
(w := g(f(w))) \ = \ \left(\begin{array}{l} right!!w \rightarrow left?w \rightarrow SKIP \\ \lll\gg \left(\begin{array}{l} \textbf{\textit{vid}}\ p;\ left?p \rightarrow right!f(p) \rightarrow \textbf{\textit{end}}\ p \\ \gg \\ \textbf{\textit{vid}}\ q;\ left?q \rightarrow right!g(q) \rightarrow \textbf{\textit{end}}\ q \end{array} \right) \end{array} \right)
$$

Proof. *Similar to the proof of Law 7.8.* □

In a more general rule, a series of updating p through different processes $F_i(p, p')$ can be replaced by a loop pipelining, in which the series of update task are allocated in different pipes.

Law 7.10 (Loop pipelining)

$$
\begin{array}{l}
(\fatsemi\, i : 1 \mathinner{..} n \bullet F_i(p, p');\ w := p) \ = \\
\quad \left(\begin{array}{l} (right!!p \rightarrow left?p \rightarrow w := p) \\ \lll\gg \\ \ggg\, i : 1 \mathinner{..} n \bullet (\textbf{\textit{vid}}\ r;\ left?r \rightarrow F_i(r, r');\ right!!r \rightarrow \textbf{\textit{end}}\ r) \end{array} \right)
\end{array}
$$

Proof. *Induction over n.* □

In the right hand side of the above law, the value of p travels from the first pipe to the series of pipes. Its final value, which is stored in w, is retrieved after p's travelling back from the series of pipes.

When updating is performed by a series of activations of the same process variable, we can move this process variable around the loop pipelining and distribute the activations in different pipes.

Lemma 7.1 (Loop pipelining) *Suppose that h is a parameterised process variable with a value parameter i and a name parameter t, then*

$$(\fatsemi i : 1 \mathinner{..} n \bullet h(i,t);\ w := t) \ = \left(\begin{array}{l} (\mathit{right}!!h!!t \rightarrow \mathit{left}?h?t \rightarrow w := t) \\ \Diamond \\ \gg i : 1 \mathinner{..} n \bullet \\ \quad (\mathbf{proc}\, g;\ \mathbf{vid}\, r;\ \mathit{left}?g?r \rightarrow g(i,r);\ \mathit{right}!!g!!r \rightarrow \mathbf{end}\, g, r) \end{array} \right)$$

□

In practice, t is the local state of mobile process variable h. When the mobile process variable moves, it takes not only its process value but also its local state; however, in our current work, we have not formalised the local state of a mobile process, therefore we simply send the local state together with the process variable in multiple data transfer [24], which also says that the two variables h and t are output at the same time.

8 Decentralising the Data Centre

Using the laws in Section 7, we are able to show the derivation of the distributed system from the centralised one in Section 3. As the behaviour of the hosts to send out the information is not our main concern, we simply ignore this part in our specification.

In the centralised system, initially, the variable t is initialised to 0. The data centre then repeats the task of communicating with each host, obtaining the information and updating t until all the hosts have been processed. Finally the data centre outputs the data through channel *result*. This specification can be written as

$$\mathbf{var}\ t := 0 \bullet (\fatsemi i : 1 \mathinner{..} n \bullet c.i?v \rightarrow t := t + v);\ \mathit{result}!t \rightarrow \mathit{SKIP}$$

where $c.i$ is the channel connecting the data centre and the ith host. We notice that similar pieces of specification involving input and update occurs iteratively, therefore we can assign a parameterised process to a process variable, and then activate it repeatedly with proper arguments. In the parameterised process, the initial value of t needs to be known at the beginning of every communication with the host, and certainly we need to store the result of t after its update, therefore we select t as a name parameter. By applying the

iterated parameterised copy rule (Law 7.2), we reach the following development:

$$
\begin{array}{l}
= \ \{\text{iterated-parameterised-copy-rule}\} \\
\textbf{\textit{var}}\ t := 0 \bullet \\
\quad \left(\begin{array}{l} \textbf{\textit{proc}}\ p := \{\!|\lambda\, j : val(1\,..\,n);\ u : var(\mathbb{N}) \bullet c.j?v \rightarrow u := u + v|\!\} \bullet \\ \quad\ (\fatsemi\, i : 1\,..\,n \bullet p(i,t)) \end{array} \right); \\
\quad result!t \rightarrow SKIP
\end{array}
$$

The channel index j is only of relevance at the beginning of the variable activation, therefore it is a value parameter.

As the output $result!t$ does not contain variable p, it can be moved inside the scope of p. By an application of Law 6.12, we calculate the following:

$$
\begin{array}{l}
= \ \{\text{variable-}p\text{-scope-extension}\} \\
\textbf{\textit{var}}\ t := 0 \bullet \\
\quad \left(\begin{array}{l} \textbf{\textit{proc}}\ p := \{\!|\lambda\, j : val(1\,..\,n);\ u : var(\mathbb{N}) \bullet c.j?v \rightarrow u := u + v|\!\} \bullet \\ \quad\ \fatsemi\, i : 1\,..\,n \bullet p(i,t);\ result!t \rightarrow SKIP \end{array} \right)
\end{array}
$$

The task of updating t by activating p can be completed using a series of pipes. The whole specification can be replaced by double chaining this series of pipes with another pipe, in which the initial value of t is sent out and the final value of t is retrieved. As the intermediate values of t and process variable p are of no concern, we use mobile output for t and p. When the activations have been completed in every host, the process variable is of no use to us, therefore we discard it in the last-visited host n and only the variable w that stores the data is output to the first pipe. By applying the loop-pipelining law (Lemma 7.1), we get the following specification.

$$
\begin{array}{l}
= \ \{\text{loop-pipelining}\} \\
\textbf{\textit{var}}\ t := 0 \bullet \\
\quad \left(\begin{array}{l} \textbf{\textit{proc}}\ p := \{\!|\lambda\, j : val(1\,..\,n);\ u : var(\mathbb{N}) \bullet c.j?v \rightarrow u := u + v|\!\} \bullet \\ \quad \left(\begin{array}{l} right!!p!!t \rightarrow left?t \rightarrow result!t \rightarrow SKIP \\ \Diamond\!\!\gg \\ \left(\begin{array}{l} \gg\, i : 1\,..\,(n-1) \bullet \\ \quad \left(\begin{array}{l} \textbf{\textit{proc}}\ q;\ \textbf{\textit{var}}\ w;\ left?q?w \rightarrow q(i,w); \\ right!!q!!w \rightarrow \textbf{\textit{end}}\ q,w \end{array} \right) \\ \gg \left(\begin{array}{l} \textbf{\textit{proc}}\ q;\ \textbf{\textit{var}}\ w;\ left?q?w \rightarrow q(n,w); \\ right!!w \rightarrow \textbf{\textit{end}}\ q,w \end{array} \right) \end{array} \right) \end{array} \right) \end{array} \right)
\end{array}
$$

This specification is still centralised, as the scopes of p and t are valid for all the pipes. We notice, however, these two variables do not occur in the pipes involved in updating, therefore we can take these pipes out of the scope of p and t. Applying the variable scope shrinkage (Law 6.12), we reach a distributed

system.

$$
\begin{array}{l}
= \ \{\text{variable-}t\text{-and-}p\text{-scope-shrinkage}\} \\
\left(\begin{array}{l}
\textbf{\textit{var}}\ t := 0 \bullet \\
\quad \textbf{\textit{proc}}\ p := \{\!|\lambda\, j : val(1\,..\,n);\ u : var(\mathbb{N}) \bullet c.j?v \rightarrow u := u + v|\!\} \bullet \\
\quad\quad right!!p!!t \rightarrow left?t \rightarrow result!t \rightarrow SKIP
\end{array}\right) \\
\ll\gg \\
\left(\begin{array}{l}
\left(\begin{array}{l}
\gg\, i : 1\,..\,(n-1) \bullet \\
\quad \textbf{\textit{proc}}\ q;\ \textbf{\textit{var}}\ w;\ left?q?w \rightarrow q(i,w);\ right!!q!!w \rightarrow \textbf{\textit{end}} q, w
\end{array}\right) \\
\gg(\textbf{\textit{proc}}\ q;\ \textbf{\textit{var}}\ w;\ left?q?w \rightarrow q(n,w);\ right!!w \rightarrow \textbf{\textit{end}}\ q, w)
\end{array}\right)
\end{array}
$$

In the distributed system, the data centre and the hosts are arranged as a ring. The first component of the double chaining is the data centre, in which the variables t and p are mobile sent out, along channel *right*, after initialisation. The hosts are specified as a series of pipes, in which the process variable and the data are received from channel *left*, and then output to channel *right* after activation of the process variable.

9 Conclusions and Future Work

We have presented the UTP denotational semantics of mobile processes and a set of laws for the development of mobile distributed systems. The correctness of these laws can be guaranteed by the semantics, and we have proved most of the presented. Through a simple example that can be implemented in both a centralised and a distributed fashion, we have shown these laws are suitable for a step-wise development, starting with a centralised specification and ending up with a distributed implementation, while we cannot achieve this using the π-calculus.

Our current work on the semantics of mobile processes mainly focuses on their mobility, and our development method applies to an initial specification without any mobile process; however, we simply ignore the process type and rich data structure within a mobile process. In $\mathsf{occam}_\mathsf{M}$ [1], a process type determines an interface, a mobile process can implement multiple process types, and the value of a process variable is an instance of a mobile process that implements the type of this variable. Therefore, the activation of a process variable is determined by its type, and the process from which it is initialised. We formalise these issues, and then study the refinement of a mobile process itself.

In $\mathsf{occam}_\mathsf{M}$ [1], channels have mobility. Channel variables reference only one of the ends of a channel bundle and those ends are mobile, and can be passed through channels. Moving one of the channel-bundle ends around a network enables processes to exchange communication capabilities, making the communication highly flexible. We intend to formalise this channel-ends mobility in the UTP and study its refinement calculus.

In the example in this paper, the derivation from the centralised system to the distributed one centres around the introduction of a higher-order variable.

Actually, moving processes can decrease network cost by replacing remote communication with local communication. Furthermore, after a mobile assignment or a mobile communication, the source variable is undefined and its allocated memory space is released to the environment. Clearly, consuming less network cost and occupying less memory space can be a performance enhancement, and in this sense, we have not demonstrated a performance improvement in our mobile system. We intend to investigate techniques for reasoning about this.

Our main objective is to include the semantics of mobile processes and its associated refinement calculus in *Circus* [34], a unified language for describing state-based reactive systems. The semantics [35] of *Circus* is based on UTP and a development method for *Circus* based on refinement [25, 3, 4] has been proposed. This inclusion will enhance the *Circus* model and allow *Circus* specification to reason about mobility.

References

[1] F. R. M. Barnes and P. H. Welch. Prioritied dynamic communicating and mobile processes. *IEE Proceedings Software*, 150(2):121–136, April 2003.

[2] Geoef Barrett. occam3 reference manual. Technical report, INMOS Limited, Mar 1992.

[3] A. L. C. Cavalcanti, A. C. A. Sampaio, and J. C. P. Woodcock. Refinement of actions in *Circus*. In *REFINE'2002, Electronic notes in Theoretical Computer Science*, 2002.

[4] A. L. C. Cavalcanti, A. C. A. Sampaio, and J. C. P. Woodcock. A refinement strategy for *Circus*. *Formal Aspects of Computing*, 2003(15):146–181, 2003.

[5] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Engewood Cliffs, 1976.

[6] M. P. Fiore, E. Moggi, and D. Sangiorgi. A fully-abstract model for the π-calculus. In *Proceedings of the Eleventh Annual IEEE Symposium On Logic In Computer Science (LICS'96)*, pages 43–54, New York, USA, July 1996. IEEE Computer Society Press.

[7] The RAISE Language Group. *The RAISE Specification Language*. BCS Practitioner Series. Prentice Hall, 1992.

[8] He Jifeng, Liu Zhiming, and Li Xiaoshan. A relational model for object-oriented programming. Technical Report 231, UNU/IIST, P.O.Box 3058, Macau, May 2001.

[9] He Jifeng, Liu Zhiming, and Li Xiaoshan. Towards a refinement calculus for object systems. In *Proceedings of the Conference ICCI2002*, pages 69–77, Calgary, Canada, 2002. IEEE Computer Society Press.

[10] He Jifeng, Liu Zhiming, and Li Xiaoshan. Modelling object-oriented programming with reference type and dynamic binding. Technical Report 280, UNU/IIST, P.O.Box 3058, Macau, May 2003.

[11] M. Hennessy. A fully abstract denotational model for higher-order processes. In *8th Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, 1993.

[12] C. A. R. Hoare. *Communicating Sequential Process*. Prentice Hall, 1985.

[13] C. A. R. Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice Hall, 1998.

[14] Jin Naiyong and He Jifeng. Resource semantic models for programming languages. Technical Report 277, UNU/IIST, P.O.Box 3058, Macau, April 2003. unpublished.

[15] Li Li and He Jifeng. A denotational semantics of timed RSL using duration calculus. In *Proceedings of the 6th International Conference on Real-Time Computing Systems and Applications (RTCSA'99)*, pages 492–503, Hong Kong, 13-15 Decemeber 1999. IEEE Computer Society Press.

[16] INMOS Limited. occam2.1 reference manual. Technical report, INMOS Limited, May 1995.

[17] B. Mahony and J. S. Dong. Timed communicating object-Z. *IEEE Transactions on Software Engineering*, 26(2):150–177, February 2000.

[18] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, parts I and II. Technical Report ECS-LFCS-89-85 and -86, University of Edinburgh, 1989.

[19] Robin Milner. *Communication and Concurrency*. Computer Science. Prentice Hall, 1989.

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

[21] Carrol Morgan. *Programming from Specifications*. Prentice Hall, second edition, 1998.

[22] Shengchao Qin, Jin Song Dong, and Wei-Ngan Chin. A semantic foundation for TCOZ in unifying theories of programming. In *FM03*, To appear, 2003.

[23] Hyon Sul Ri and He Jifeng. A complete verification system for timed RSL. Technical Report 275, UNU/IIST, P.O.Box 3058, Macau, March 2003.

[24] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1998.

[25] A. C. A. Sampaio, J. C. P. Woodcock, and A. L. C. Cavalcanti. Refinement in *Circus*. In L. Eriksson and P. A. Lindsay, editors, *FME 2002: Formal Methods — Getting IT Right*, volume 2391 of *Lecture Notes in Computer Science*, pages 451–470. Springer-Verlag, 2002.

[26] D. Sangiorgi. *Expressing Mobility in Process Algebras: First-order and Higher-order Paradigms*. PhD thesis, Department of Computer Science, University of Edinburgh, 1992.

[27] Davide Sangiorgi and David Walker. *The π-calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001.

[28] Adnan Sherif and He Jifeng. A framework for the specification, verification and development of real time systems using *Circus*. Technical Report 270, UNU/IIST, P.O.Box 3058, Macau, November 2002.

[29] Adnan Sherif and He Jifeng. Toward a time model for *Circus*. In C. George and H. Miao, editors, *ICFEM 2002*, volume 2495 of *Lecture Notes in Computer Science*, pages 613–624. Springer-Verlag, 2002.

[30] Ian Stark. A fully abstract domain model for the π-calculus. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science*, pages 36–42. IEEE Computer Society Press, 1996.

[31] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[32] J. C. P. Woodcock. Unifying theories of parallel programming. In *Logic and Algebra for Engineering Software*. IOS Press, 2002. Also Keynote speech in ICFEM 2002: 4th International Conference on Formal Engineering Methods, Shanghai, IEEE Computer Society Press.

[33] J. C. P. Woodcock and A. L. C. Cavalcanti. *Circus*: a concurrent refinement language. Technical report, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford OX1 3QD, UK, July 2001.

[34] J. C. P. Woodcock and A. L. C. Cavalcanti. A concurrent language for refinement. In A. Butterfield and C. Paul, editors, *IWFM'01: 5th Irish Workshop in Formal Methods*, Dublin, Ireland, July 2001.

[35] J. C. P. Woodcock and A. L. C. Cavalcanti. The semantics of *Circus*. In J. P. Bowen, M. C. Henson, and K. Robinson, editors, *ZB2002: Formal Specification and Development in Z and B*, volume 2272 of *Lecture Notes in Computer Science*, pages 184–203. Springer-Verlag, 2002.

[36] Jim Woodcock and Jim Davies. *Using Z: Specification, Refinement and Proof*. Prentice Hall, 1996.

[37] Zhou Chaochen, C. A. R. Hoare, and A. P. Ravn. A calculus of durations. *Information Processing Letters*, 40(5):269–276, 1991.