The Journal of Logic and Algebraic Programming 81 (2012) 2–25

Contents lists available at SciVerse ScienceDirect

The Journal of Logic and Algebraic Programming

journal homepage: www.elsevier.com/locate/jlap

Linking operational semantics and algebraic semantics for a probabilistic timed shared-variable language

Huibiao Zhu [a,*], Fan Yang [a], Jifeng He [a], Jonathan P. Bowen [b], Jeff W. Sanders [c], Shengchao Qin [d]

[a] Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, Shanghai 200062, China
[b] Museophile Limited, Oak Barn, Sonning Eye, Reading RG4 6TN, United Kingdom
[c] International Institute for Software Technology, United Nations University, Macau SAR, China
[d] School of Computing, Teesside University, Middlesbrough TS1 3BA, United Kingdom

ARTICLE INFO

Article history:
Available online 5 September 2011

Keywords:
PTSC
Operational semantics
Algebraic semantics
Semantic linking
Head normal form
Animation

ABSTRACT

Complex software systems typically involve features like time, concurrency and probability, with probabilistic computations playing an increasing role. However, it is currently challenging to formalize languages incorporating all those features. Recently, the language *PTSC* has been proposed to integrate probability and time with shared-variable concurrency (Zhu et al. (2006, 2009) [51,53]), where the operational semantics has been explored and a set of algebraic laws has been investigated via bisimulation. This paper investigates the link between the operational and algebraic semantics of *PTSC*, highlighting both its theoretical and practical aspects.

The link is obtained by deriving the operational semantics from the algebraic semantics, an approach that may be understood as establishing soundness of the operational semantics with respect to the algebraic semantics. Algebraic laws are provided that suffice to convert any *PTSC* program into a form consisting of a guarded choice or an internal choice between programs, which are initially deterministic. That form corresponds to a simple execution of the program, so it is used as a basis for an operational semantics. In that way, the operational semantics is derived from the algebraic semantics, with transition rules resulting from the derivation strategy. In fact the derived transition rules and the derivation strategy are shown to be equivalent, which may be understood as establishing completeness of the operational semantics with respect to the algebraic semantics.

That theoretical approach to the link is complemented by a practical one, which animates the link using Prolog. The link between the two semantics proceeds via head normal form. Firstly, the generation of head normal form is explored, in particular animating the expansion laws for probabilistic interleaving. Then the derivation of the operational semantics is animated using a strategy that exploits head normal form. The operational semantics is also animated. These animations, which again supports to claim soundness and completeness of the operational semantics with respect to the algebraic, are interesting because they provide a practical demonstration of the theoretical results.

1. Introduction

Probabilistic computations play an increasingly important role in solving various problems [44]. As a consequence, a number of probabilistic languages have been proposed [11,12,14,21,34–36,41–43]. In addition to probability, complex software systems can often involve important features like real-time [40] and shared-memory based concurrency. The shared-variable mechanism is typically used for communications among components running in parallel, e.g., the case for

* Corresponding author.
E-mail address: hbzhu@sei.ecnu.edu.cn (H. Zhu)

1567-8326/$ - see front matter © 2011 Elsevier Inc. All rights reserved.
doi:10.1016/j.jlap.2011.06.003

Java and the case for the Verilog hardware description language. However, it proves to be rather challenging to formalize a system comprising those features [49]. A formal model integrating probability, real-time and shared-variable concurrency would be expected by practical system designers since it can offer better support for the specification and modelling of complex software systems.

In our previous work [51,53], we have proposed an integrated language *PTSC* equipped with probability, time and shared-variable concurrency features, to facilitate the specification of complex software systems. In the language, the probability feature is reflected by the probabilistic nondeterministic choice, probabilistic guarded choice and the probabilistic scheduling of actions from different concurrent components in a program. In that work, we have formalized an operational semantics for this language, on top of which an abstract bisimulation relation has been defined and several algebraic laws have been derived for program equivalence.

The *PTSC* language has recently been used to specify a circuit in the register-transfer level [38,39]. The circuit takes two integers as the input and sums up them as the output, where the register containing one of the inputs may be faulty. The algebraic laws proposed for the *PTSC* language have also been employed to verify that an implementation of the circuit with probabilistic behaviour conforms to the probabilistic specification.

As described in Hoare and He's Unifying Theories of Programming (abbreviated as UTP) [28], three different mathematical models are often used to represent a theory of programming, namely, the operational, the denotational, and the algebraic approaches [29,45,48]. Each of these representations has its distinctive advantages for theories of programming. For instance, the operational semantics provides a set of transition rules that model how a program performs step by step. The correctness of the transitions of an operational semantics is essential for applying the operational semantics in further study. The algebraic semantics is well suited in symbolic calculation of parameters and structures of an optimal design. The algebraic approach has been successfully applied in provably correct compilation. A comprehensive theory of programming should offer all these semantic models and should ensure that all the models are pairwise consistent [28].

For our proposed language *PTSC*, we have previously defined the operational semantics from which a set of algebraic laws are derived. To ensure the consistency between the operational and algebraic semantics, a link should be constructed between the two models, so that one model can be derived from the other. In this paper, we tackle this problem by constructing a link from the algebraic semantics to the operational semantics. We first derive an operational semantics from the algebraic semantics. If the derived operational semantics is the same as our previous achieved one [51,53], we can conclude that our operational semantics is consistent with the algebraic semantics. This can be considered as the soundness exploration of operational semantics from the algebraic viewpoint. The investigation can be understood as the inverse work of [51,53] and a significant contribution in unifying theories of *PTSC* [28].

In order to support the derivation of operational semantics from algebraic semantics, a derivation strategy is required to be defined. Therefore, we introduce the concept of head normal form, where every program can be expressed as either a guarded choice or the summation of a set of processes that are deterministic initially. Our definition for the derivation strategy is based on head normal form. We study the algebraic laws for *PTSC*, which supports the definition of head normal form. Based on the derivation strategy, we can achieve a transition system (i.e., an operational semantics). There remains a question concerning the equivalence of the derivation strategy and the derived transition system. The advantage of this equivalence is that we can use either the derivation strategy or the derived transition system when working on the application of operational semantics. Further, this result also indicates the completeness of our operational semantics from the viewpoint of algebraic semantics.

For the derived operational semantics, if we can have an executed version, the correctness of the operational semantics can be checked from various test results. Moreover, if we can have the animation of the derivation strategy, we can also model the execution of a program. From various test examples, if the execution results of the above two animation approaches are the same, we can claim that the derivation strategy is the same as the derived operational semantics. This supports the claim that our derived operational semantics is sound and complete with respect to head normal form (i.e., algebraic semantics in general) from the practical viewpoint. Therefore, in this paper we also consider the animation of the linking between operational semantics and algebraic semantics for *PTSC*. The logic programming language Prolog [7] is used for the development. To support the animation, we also consider the mechanical generation of the head normal form for each program in Prolog, where twenty-five parallel expansion laws are animated.

The remainder of this paper is organized as follows. Section 2 introduces our language *PTSC*. Section 3 lists a set of algebraic laws, where every program can be represented as either a guarded choice, or the summation of a set of processes that are initially deterministic. We also give the concept of head normal form, which is the key point for studying the link between the operational semantics and algebraic semantics. We explore how to generate the algebraic laws via Prolog in Section 4, especially the parallel expansion laws. Meanwhile, Section 4 also calculates the head normal form of a program via Prolog. Section 5 investigates the derivation of the operational semantics from the algebraic semantics. We give the derivation strategy and show that a transition system (i.e., the operational semantics) can be derived by strict proof. Section 5 also studies the equivalence of the derivation strategy and the derived transition system. Section 6 is about the animation of operational semantics from two viewpoints. Firstly, for our derived operational semantics in Section 5.3 from the transition system viewpoint, we explore the animation of the achieved operational semantics. All transition rules are expressed in the form that is compatible with Prolog. Secondly, we also explore the generation of operational semantics from the viewpoint of derivation strategy via head normal form mechanically. Section 7 discusses the related work about probability, time, shared-variable concurrency and semantic linking. Section 8 concludes the paper.

2. The *PTSC* language

Shared-variable concurrency has long been used as an alternative to message passing in describing systems composed of concurrently interacting components. Originally termed 'multiprogramming' in the 1960s and 70s [15,16], its subtleties have continued to provoke research [4,5,9,46]. It finds recent application in the analysis of threads, for instance in the Java memory model [33], and for reasoning about Verilog [20].

The purpose of this paper is to continue the study of the language *PTSC*, which integrates probability and time with shared-variable concurrency. *PTSC* has been designed to express the scheduling of threads, incorporating concurrency and nondeterminism as well as probability and time. It is thus well suited to *discrete event simulation* where those features are present, as is the case in a growing number of distributed protocols that employ random-number generators to facilitate choice between alternatives [37].

Our language *PTSC* integrates of probability, time and shared-variable concurrency. It has the following syntactical elements:

$$P ::= \mathbf{Skip} \mid x := e \mid \mathbf{if}\ b\ \mathbf{then}\ P\ \mathbf{else}\ P \mid \mathbf{while}\ b\ \mathbf{do}\ P \mid @b\ P \mid \#n\ P \mid P\ ;\ P \mid P \sqcap P \mid P \sqcap_p P \mid P \parallel_p P$$

Note that:

(1) Assignment, $x := e$, is *atomic*, so that no action can interleave between reading the variables involved in the expression e and writing the result of e to x. The vacuous assignment $x := x$ is abbreviated **Skip**. Similar to a conventional language, **if** b **then** P **else** Q stands for the conditional, whereas **while** b **do** P stands for the iteration. $P\ ;\ Q$ stands for sequential composition. Furthermore, we introduce the notation ε to represent the empty process.
(2) The event-guarded program, $@b\,P$, is enabled when the guard b holds in which case P may be scheduled; otherwise it is disabled (i.e., not able to be scheduled) and waits. Here b is a Boolean condition. Since we model closed systems, time advances only when the guard b is false. Between evaluation of b and commencing execution of P, other commands may be scheduled. In this paper we take the understanding that $@b$ is atomic. When $@b$ appears in a guarded choice (to be introduced slightly later), it is also atomic. For $\#n\,P$, after n time units elapse, process P can be scheduled. Time advances in unit steps.
(3) $\sqcap$ stands for the nondeterministic choice, where $\sqcap_p$ stands for the probabilistic nondeterministic choice. $P \sqcap_p Q$ indicates that the probability for $P \sqcap_p Q$ to behave as P is p, where the probability for $P \sqcap_p Q$ to behave as Q is $1-p$.
(4) The mechanism for parallel composition $P \parallel_p Q$ is a shared-variable interleaving model with probability feature (i.e., probabilistic interleaving model). If process P can perform an atomic action, $P \parallel_p Q$ has conditional probability p to do that atomic action. On the other hand, if process Q can perform an atomic action, $P \parallel_p Q$ has conditional probability $1-p$ to perform that action.

Probabilistic interleaving models the situation in which priority is given to an enabled action. For example if one thread performs assignments $a1\ ;\ a2$ and another performs assignment $a3$ then the result of a scheduler that chooses $a3$ before the other assignments three quarters of the time, between them a little less than a quarter of the time, and after them the rest of the time, can be expressed as a probabilistic interleaving:

$$(a1\ ;\ a2) \parallel_{\frac{1}{4}} a3 \ = \ (a3\ ;\ a1\ ;\ a2) \parallel_{\frac{3}{4}} [\,a1\ ;\ ((a2\ ;\ a3) \parallel_{\frac{1}{4}} (a3\ ;\ a2))]$$

Indeed, by repeated expansion, the left-hand side gives the interleaving $a3\ ;\ a1\ ;\ a2$ with probability 3/4, the interleaving $a1\ ;\ a3\ ;\ a2$ with probability 3/16 and the remaining interleaving $a1\ ;\ a2\ ;\ a3$ with probability 1/16.

To facilitate algebraic reasoning, we enrich our language with a *guarded choice* [17]. As our parallel composition has probability feature, the guarded choice also shares this feature. Guarded choice is classified into five types.

The first type is composed of a set of assignment-guarded components.

(gtype-1) $[]_{i\in I}\{[p_i]\ choice_{j\in J_i}(b_{ij}\&(x_{ij} := e_{ij})\ P_{ij})\}$

Note that I and J_i stand for finite index sets. The notation $b\&(x := e)\,P$ stands for the assignment-guarded component. Assignment $x := e$ will be executed only when Boolean condition b is satisfied. After that, the subsequent behaviour can be expressed as process P. For the construct *choice*, its form can be expressed as $choice_{m\in M}(b_m\&(x_m := e_m)\ P_m)$. Its Boolean conditions should be pairwise disjoint or exhaustive, i.e., it should satisfy the condition below.

$$(\bigvee_{m\in M} b_m = true) \text{ and } (\forall m_1, m_2 \bullet (m_1 \neq m_2) \Rightarrow ((b_{m_1} \wedge b_{m_2}) = false))$$

The meaning of the *choice* construct is that at any moment there is one and only one Boolean condition to be satisfied, which indicates that the corresponding assignment can be scheduled.

For the notation $[p]choice_{m\in M}(b_m\&(x_m := e_m)\ P_m)$, it means that the probability for the *choice* construct to be performed is p; i.e., if the Boolean condition in one assignment-guarded component is satisfied, the probability for selecting the corresponding assignment is p. The first type of guarded choice is composed of a set of *choice* construct with probability. For $(gtype-1)$ above, all $p_i\ (i \in I)$ should satisfy the condition $+_{i\in I}\ p_i = 1$.

The second type is composed of a set of event-guarded components. It waits any of the guards to be fired. If one guard is satisfied, the subsequent behaviour for the whole process will be followed by its subsequent behaviour of the satisfied component.

(gtype-2) $[]_{i\in I}\{@b_i\, P_i\}$

The third type is composed of one time-delay component. Initially, it cannot do anything except letting time advance one unit.

(gtype-3) $[]\{\#1\ R\}$

The fourth type is the guarded choice composition of the first and second type of guarded choice.[1]

(gtype-4) $[]_{i\in I}\{[p_i]\, choice_{j\in J_i}(b_{ij}\&(x_{ij} := e_{ij})\, P_{ij})\}$
$[][]_{k\in K}\{@c_k\, Q_k\}$

If there exists one c_k ($k \in K$) being satisfied currently, then the event $@\, c_k$ is fired and the subsequent behaviour is Q_k. If there is no satisfied c_k, any of the assignment-guarded components can be scheduled provided that the corresponding Boolean condition is true.

The fifth type is the compound of the second and third type of guarded choice.

(gtype-5) $[]_{i\in I}\{@b_i\, P_i\}[]\{\#1\ R\}$

Currently, if there exists i ($i \in I$) such that b_i is satisfied, then the subsequent behaviour of the whole guarded choice is as P_i. On the other hand, if there is no i ($i \in I$) such that b_i is satisfied currently, then the whole guarded choice cannot do anything initially except letting time advance one unit. The subsequent behaviour is the same as the behaviour of R.

A *choice* construct contains a set of assignment-guarded components. An assignment-guarded component can be scheduled at the current time point provided that the corresponding Boolean condition is satisfied. And the execution of assignment is instantaneous. On the other hand, a delay component might be scheduled to make time advance one unit. These facts indicate that, if a *choice* construct and a time-delay component appear in the same set of a guarded choice, the time-delay component will not have a chance to be scheduled. Therefore, we assume that there is no type of guarded choice composing of the first and third type of guarded choice, which indicates that guarded choice can only have the above five types of guarded choice.

Example 2.1. Let $P = []\{\, [0.7]choice(\, true\&(x := 5)\, P_1\,),$
$[0.3]choice(\, (x > 2)\&(x := x)\, P_2,\ \ (x \le 2)\&(x := x)\, P_3\,)$
$\}$

For program P, it is of the form of the first type guarded choice, which has two *choice* constructs. The probability of selecting the first *choice* construct is 0.7, whereas the probability of selecting the second *choice* construct is 0.3. For the first *choice* construct, it has only one component, which means that assignment $x := 5$ will be scheduled and subsequent behaviour is P_1. For the second *choice* construct, it has two components. The execution of the first one is under condition $x > 2$ and the execution of the second one is under condition $x \le 2$. The second *choice* construct behaves the same as **if** $(x > 2)$ **then** P_1 **else** P_2 . Therefore, this process selects $x := 5$ with probability 0.7, followed by process P_1. It can also select the conditional statement with probability 0.3.

3. Algebraic semantics and head normal form

This section considers the algebraic semantics for *PTSC*, where every program can be expressed as a guarded choice or the summation of a set of processes which are initially deterministic. We also introduce the concept of head normal form in this section, which is the key point for exploring the link between operational semantics and algebraic semantics.

3.1. Algebraic semantics for guarded choice

Now we explore the algebraic laws for guarded choice. The order of the guarded components in a guarded choice can be rearranged.

(gchoice-1) $[]\{C_1, \ldots\ldots, C_n\} = []\{C_{i_1}, \ldots\ldots, C_{i_n}\}$, where, $i_1, \ldots, i_n$ is a permutation of $1, \ldots, n$.

The assignment-guarded component can be eliminated if its Boolean condition is always false.

(gchoice-2) $[]\{[p]choice\{\mathbf{false}\&(x := e)\, P,\ G_1\},\ G_2\} = []\{[p]choice\{G_1\},\ G_2\}$

[1] The notation $[]\{P_1, \ldots, P_n\}[][]\{Q_1, \ldots, Q_m\}$ stands for $[]\{P_1, \ldots, P_n, Q_1, \ldots, Q_m\}$.

Two assignment-guarded components can be combined into a single one if the assignment guards and their subsequent processes are the same. This can be represented by combining two Boolean conditions together as disjunction.

(gchoice-3) $[]\{[p]choice\{b_1\&(x := e)\,P,\ b_2\&(x := e)\,P,\ G_1\},\ G_2\} = []\{[p]choice\{(b_1 \vee b_2)\&(x := e)\,P,\ G_1\},\ G_2\}$

Two event-guarded components can be combined into a single one if their subsequent processes are the same. This combination can be represented by an or-compound guard.

(gchoice-4) $[]\{@b\,P,\ @c\,P,\ G\} = []\{@(b \vee c)\,P,\ G\}$

The self-assignment can be represented by any program variables.

(gchoice-5) $[]\{[p]choice\{b\&(x := x)\,P,\ G_1\},\ G_2\} = []\{[p]choice\{(b\&(y := y)\,P,\ G_1\},\ G_2\}$

Two same choice components can be combined into a single one. The associated probability for the combined component is the addition of the probabilities of the two separate ones.

(gchoice-6) $[]\{[p]choice\{G_1\},\ [q]choice\{G_1\},\ G_2\} = []\{[p+q]choice\{G_1\},\ G_2\}$

3.2. Algebraic semantics for sequential constructs

In this section we list the laws for sequential constructs. The laws below indicate that a sequential program can be represented as a guarded choice. We have the following considerations.

Assignment can be expressed as a guarded choice composed of only one guarded component with probability 1.

(assign-1) $x := e = []\{\,[1]choice\{\mathbf{true}\&(x := e)\ \varepsilon\}\,\}$

$\mathbf{Skip} = []\{\,[1]choice\{\mathbf{true}\&(x := x)\ \varepsilon\}\,\}$

An event guard can be expressed as a guarded choice composed of one event guard component. Similar consideration also applies to a time-delay guard.

(guard-1) $@b = []\{@b\ \varepsilon\}$

(delay-1) $\#1 = []\{\#1\ \varepsilon\}$

$\#n = []\{\#1\ \#(n-1)\}, \quad \text{where } n > 1$

Conditionals can also be expressed as a guarded choice composed of only one choice component with probability 1. The choice component has two assignment-guarded subcomponents. Iteration has a similar structure.

(cond-1) $\mathbf{if}\ b\ \mathbf{then}\ P\ \mathbf{else}\ Q = []\{\,[1]choice\{b\&(x := x)\ P,\ \neg b\&(x := x)\ Q\}\,\}$

(itera-1) $\mathbf{while}\ b\ \mathbf{do}\ P = []\{\,[1]choice\{b\&(x := x)\ (P\,;\ \mathbf{while}\ b\ \mathbf{do}\ P),\ \neg b\&(x := x)\ \varepsilon\}\,\}$

If program P is expressed as a guarded choice, its sequential composition with Q is also a guarded choice, where each component is the combination of the corresponding component in P with Q via the function **seq** defined below.

(seq-2) Assume $P = []\{C_1, \ldots, C_n\}$, then $P\,;\ Q = []\{\mathbf{seq}(C_1, Q), \ldots, \mathbf{seq}(C_n, Q)\}$.

where,

$$\mathbf{seq}(C, Q) =_{df} \begin{cases} [p]choice_{j\in J}\{b_j\&(x_j := e_j)\ \mathbf{seq1}(P_j, Q)\} & \mathbf{if}\quad C = [p]choice_{j\in J}\{b_j\&(x_j := e_j)\ P_j\} \\ @b\ \mathbf{seq1}(P', Q) & \mathbf{if}\quad C = @b\,P' \\ \#1\ \mathbf{seq1}(P', Q) & \mathbf{if}\quad C = \#1\,P' \end{cases}$$

$$\mathbf{seq1}(P, Q) =_{df} \begin{cases} P\,;\ Q & \mathbf{if}\quad P \neq \varepsilon \\ Q & \mathbf{if}\quad P = \varepsilon \end{cases}$$

The definition of $\mathbf{seq}(C, Q)$ is based on the type of guarded component C. The subsequent behaviour after the corresponding guard can be expressed via the function **seq1**. Here $\mathbf{seq1}(X, Q)$ stands for Q if X is the empty process. Otherwise, it stands for $X; Q$.

3.3. Algebraic laws for parallel construct

Probabilistic parallel composition is not purely symmetric and associative. Its symmetry and associativity rely on the change of the associated probabilities as well.

(par-1) $P \parallel_p Q = Q \parallel_{1-p} P$

(par-2) $P \parallel_p (Q \parallel_q R) = (P \parallel_x Q) \parallel_y R, \qquad \text{where, } x = p/(p+q-p\times q) \text{ and } y = p+q-p\times q.$

For (par-2), the formulae for x and y indicate that the probability for an assignment in the left hand side to be scheduled is the same as the probability of the corresponding assignment on the right hand side. For example, for an assignment in

process Q of the left hand side process, its probability is $(1-p)\times q$. For the corresponding assignment in process Q of the right hand side process, its probability is $(1-x)\times y = (1-p/(p+q-p\times q))\times(p+q-p\times q) = (1-p)\times q$.

Now we give the definition for the function **par**, which can be used in reducing the number of parallel expansion laws relating to empty process. It can also be used to reduce the number of transition rules for parallel composition.

$$\mathbf{par}(P,Q,p_1) =_{df} \begin{cases} P \parallel_{p_1} Q & \text{if } P \neq \varepsilon \text{ and } Q \neq \varepsilon \\ P & \text{if } P \neq \varepsilon \text{ and } Q = \varepsilon \\ Q & \text{if } P = \varepsilon \text{ and } Q \neq \varepsilon \\ \varepsilon & \text{if } P = \varepsilon \text{ and } Q = \varepsilon \end{cases}$$

In what follows we give a collection of parallel expansion laws. As mentioned earlier, there are five types of guarded choice. To take into account a parallel composition of two arbitrary guarded choices, we end up with 25 different expansion laws. Due to the above symmetric law (associated with probability), these 25 expansion laws can be reduced into 15 different laws. Here we highlight some expansion laws with the rest listed in the Appendix.

For a parallel process, if the first component is an assignment-guarded choice and the second component is also an assignment-guarded choice, the scheduling rule is that any assignment could be scheduled with the associated probability provided that its Boolean condition is satisfied. Law (par-3-1) below depicts this case.

(par-3-1) Let $P = []_{i\in I}\{[p_i]\, choice_{j\in J_i}(b_{ij}\&(x_{ij} := e_{ij})\, P_{ij})\}$ and $Q = []_{k\in K}\{[q_k]\, choice_{l\in L_k}(b_{kl}\&(x_{kl} := e_{kl})\, Q_{kl})\}$

Then

$$\begin{aligned} P \parallel_r Q \;=\; & []_{i\in I}\{[r\times p_i]\, choice_{j\in J_i}(b_{ij}\&(x_{ij} := e_{ij}) \;\; \mathbf{par}(P_{ij}, Q, r)\} \\ & [][]_{k\in K}\{[(1-r)\times q_k]\, choice_{l\in L_k}(b_{kl}\&(x_{kl} := e_{kl}) \;\; \mathbf{par}(P, Q_{kl}, r)\} \end{aligned}$$

If the first component is an assignment-guarded choice and the second component is an event-guarded choice, the behaviour of the parallel composition can be described as the guarded choice of a set of assignment-guarded components and a set of event-guarded components. This case is presented in law (par-3-2).

(par-3-2) Let $P = []_{i\in I}\{[p_i]\, choice_{j\in J_i}(b_{ij}\&(x_{ij} := e_{ij})\, P_{ij})\}$ and $Q = []_{k\in K}\{@c_k\, Q_k\}$

Then

$$\begin{aligned} P \parallel_r Q \;=\; & []_{i\in I}\{[p_i]\, choice_{j\in J_i}(b_{ij}\&(x_{ij} := e_{ij})\, \mathbf{par}(P_{ij}, Q, r)\} \\ & [][]_{k\in K}\{@c_k\, \mathbf{par}(P, Q_k, r)\} \end{aligned}$$

If both of the two components are event-guarded choices, there are several scenarios. If one guard from the first component is triggered but no guards from the second component are triggered, the subsequent behaviour is the parallel composition of the remaining part of the first component with the second component. If two guards from both components are triggered simultaneously, the subsequent behaviour is the parallel composition of the remaining processes of both sides. This is illustrated in law (par-3-6).

(par-3-6) Let $P = []_{i\in I}\{@b_i\, P_i\}$ and $Q = []_{j\in J}\{@c_j\, Q_j\}$

Then

$$\begin{aligned} P \parallel_r Q = \; & []_{i\in I}\{@(b_i\wedge\neg c)\, \mathbf{par}(P_i, Q, r)\} \\ & [][]_{j\in J}\{@(c_j\wedge\neg b)\, \mathbf{par}(P, Q_j, r)\} \\ & [][]_{i\in I\wedge j\in J}\{@(b_i\wedge c_j)\, \mathbf{par}(P_i, Q_j, r)\} \end{aligned}$$

where, $b = \vee_{i\in I}\, b_i$ and $c = \vee_{j\in J}\, c_j$

3.4. Algebraic laws for probabilistic nondeterministic choice and nondeterministic choice

Probabilistic nondeterministic choice can be expressed as a guarded choice comprising two assignment-guarded components. The probability for the selection of one guarded component is p, whereas the probability for the selection of another component is $1-p$.

(pnonde-1) $P \sqcap_p Q \;=\; []\{\,[p]choice\{\mathbf{true}\&(x := x)\, P\},\;\; [1-p]choice\{\mathbf{true}\&(x := x)\, Q\}\}$

Now we introduce a new construct, called summation, which is denoted as $\bigoplus\{P_1,\cdots,P_n\}$, where each P_i is initially deterministic. For summation, the selection among all components P_i is nondeterministic. For a process whose outmost structure is not parallel structure, it is called *initially deterministic* if its outmost structure is not nondeterministic choice. On the other hand, if the outmost structure of a process is parallel composition, it is called initially deterministic if both of its two parallel components (i.e., left hand side and right hand side of parallel composition) are initially deterministic.

For example, let $P = x := 1 \sqcap x := 2$ and $Q = y := 1 \sqcap y := 2$. Then P can be expressed as $\bigoplus\{x := 1,\ x := 2\}$ and Q can be expressed as $\bigoplus\{y := 1,\ y := 2\}$. Further, $P \parallel_r Q$ can be expressed as $\bigoplus\{x := 1 \parallel_r y := 1,\ x := 1 \parallel_r y := 2,\ x := 2 \parallel_r y := 1,\ x := 2 \parallel y := 2\}$ and every element in this summation set is initially deterministic. Further, we have the law below for $\bigoplus$.

($\bigoplus$-1) $\bigoplus\{P_1, \ldots, P_n\} = \bigoplus\{P_{i_1}, \ldots, P_{i_n}\}$, where, $i_1, \ldots, i_n$ is the permutation of $1, \ldots, n$.

For $\sqcap$, it also shares the symmetric law and associative law. Now we consider the transformation of nondeterministic choice to the form of summation.

(nonde-1) If $P = \bigoplus\{P_1, \ldots, P_n\}$ and $Q = \bigoplus\{Q_1, \ldots, Q_m\}$, then $P \sqcap Q = \bigoplus\{P_1, \ldots, P_n, Q_1, \ldots, Q_m\}$

The law below indicates how we can transform sequential composition into the form of summation.

(seq-3) If $P = \bigoplus\{P_1, \ldots, P_n\}$, then $P ; Q = \bigoplus\{(P_1; Q), \ldots, (P_n; Q)\}$

3.5. Head normal form

Now we assign every program P a normal form called head normal form $\mathcal{HF}(P)$ [18]. Our later discussion about relating algebraic semantics with operational semantics is based on head normal form.

(1) $\mathcal{HF}(x := e) =_{df} []\{[1]choice\{\mathbf{true}\&(x := e)\ \varepsilon\}\}$

$\mathcal{HF}(\mathbf{Skip}) =_{df} []\{[1]\{\mathbf{true}\&(x := x)\ \varepsilon\}\}$

(2) $\mathcal{HF}(@b) =_{df} []\{@b\ \varepsilon\}$

(3) $\mathcal{HF}(\#1) =_{df} []\{\#1\ \varepsilon\}$

$\mathcal{HF}(\#n) =_{df} []\{\#1\ \#(n-1)\}$, where $n > 1$

(4) $\mathcal{HF}(\mathbf{if}\ b\ \mathbf{then}\ P\ \mathbf{else}\ Q) =_{df} []\{[1]choice\{b\&(x := x)\ P,\ \neg b\&(x := x)\ Q\}\}$

(5) $\mathcal{HF}(\mathbf{while}\ b\ \mathbf{do}\ P) =_{df} []\{[1]choice\{b\&(x := x)\ (P ;\ \mathbf{while}\ b\ \mathbf{do}\ P),\ \neg b\&(x := x)\ \varepsilon\}\}$

(6) If P is a guarded choice, then $\mathcal{HF}(P) =_{df} P$

(7) If $\mathcal{HF}(P) = []\{C_1, \ldots, C_n\}$, then $\mathcal{HF}(P ; Q) = []\{\mathbf{seq}(C_1, Q), \ldots, \mathbf{seq}(C_n, Q)\}$

If $\mathcal{HF}(P) = \bigoplus_{i\in I} P_i$ and $|I| > 1$, then $\mathcal{HF}(P ; Q) =_{df} \bigoplus_{i\in I} (P_i ; Q)$

(8) $\mathcal{HF}(P \sqcap_p Q) =_{df} []\{[p]choice\{\mathbf{true}\&(x := x)\ P\}, [1-p]choice\{\mathbf{true}\&(x := x)\ Q\}\}$

(9) If $\mathcal{HF}(P) = \bigoplus\{P_1, \ldots, P_n\}$ and $\mathcal{HF}(Q) = \bigoplus\{Q_1, \ldots, Q_n\}$,
then $\mathcal{HF}(P \sqcap Q) =_{df} \bigoplus\{P_1, \ldots, P_n, Q_1, \ldots, Q_n\}$

The head normal form of parallel composition can be expressed as the summation of a set of processes that are initially deterministic. We can take one component from one parallel branch and another component from another parallel branch. Then the guarded choice can be defined by applying the corresponding parallel expansion laws based on the selected two components.

(10) If $\mathcal{HF}(P) = \bigoplus_{i\in I} P_i$ and $\mathcal{HF}(Q) = \bigoplus_{j\in J} Q_j$,
then $\mathcal{HF}(P \parallel_r Q) =_{df} \bigoplus_{i\in I, j\in J}(P_i \parallel_r Q_j)$
For $\mathcal{HF}(P_i \parallel_r Q_j)$, it can be defined as the result of applying the parallel expansion laws for $HF(P_i) \parallel_r HF(Q_j)$, as illustrated in Section 3.3.

Example 3.1. Let $U =_{df} \mathbf{if}\ x > 1\ \mathbf{then}\ P_1\ \mathbf{else}\ P_2$, $V =_{df} y := x + 1 ;\ Q$, $W =_{df} z := x + 1 ;\ R$, $S =_{df} (U \sqcap V) \parallel_{0.7} W$
Now we consider the head normal form of $\mathcal{HF}(S)$. From the definition of head normal form, we know:

$$\mathcal{HF}((U \sqcap V) \parallel_{0.7} W) = U \parallel_{0.7} W \oplus V \parallel_{0.7} W$$

Moreover, we have:

$$\begin{aligned}
&\mathcal{HF}(U \parallel_{0.7} W)\\
=\ &[]\{[0.7]choice\{x > 1\&(x := x)\ (P_1 \parallel_{0.7} W), x \leq 1\&(x := x)\ (P_2 \parallel_{0.7} W)\},\\
&\quad [0.3]choice\{\mathbf{true}\&(z := x + 1)\ (U \parallel_{0.7} R)\}\}\\
&\mathcal{HF}(V \parallel_{0.7} W)\\
=\ &[]\{[0.7]choice\{\mathbf{true}\&(y := x + 1)\ (Q \parallel_{0.7} W)\},\\
&\quad [0.3]choice\{\mathbf{true}\&(z := x + 1)\ (V \parallel_{0.7} R)\}\}
\end{aligned}$$

□

From the definition of head normal form, we know that it is in the form of one step forward expansion. This can support the derivation of operational semantics from head normal form because operational semantics also makes one step forward for each transition. Further, the definition of head normal form indicates that the head normal form of a program can be derived from the algebraic laws.

4. Animation of algebraic semantics and head normal form

As mentioned earlier, head normal form can be applied in linking the algebraic semantics and operational semantics. Our approach is to derive the operational semantics from algebraic semantics, which will be explored in Section 5. In order to explore the mechanical (i.e., automatic) derivation of the operational semantics from the algebraic semantics, we consider the animation of the algebraic semantics and head normal form (i.e., the simulation of the algebraic semantics and automatic generation of the head normal form) in this section. The animation can be processed in Prolog.

Programs written in Prolog are composed of clauses describing relations. Rules and facts are the two types of clause. A rule is in the form of "$H :- B_1, B_2, \ldots, B_n$.", with H as its head and $B_1, B_2, \ldots, B_n$ as its body. This can be read as "H is true if B_1 and B_2 and ... and B_n are true". Facts are clauses with an empty body in the form of "H.". The execution of a Prolog program is triggered when the user proposes a query in the form "$? - G_1, G_2, \ldots, G_n$", where G_i is a goal clause.

For variables in Prolog, if a variable begins with an underscore (_), it can be instantiated as any value. A single underscore, which means "any term", is usually used to denote an anonymous variable. A list in Prolog is used for holding a group of terms and is defined inductively. A list is denoted as $[A|B]$ where A is the first element of the list and B is the tail, which is also a list. [ ] is used to denote an empty list. A list can also be in the form $[A_1, A_2, \ldots, A_n]$, where all A_i are its elements, separated by commas.

4.1. Generating algebraic laws

Now we explore the generation of algebraic laws for our probabilistic language using an animation style. Here we mainly focus on the parallel expansion laws, which enable us to expand a probabilistic parallel composition to a guarded choice construct.

To animate these laws, we define the structure of the five types of probabilistic guarded choice first. We denote the assignment-guarded choices as *assignGuardChoice*, event-guarded choices as *eventGuardChoice*, time delays as *timeDelay*. *assign_event* is used to represent the guarded choice composition of the assignment-guarded choice and the event-guarded choice. The guarded choice composition of the event-guarded choice and time delays is denoted as *event_time*.

For the *choice* construct $[Pr]choice_{j\in J}(b_j\&(V_j := E_j)\ S_j)$, it can be expressed as a set of components in the form $Pr\ for\ b_j\ then\ (V_j = E_j)\$S_j$ when animating. Therefore, for a guarded choice, when producing an implementation in Prolog, the components can be of three forms: $Pr\ for\ EB\ then\ (V = E)\$S$ (assignment under condition EB with probability Pr and subsequent behaviour S), $@EB\$S$ (event guard component), and $\#1\$S$ (time-delay component).

$assignGuardChoice([[Pr\ for\ EB\ then\ (V = E)\ \$\ S]|S']) :- bool(EB), variable(V), expr(E), probability(Pr), assignGuardChoice(S').$

$eventGuardChoice([[@EB\ \$\ S]|S']) :- bool(EB), eventGuardChoice(S').$

$timeDelay([[\#1\ \$\ S]]).$

$assign\_event(S) :- assignGuardChoice(S_1),\ eventGuardChoice(S_2),\ append(S_1, S_2, S).$

$event\_time(S) :- eventGuardChoice(S_1),\ timeDelay(S_2),\ append(S_1, S_2, S).$

Here, $bool(EB)$ is defined as a set of rules in Prolog to check if EB is a legal expression in *PTSC* with a Boolean value [3]. Similarly, $variable(V)$ is use to ensure that V is in a proper form of variable and $expr(E)$ guarantees that E is a legal expression.

The *assignGuardChoice* form can be defined recursively. It indicates that the first guarded component is an assignment-guarded component and the remaining components still satisfy *assignGuardChoice* form. The probability Pr is defined as a

variable whose value is a number in the range of [0,1] and its validity is checked by rule *probability*(*Pr*). The *eventGuardChoice* indicates the event-guarded choice and is defined similarly. For *timeDelay*, it only contains the time-delay component. For *assign_event*, it combines the *assignGuardChoice* form and *eventGuardChoice* form by using the *append* function. *append*(S_1, S_2, ...S_n, S) concatenates list S_1, S_2 and S_n into list S.

As mentioned earlier, there are five types of guarded choice and 25 parallel expansion laws. Now we are ready to generate the parallel expansion laws by giving the definition for $npar(S_1 \|_R S_2, T)$. Here, T stands for the expansion form for parallel process $S_1 \|_R S_2$ by using the corresponding parallel expansion laws. First we simulate parallel expansion law (par-3-1) (see page 7), which stands for the case that both of the two parallel branches are of the form of assignment-guarded choice. This indicates that both of the two parallel branches are of type *assignGuardChoice*. In this case, rules $assign_2L$ and $assign_2R$ will be called.

$$npar(S_1 \|_R S_2, T) :- assignGuardChoice(S_1),\ assignGuardChoice(S_2), assign_2L(S_1 \|_R S_2, T_1), assign_2R(S_1 \|_R S_2, T_2), append(T_1, T_2, T).$$

For $S_1 \|_R S_2$, when S_1 and S_2 are assignment-guarded choices, any assignment in S_1 and S_2 can be scheduled. $assign_2L(S_1 \|_R S_2, T_1)$ generates the guarded components stored in T_1 when the assignment in S_1 is scheduled. Meanwhile, $assign_2R(S_1 \|_R S_2, T_2)$ generates the guarded components stored in T_2 when the assignment in S_2 is scheduled. Their definitions can be formulated recursively. Below is the definition of $assign_2L(S_1 \|_R S_2, T_1)$. For $assign_2R(S_1 \|_R S_2, T_2)$, its definition is similar.

$$assign_2L([[Pr\ for\ EB_1\ then\ (V_1 = E_1)\ \$\ S_1]] \|_R S_2, [[(Pr * R)\ for\ EB_1\ then\ (V_1 = E_1)\ \$\ (S_1 \|_R S_2)]]).$$

$$assign_2L([[Pr\ for\ EB_1\ then\ (V_1 = E_1)\ \$\ S_1]|S_1'] \|_R S_2, T) :- assign_2L([[Pr\ for\ EB_1\ then\ (V_1 = E_1)\ \$\ S_1]] \|_R S_2, T_1), assign_2L(S_1' \|_R S_2, T_2), append(T_1, T_2, T).$$

Now we consider the generation and animation for the parallel expansion law where both of the two parallel branches are of the form of event-guarded choice. The law is illustrated in law (par-3-6) (see page 7).

For $S_1 \|_R S_2$, if both S_1 and S_2 are of the form of event-guarded choice, the following $npar(S_1 \|_R S_2, T)$ generates the event-guarded components stored in T.

$$npar(S_1 \|_R S_2, T) :- eventGuardChoice(S_1),\ eventGuardChoice(S_2), event_2L(S_1 \|_R S_2, T_1), event_2R(S_1 \|_R S_2, T_2), event_2Both(S_1 \|_R S_2, T_3), append(T_1, T_2, T_3, T).$$

From algebraic law (par-3-6) (see page 7), we know that there are three firing cases. The *append* function combines T_1, T_2 and T_3 into one single list T sequentially. $event_2L(S_1 \|_R S_2, T_1)$ stands for the case where one event in S_1 is fired and all events in S_2 are not fired at this moment. Then the resultant event-guarded components are recorded in T_1.

$$event_2L([[\_Pr\ for\ \_EB_1\ then\ (\_V_1 = \_E_1)\ \$\ \_S_1]|S_1'] \|_R S_2,\ T) :- event_2L(S_1' \|_R S_2, T).$$

$$event_2L([[\#1\ \$\ \_S_1]] \|_R \_S_2,\ [\,]).$$

$$event_2L([[@EB_1\ \$\ S_1]] \|_R S_2, [[@(EB_1\ \wedge\ B)\ \$\ (S_1 \|_R S_2)]]) :- compound(S_2, B).$$

$$event_2L([[@EB_1\ \$\ S_1]|S_1'] \|_R S_2, T) :- event_2L([[@EB_1\ \$\ S_1]] \|_R S_2, T_1), event_2L(S_1' \|_R S_2, T_2), append(T_1, T_2, T).$$

In the above definitions, $compound(S_2, B)$ combines all the events in S_2 in the disjunction form and stores the result in B.

Similarly, $event_2R(S_1 \|_R S_2, T_2)$ stands for the case where one event in S_2 is fired and all events in S_1 are not fired at this point. Then the resultant event-guarded components are recorded in T_2. Its definition is similar to $event_2L(S_1 \|_R S_2, T_1)$.

Thirdly, $event_2Both(S_1 \|_R S_2, T_3)$ stands for the case in which one event from S_1 and another event from S_2 are fired simultaneously. Then the resultant event-guarded components are recorded in T_3.

$$event_2Both([[@\_EB_1\ \$\ \_S_1]|\_S_1'] \|_R [[\#1\ \$\ \_S]],\ [\,]).$$

$$event_2Both([[@EB_1\ \$\ S_1]|S_1'] \|_R [[\_Pr\ for\ \_EB_2\ then\ (\_V_2 = \_E_2)\ \$\ \_S_2]|S_2'], T) :- event_2Both([[@EB_1\ \$\ S_1]|S_1'] \|_R S_2', T).$$

$$event_2Both([[\#1\ \$\ \_S]] \|_R [[@\_EB_1\ \$\ \_S_1]|\_S_1'],\ [\,]).$$

$$event_2Both([[\_Pr\ for\ \_EB_1\ then\ (\_V_1 = \_E_1)\ \$\ \_S_1]|S_1'] \|_R S_2, T) :- event_2Both(S_1' \|_R S_2, T).$$

$$event_2Both([[@EB_1\ \$\ S_1]] \|_R\ [[@EB_2\ \$\ S_2]],\ [[@(EB_1\ \wedge\ EB_2)\ \$\ (S_1 \|_R S_2)]]).$$

$$event_2Both([[@EB_1\ \$\ S_1]] \|_R [[@EB_2\ \$\ S_2]|S_2'],\ T) :- event_2Both([[@EB_1\ \$\ S_1]] \|_R [[@EB_2\ S_2]], T_1), event_2Both([[@EB_1\ \$\ S_1]] \|_R\ S_2', T_2), append(T_1, T_2, T).$$

$$event_2Both([[@EB_1\ \$\ S_1]|S_1'] \|_R\ [[@EB_2\ \$\ S_2]|S_2'],\ T) :- event_2Both([[@EB_1\ \$\ S_1]] \|_R [[@EB_2\ \$\ S_2]|S_2'], T_1), event_2Both(S_1' \|_R [[@EB_2\ \$\ S_2]|S_2'], T_2), append(T_1, T_2, T).$$

The following parallel expansion law captures the case where both of the two components are time-delay guarded constructs. The whole process performs a time delay and then behaves as the parallel composition of the remaining parts from both sides:

(par-3-10) Let $P = []\{\#1\ R\}$ and $Q = []\{\#1\ T\}$
Then
$$P \parallel_r Q = []\{\#1\ \mathbf{par}(R, T, r)\}$$

For the animation, the *npar* function below defines the calculation of the guarded choice for this case.

$npar([[\#1\ \$\ S_1]] \parallel_R [[\#1\ \$\ S_2]], [[\#1\ \$\ (S_1 \parallel_R S_2)]])$.

We now consider the parallel composition where the first parallel branch comprises both assignment-guarded components and event-guarded components. The following law (par-3-14) captures the scenario in which the second parallel branch consists of event-guarded components and the time delay.

(par-3-14) Let $P = []_{i\in I}\{[p_i]\ choice_{j\in J_i}(b_{ij}\&(x_{ij} := e_{ij})\ P_{ij})\}$
$[][]_{k\in K}\{@b_k\ R_k\}$
and $Q = []_{l\in L}\{@c_l\ Q_l\}[]\{\#1\ T\}$
Then $P \parallel_r Q = []_{i\in I}\{[p_i]\ choice_{j\in J}(b_{ij}\&(x_{ij} := e_{ij})\ \mathbf{par}(P_{ij}, Q, r))\}$
$[][]_{k\in K}\{@(b_k \wedge \neg c)\ \mathbf{par}(R_k, Q, r)\}$
$[][]_{l\in L}\{@(c_l \wedge b)\ \mathbf{par}(P, Q_l, r)\}$
$[][]_{k\in K\wedge l\in L}\{@(b_k \wedge c_l)\ \mathbf{par}(R_k, Q_l, r)\}$
where, $b = \vee_{k\in K}\ b_k$ and $c = \vee_{l\in L}\ c_l$

For $S_1 \parallel_R S_2$, if S_1 is in the form of combining the assignment-guarded choice and the event-guarded choice and S_2 is in the form of combining the event-guarded choice and the time-delay choice, then time delay will not be scheduled initially. It can only have event-guarded components and assignment-guarded components. We use $npar(S_1 \parallel_R S_2,\ T)$ to generate these components stored in T.

$npar(S_1 \parallel_R S_2, T) :- assign\_event(S_1), event\_time(S_2), trans1\_2(S_1 \parallel_R S_2, T_1), event_2L(S_1 \parallel_R S_2, T_2), event_2R(S_1 \parallel_R S_2, T_3),$
$event_2Both(S_1 \parallel_R S_2, T_4), append(T_1, T_2, T_3, T_4, T).$

Here, *assign_event*(S_1) and *event_time*(S_2) stand for the guarded choice types of S_1 and S_2 respectively. For algebraic law (par-3-14), there are three firing cases. Therefore, *event*$_2$*L*, *event*$_2$*R* and *event*$_2$*Both* have also been applied. As S_1 contains assignment-guarded components, $assign_1L(S_1 \parallel_R S_2, T)$ stands for the case where assignment can be scheduled first, which keeps the scheduling probability unchanged.

$assign_1L([[Pr\ for\ EB_1\ then\ (V_1 = E_1)\ \$\ S_1]] \parallel_R S_2, [[Pr\ for\ EB_1\ then\ (V_1 = E_1)\ \$\ (S_1 \parallel_R S_2)]])$.

$assign_1L([[Pr\ for\ EB_1\ then\ (V_1 = E_1)\ \$\ S_1]|S_1'] \parallel_R S_2, T)$
$:- assign_1L([[Pr\ for\ EB_1\ then\ (V_1 = E_1)\ \$\ S_1] \parallel_R S_2, T_1), assign_1L(S_1' \parallel_R S_2, T_2), append(T_1, T_2, T).$

In this section we have considered the mechanical generation of algebraic laws using an animation approach. Among the 25 parallel expansion laws, here we only listed four of them (i.e., (par-3-1), (par-3-6), (par-3-10), (par-3-14)) for the purpose of introducing the animation approach. The consideration for the animation of other parallel expansion laws is similar.

4.2. Generating head normal forms

With the aim of linking the algebraic semantics and the operational semantics, we have introduced the concept of head normal form for every program. In this section, we explore how to generate the head normal form via the animation approach.

For program P, we use the function below to generate the head normal form of P:

$$hf(P,\ T)$$

where, T stands for the head normal form of program P. Here T can be expressed as the summation of a set of guarded choices.[2] We use a list to represent the summation.

Assignment can be expressed as a guarded choice composed of only one guarded component with probability 1.

(1) $hf\ (V = E,\quad [\ [\ 1\ for\ true\ then\ V = E\ \$\ epsilon\ ]\ ])$.[3]

The whole clause means that the head normal form of $V = E$ is $[[1\ for\ true\ then\ V = E\ \$\ epsilon]]$, which is a list containing only one element.

An event guard can be expressed as a guarded choice composed of one event guard component. Similar consideration also applies to a time-delay guard.

[2] If there is only one guarded choice in the summation, the summation can also be understood as a form of guarded choice.
[3] In Sections 4 and 6, assignment $V := E$ is rewritten as $V = E$ when doing the animation.

(2) $hf\ (@EB,\ [\ [\ @EB\ \$\ epsilon\ ]\ ]).$

$hf\ (\#1,\ [\ [\ \#1\ \$\ epsilon\ ]\ ]).$

$hf\ (\#N,\ [\ [\ \#1\ \$\ \#(N-1)\ ]\ ])\ :\!-\ N > 1.$

Conditional can also be expressed as a guarded choice composed of two assignment-guarded subcomponents with probability 1. Iteration has a similar structure.

(3) $hf\ (if\ (EB)\ \$\ S_1\ else\ S_2,\ [\ [\ 1\ for\ EB\ then\ [\ ]\ \$\ S_1\ ],\ [\ 1\ for\ \sim EB\ then\ [\ ]\ \$\ S_2\ ]\ ]).$

$hf\ (while\ (EB)\ \$\ S,\ [\ [\ 1\ for\ EB\ then\ [\ ]\ \$\ (\ S;\ while\ EB\ \$\ S)\ ],\ [\ 1\ for\ \sim EB\ then\ [\ ]\ \$\ epsilon\ ]\ ]).$

Here [ ] stands for the empty assignment which does nothing in the state update. "$\sim EB$" stands for the logical formula $\neg EB$.

The head normal form of guarded choice has the same form as the guard choice itself.

(4) $hf(S, S)\ :\!-\ pgc(S).$

Here $pgc(S)$ indicates that S is in the form of guarded choice.

A process may have nondeterministic behaviours. We can express a process as the summation of a set of guarded choices. Each guarded choice can be regarded as an initially deterministic process for the whole program.

(5) $hf\ (S_1 \sqcap S_2, T)\ :\!-\ summation\ (S_1 \sqcap S_2, T).$

The summation rules are designed to differentiate a nondeterministic process from the others. A nondeterministic process will be represented by a list whose elements are initially deterministic.

$summation\ (L_1 \sqcap L_2, T)\ :\!-\ summation\ (L_1, T_1),\ summation\ (L_2, T_2),\ append\ (T_1, T_2, T).$

$summation\ (S, [S]).$

Now we start to consider the calculation of the head normal form for sequential composition. The definition can be done in two steps. If a process is initially deterministic, we first translate it to its head normal form and then distribute ";" to each component of it. The distribution behaviour can be performed by a new function *subdistr*.

(6) $hf\ ([\ ];\ \_Q, [\ ])\ :\!-\ !.$

$hf\ (S;\ Q, T)\ :\!-\ hf\ (S, S'), subdistr\ (S';\ Q, T).$

Below is the definition for *subdistr*. Here we only list some of the rules.

$subdistr\ ([\ ];\ \_, [\ ]).$

$subdistr\ ([Pr\ for\ EB\ then\ (V = E)\ \$\ S];\ Q, [[Pr\ for\ EB\ then\ (V = E)\ \$\ (S;\ Q)]]).$

$subdistr\ ([@EB\ \$\ S];\ Q, [[@EB\ \$\ (S;\ Q)]]).$

$subdistr\ ([\#1\ \$\ S];\ Q, [[\#1\ \$\ (S;\ Q)]]).$

For a nondeterministic process, it can be transformed into a set of guarded choices. Then the sequential composition of the nondeterministic process and Q can be transformed into a summation comprising of the sequential composition of each guarded choice and Q.

$hf\ (S_1 \sqcap S_2;\ Q, T)\ :\!-\ summation\ (S_1 \sqcap S_2, S), distr(S;\ Q, T).$

For each guarded choice in the new summation, the subsequent behaviour after the corresponding guard can be handled by introducing the function *distr*.

$distr\ ([X];\ Q, [X;\ Q]).$

$distr\ ([X|S];\ Q, T)\ :\!-\ distr\ ([X];\ Q, T_1),\ distr\ (S;\ Q, T_2), append\ (T_1, T_2, T).$

Now we consider the generation of head normal form for the parallel process $S_1 \parallel_R S_2$. First we need to calculate the components that are initially deterministic for S_1 and S_2 respectively; i.e., each initially deterministic component is represented by a guarded choice. This can be done by using the two *summation* functions shown below.

(7) $hf(S_1 \parallel_R S_2, T)\ :\!-\ summation(S_1, S_1'), summation(S_2, S_2'), combination(S_1' \parallel_R S_2', T).$

In the above definition, the head normal form of $S_1 \parallel_R S_2$ is the summation of a set of new guarded choice using the *combination* function. Here each new guarded choice is the parallel composition of one guarded choice from S_1' and another guarded choice from S_2'. This can be processed by using the algebraic laws (mainly using the function *npar*) in Section 4. Below is the definition of the *combination* function.

$combination([X] \|_R [Y], T) :- npar(X \|_R Y, T).$

$combination([X] \|_R [Y|S_2], T) :- combination([X] \|_R [Y], T_1), combination([X] \|_R S_2, T_2), append(T_1, T_2, T).$

$combination([X|S_1] \|_R [Y|S_2], T) :- combination([X] \|_R [Y|S_2], T_1), combination(S_1 \|_R [Y|S_2], T_2), append(T_1, T_2, T).$

Example 4.1. Let $P = []\{[1]choice(true\&(a = 10)\ a = 11\}$
$\|_{0.2}\ []\{\{@(b = 3)\ b = 0, @(b = 4)\ b = 6\}[]\{\#1\ b = 5\}\}$

Then the query for the head normal form of P is in the form $hf(P, NF)$, where P is the program text we introduced above and NF is a variable storing the head normal form. Here, the program P is the parallel composition of *assignGuardChoice* and *event_time*; its head normal form is defined using expansion law (par-3-5) (given in the Appendix).

$? - hf(P, NF).$
$NF = [[1\ for\ true\ then\ a = 10\ \$\ (a = 11\ \|_{0.2}\ [[@b = 3\ \$\ b = 0], [@b = 4\ \$\ b = 6], [\#1\ \$\ b = 5]])],$
$[@b = 3\ \$\ ([[1\ for\ true\ then\ a = 10\ \$\ a = 11]]\ \|_{0.2}\ b = 0)],$
$[@b = 4\ \$\ ([[1\ for\ true\ then\ a = 10\ \$\ a = 11]]\ \|_{0.2}\ b = 6)]]$

5. Deriving operational semantics from algebraic semantics

The traditional way of defining an operational semantics is to provide a set of individual transition steps directly. In contrast to the standard style of defining operational semantics, this section derives the operational semantics from the algebraic semantics for our probabilistic language. This approach aims to guarantee the consistency of the operational and algebraic semantics for our language.

5.1. Transition types

The operational semantics of a language is represented by transition relations. In our operational model, the transitions are expressed in the form of Plotkin's Structural Operational Semantics (SOS) [45]:

$$\langle P,\ \sigma\rangle \xrightarrow{\beta} \langle P',\ \sigma'\rangle$$

where, P stands for the program text that remains to be executed, and σ is the current state of the program.
The transitions can be classified into four kinds:

(1) The first kind of transition models the execution of an atomic action with a certain probability.
$$\langle P,\ \sigma\rangle \xrightarrow{c}_p \langle P',\ \sigma'\rangle$$
where, p stands for the probability of program P performing the execution.
(2) The second type models the transition of a time delay. Time advances in unit steps.
$$\langle P,\ \sigma\rangle \xrightarrow{1} \langle P',\ \sigma'\rangle$$
(3) The third type models the selection of the two components for nondeterministic choice. It can be expressed as:
$$\langle P,\ \sigma\rangle \xrightarrow{\tau} \langle P',\ \sigma\rangle$$
(4) The fourth type models the triggered case of event @ b:
$$\langle P,\ \sigma\rangle \xrightarrow{v} \langle P',\ \sigma\rangle$$

5.2. Derivation strategy

The main purpose of this section is to derive the transition system for our probabilistic language from its algebraic laws. This approach allows the operational semantics to be derived as theorems (see Section 5.3), rather than being presented as postulates or definitions. First we give the derivation strategy.

Definition 5.1 (Derivation strategy)**.** Let $\mathcal{HF}(P) = \bigoplus_{i\in I} P_i$.

(1) If $|I| > 1$, then $\langle P,\ \sigma\rangle \xrightarrow{\tau} \langle P_i,\ \sigma\rangle\ (i \in I)$.
(2) Otherwise,
(a) If $\mathcal{HF}(P) = []_{i\in I}\{[p_i]\ choice_{j\in J_i}(b_{ij}\&(x_{ij} := e_{ij})\ P_{ij})\}$,then $\langle P,\ \sigma\rangle \xrightarrow{c}_{p_i} \langle P_{ij},\ \sigma[e_{ij}/x_{ij}]\rangle$, if $b_{ij}(\sigma)$
(b) If $\mathcal{HF}(P) = []_{i\in I}\{@b_i\ P_i\}$, then $\langle P,\ \sigma\rangle \xrightarrow{v} \langle P_i,\ \sigma\rangle$, if $b_i(\sigma)$
$\langle P,\ \sigma\rangle \xrightarrow{1} \langle P,\ \sigma\rangle$, if $\bigwedge_{i\in I} \neg b_i(\sigma)$

(c) If $\mathcal{HF}(P) = []\{\#1\, R\}$, then $\langle P,\ \sigma\rangle \xrightarrow{1} \langle R,\ \sigma\rangle$.

(d) If $\mathcal{HF}(P) = []_{i\in I}\{[p_i]\, choice_{j\in J}(b_{ij}\&(x_{ij} := e_{ij})\, P_{ij})\}[][]_{k\in K}\{@c_k\, Q_k\}$,

then $\langle P,\ \sigma\rangle \xrightarrow{v} \langle Q_k,\ \sigma\rangle$, if $c_k(\sigma)$

$\langle P,\ \sigma\rangle \xrightarrow{c}_{p_i} \langle P_{ij},\ \sigma[e_{ij}/x_{ij}]\rangle$, if $b_{ij}(\sigma) \wedge (\bigwedge_{k\in K} \neg c_k(\sigma))$

(e) If $P = []_{i\in I}\{@b_i\, P_i\}[]\{\#1\, R\}$,

then $\langle P,\ \sigma\rangle \xrightarrow{v} \langle P_i,\ \sigma\rangle$, if $b_i(\sigma)$

$\langle P,\ \sigma\rangle \xrightarrow{1} \langle R,\ \sigma\rangle$, if $\bigwedge_{i\in I} \neg b_i(\sigma)$ □

Item (1) indicates that if a program can be expressed as the summation of a set of processes that are initially deterministic, then the program can make a nondeterministic selection among all these processes. Item (2) considers the transition rules for the case that a program can be expressed as a guarded choice. This can be divided into five subcases, depending on the types of guarded choice.

If a program can be expressed as a guarded choice comprised of a set of assignment-guarded components, then the program can execute each assignment with the associated probability provided that the corresponding Boolean condition is satisfied. If a program can be expressed as a guarded choice composed of a set of event-guarded components, any event can be triggered provided that the event is satisfied. In the case that none of these events are satisfied, time will advance one unit. If a program can only be expressed as a guarded choice composed of the time-delay component, only time advancing can be executed.

On the other hand, if a program can be expressed as a guarded choice composed of a set of assignment-guarded components and a set of event guard components, any event can be triggered, provided that the event is satisfied. Further, if none of these events are satisfied, any assignment can be executed with the associated probability provided that its Boolean condition is satisfied. On the other hand, time cannot advance for a program in this case.

Finally, if a program is expressed as a guarded choice composed of a set of assignment-guarded components and the time-delay component, any event can be triggered provided that the event is satisfied. On the other hand, if none of the events are satisfied, time can advance by one unit. The subsequent behaviour after one time unit has elapsed is just the behaviour followed the time-delay event.

5.3. Deriving operational semantics by strict proof

This section aims to derive the operational semantics for all the statements according to the derivation strategy. Thus, our operational semantics can be considered as consistent with its algebraic semantics.

Theorem 5.2.

(1) $\langle x := e,\ \sigma\rangle \xrightarrow{c}_1 \langle \varepsilon,\ \sigma[e/x]\rangle$

(2) $\langle$**if** b **then** P **else** Q , $\sigma\rangle \xrightarrow{c}_1 \langle P,\ \sigma\rangle$, *if* $b(\sigma)$

$\langle$**if** b **then** P **else** Q , $\sigma\rangle \xrightarrow{c}_1 \langle Q,\ \sigma\rangle$, *if* $\neg b(\sigma)$

(3) $\langle$**while** b **do** $P,\ \sigma\rangle \xrightarrow{c}_1 \langle P$; **while** b **do** $P,\ \sigma\rangle$, *if* $b(\sigma)$

$\langle$**while** b **do** $P,\ \sigma\rangle \xrightarrow{c}_1 \langle \varepsilon,\ \sigma\rangle$, *if* $\neg b(\sigma)$

(4) $\langle \#n,\ \sigma\rangle \xrightarrow{1} \langle \#(n-1),\ \sigma\rangle$, *where* $n > 1$.

$\langle \#1,\ \sigma\rangle \xrightarrow{1} \langle \varepsilon,\ \sigma\rangle$

(5) $\langle @b\, P,\ \sigma\rangle \xrightarrow{v} \langle P,\ \sigma\rangle$, *if* $b(\sigma)$

$\langle @b\, P,\ \sigma\rangle \xrightarrow{1} \langle @b\, P,\ \sigma\rangle$, *if* $\neg b(\sigma)$

(6) $\langle P \sqcap_p Q,\ \sigma\rangle \xrightarrow{c}_p \langle P,\ \sigma\rangle$

$\langle P \sqcap_p Q,\ \sigma\rangle \xrightarrow{c}_{1-p} \langle Q,\ \sigma\rangle$

Proof. The proof can be proceeded directly from the head normal form of each program and the derivation strategy. Here we give the proof for (6). Others are similar. The head normal form of $P \sqcap_p Q$ is as below:

$$\mathcal{HF}(P \sqcap_p Q) =_{df} []\{[p]choice\{\mathbf{true}\&(x := x)\, P\}, [1-p]choice\{\mathbf{true}\&(x := x)\, Q\}\}$$

By the derivation strategy 2(a), we can directly get the transition rules for $P \sqcap_p Q$. □

The transitions in (1), (2), and (3) model the case that a program does an atomic action with probability 1. For the time delay statement, only time advancing can be performed. For an event guard, it can be immediately fired provided that the Boolean condition is satisfied. However, if the Boolean condition is not satisfied, time will advance by one unit. Item (6) models the transition for probabilistic nondeterministic choice. For $P \sqcap_p Q$, the selection of P is with probability p and the selection of Q is with probability $1 - p$.

We can also have the transition rules for sequential composition, which are the same as those in a traditional programming language. The proof is based on the head normal form of sequential composition and the derivation strategy.

Theorem 5.3. *If* $\langle P,\ \sigma\rangle \xrightarrow{\beta} \langle P',\ \sigma'\rangle$, *then* $\langle P; Q,\ \sigma\rangle \xrightarrow{\beta} \langle \mathbf{seq1}(P', Q),\ \sigma'\rangle$.

To simplify the later proof for the transition rules of other constructs, we introduce a function $GC(P)$ for program P based on $\mathcal{HF}(P)$. Let $\mathcal{HF}(P) = \bigoplus_{i\in I} P_i$. The definition of $GC(P)$ can be defined as below:

$$GC(P) =_{df} \begin{cases} \{P\} & \text{if } |I| = 1 \\ \cup_{i\in I}\{P_i\} & \text{if } |I| > 1 \end{cases}$$

Moreover, if P is initially deterministic, $GC1(P)$ is introduced, which contains all the guarded components in $\mathcal{HF}(P)$.

Further, we define two functions:

$stable(\langle P, \sigma\rangle) =_{df} \neg(\ \langle P, \sigma\rangle \xrightarrow{\tau}\ )$ and

$stableE(\langle P, \sigma\rangle) =_{df} \neg(\ \langle P, \sigma\rangle \xrightarrow{v}\ )$

The notation $stable(\langle P, \sigma\rangle)$ indicates that process P cannot perform the transition representing nondeterministic choice under state σ, while $stableE(\langle P, \sigma\rangle)$ indicates that process P cannot perform event-triggered transitions under state σ.

Now we consider the derivation of the transitions for nondeterministic choice. A nondeterministic process can perform a "τ" transition and directly reach a stable state. '

Theorem 5.4.

(1) *If* $\langle P,\ \sigma\rangle \xrightarrow{\tau} \langle P',\ \sigma'\rangle$, *then* $\langle P \sqcap Q,\ \sigma\rangle \xrightarrow{\tau} \langle P',\ \sigma'\rangle$
$\langle Q \sqcap P,\ \sigma\rangle \xrightarrow{\tau} \langle P',\ \sigma'\rangle$

(2) *If* $stable(\langle P, \sigma\rangle)$, *then* $\langle P \sqcap Q,\ \sigma\rangle \xrightarrow{\tau} \langle P,\ \sigma'\rangle$
$\langle Q \sqcap P,\ \sigma\rangle \xrightarrow{\tau} \langle P,\ \sigma'\rangle$

Proof. Here we give the proof for (1). The proof for (2) is similar.

$$\begin{aligned} & \langle P,\ \sigma\rangle \xrightarrow{\tau} \langle P',\ \sigma'\rangle && \{\text{Derivation Strategy}\} \\ \Rightarrow\ & P' \in GC(P) \wedge |GC(P)| > 1 && \{\text{Set Calculus}\} \\ \Rightarrow\ & P' \in GC(P) \cup GC(Q)\ \wedge |GC(P) \cup GC(Q)| > 1 && \{\text{Derivation Strategy}\} \\ \Rightarrow\ & \langle P \sqcap Q,\ \sigma\rangle \xrightarrow{\tau} \langle P',\ \sigma'\rangle && \square \end{aligned}$$

Next we explore the derivation of the transition rules for parallel composition. The approach is based on the four transition types of a program.

Theorem 5.5.

(1) (a) *If* $\langle P,\ \sigma\rangle \xrightarrow{\tau} \langle P',\ \sigma\rangle$ *and* $stable(\langle Q, \sigma\rangle)$,
then $\langle P \parallel_{p_1} Q,\ \sigma\rangle \xrightarrow{\tau} \langle \mathbf{par}(P', Q, p_1),\ \sigma\rangle$.
$\langle Q \parallel_{p_1} P,\ \sigma\rangle \xrightarrow{\tau} \langle \mathbf{par}(Q, P', p_1),\ \sigma\rangle$.

(b) *If* $\langle P,\ \sigma\rangle \xrightarrow{\tau} \langle P',\ \sigma,\rangle$ *and* $\langle Q,\ \sigma\rangle \xrightarrow{\tau} \langle Q',\ \sigma\rangle$,
then $\langle P \parallel_{p_1} Q,\ \sigma\rangle \xrightarrow{\tau} \langle \mathbf{par}(P', Q', p_1),\ \sigma\rangle$

(2) (a) *If* $\langle P,\ \sigma\rangle \xrightarrow{v} \langle P',\ \sigma\rangle$ *and* $stable(\langle Q, \sigma\rangle)$ *and* $stableE(\langle Q, \sigma\rangle)$,
then $\langle P \parallel_{p_1} Q,\ \sigma\rangle \xrightarrow{v} \langle \mathbf{par}(P', Q, p_1),\ \sigma\rangle$.
$\langle Q \parallel_{p_1} P,\ \sigma\rangle \xrightarrow{v} \langle \mathbf{par}(Q, P', p_1),\ \sigma\rangle$.

(b) *If* $\langle P,\ \sigma\rangle \xrightarrow{\nu} \langle P',\ \sigma\rangle$ *and* $\langle Q,\ \sigma\rangle \xrightarrow{\nu} \langle Q',\ \sigma\rangle$,

then $\langle P \parallel_{p_1} Q,\ \sigma\rangle \xrightarrow{\nu} \langle \mathbf{par}(P', Q', p_1),\ \sigma\rangle$

(3) *If* $\langle P,\ \sigma\rangle \xrightarrow{c}_{p_2} \langle P',\ \sigma'\rangle$ *and stable*$(\langle x, \sigma\rangle)$ *and stableE*$(\langle x, \sigma\rangle)$ $(x = P,\ Q)$,

then $\langle P \parallel_{p_1} Q,\ \sigma\rangle \xrightarrow{c}_{p_1\times p_2} \langle \mathbf{par}(P', Q, p_1),\ \sigma'\rangle$

$\langle Q \parallel_{p_1} P,\ \sigma\rangle \xrightarrow{c}_{(1-p_1)\times p_2} \langle \mathbf{par}(Q, P', p_1),\ \sigma'\rangle$

(4) *If* $\langle P,\ \sigma\rangle \xrightarrow{1} \langle P',\ \sigma'\rangle$ *and* $\langle Q,\ \sigma\rangle \xrightarrow{1} \langle Q',\ \sigma'\rangle$ *and stable*$(\langle x, \sigma\rangle)$ *and stableE*$(\langle x, \sigma\rangle)$ $(x = P,\ Q)$,

then $\langle P \parallel_{p_1} Q,\ \sigma\rangle \xrightarrow{1} \langle \mathbf{par}(P', Q', p_1),\ \sigma'\rangle$.

Proof. Firstly we consider the proof for 1(a). The proof for 1(b) is similar.

$$\langle P,\ \sigma\rangle \xrightarrow{\tau} \langle P',\ \sigma\rangle \qquad \{\text{Derivation Strategy}\}$$

$$\Rightarrow P' \in GC(P) \wedge |GC(P)| > 1 \wedge |GC(Q)| = 1 \qquad \{\mathcal{HF} \text{ for Parallel Composition}\}$$

$$\Rightarrow P' \parallel Q \in GC(P \parallel_{p_1} Q) \qquad \{\text{Derivation Strategy}\}$$

$$\Rightarrow \langle P \parallel_{p_1} Q,\ \sigma\rangle \xrightarrow{\tau} \langle \mathbf{par}(P', Q, p_1),\ \sigma\rangle$$

Secondly, we consider the proof for 2(a). As $\langle P,\ \sigma\rangle \xrightarrow{\nu} \langle P',\ \sigma\rangle$, this indicates that P is stable currently. From the assumption, we also know that Q cannot perform τ transition and ν transition. We can enumerate all the parallel expansion cases. Here we only consider the proof for the case below.

Let $\mathcal{HF}(P) = []_{k\in K}\{[q_k]\ choice_{l\in L_k}(b_{kl}\&(x_{kl} := e_{kl})\ P_{kl})\}[][]_{m\in M}\{@c_m\ R_m\}$

$\mathcal{HF}(Q) = []_{i\in I}\{[p_i]\ choice_{j\in J_i}(b_{ij}\&(x_{ij} := e_{ij})\ Q_{ij})\}$

From the expansion laws, we know:

$$@c_k\ \mathbf{par}(R_m, Q, p_1) \in GC1(P \parallel_{p_1} Q)$$

This indicates that the corresponding transition can be derived from the derivation strategy.

Thirdly, we consider the proof for item (3). We use the above example case for exploring the proof.

From $\langle P,\ \sigma\rangle \xrightarrow{c} \langle P_{kl},\ \sigma, \rangle$, we can have:

$$\forall m \bullet (c_m(\sigma) = \mathbf{false}) \wedge b_{kl}(\sigma) \tag{1}$$

$$[q_k]\ choice_{l\in L_k}(b_{kl}\&(x_{kl} := e_{kl})\ P_{kl}) \in GC1(P) \wedge \forall m \bullet (@c_m\ R_m) \in GC1(P) \tag{2}$$

By the parallel expansion laws, we know:

$$[q_k \times p]\ choice_{l\in L_k}(b_{kl}\&(x_{kl} := e_{kl})\ \mathbf{par}(P_{kl}, Q, p)) \in GC1(P \parallel_{p_1} Q) \wedge$$
$$\forall m \bullet (@c_m\ \mathbf{par}(R_m, Q, p)) \in GC1(P \parallel_{p_1} Q) \tag{3}$$

Based on (1) and (3), we can derive the transition rule for item (3).

Finally, we consider the proof for item (4). We use the example case below for exploration.

$\mathcal{HF}(P) = []_{i\in I}\{@b_i\ P_i\}[]\{\#1\ R\}$,

$\mathcal{HF}(Q) = []_{j\in J}\{@c_j\ Q_j\}[]\{\#1\ T\}$

From the parallel expansion law, we can have:

$$\#1\mathbf{par}(R, T, p_1) \in GC1(P \parallel_{p_1} Q)$$

This indicates that $P \parallel_{p_1} Q$ can perform the required time-delay transition. □

For the above derived transition rules for parallel composition, transition (1)(a) stands for the case that one component makes a nondeterministic choice and another component is stable. The whole process also performs a nondeterministic choice in this case. However, if both components make nondeterministic choices, then the whole process can perform a nondeterministic choice and the subsequent behaviour is the parallel composition of the remaining components. Transition (1)(b) reflects this situation.

The second type stands for the event-fired case. The analysis is similar to the transitions of type (1). Transition (3) covers the case of performing an atomic action. If the process P can perform an atomic action with probability p_2, then process $P \parallel_{p_1} Q$ and $Q \parallel_{p_1} P$ can also perform the same atomic action with probability $p_1 \times p_2$ and $(1 - p_1) \times p_2$ respectively.

If both components can perform a time-advancing transition, then the whole parallel process can also let time advance. The aspect is reflected in transition (4).

This section has considered the derivation of an operational semantics from algebraic semantics for our proposed probabilistic language. The derivation strategy is based on the head normal form of each process. A transition system (i.e., operational semantics) has been derived for programs based on the derivation strategy by strict proof. This gives us a way to demonstrate the consistency of the algebraic semantics and operational semantics.

5.4. Equivalence of derivation strategy and transition system

In the previous sections, we have derived a set of transition rules following to a derivation strategy. The set of transition rules can be considered as a transition system (i.e., operational semantics) for our probabilistic language. However, there remains an issue to be considered. *The derivation strategy may derive more transitions, compared with our transition system* (Theorems 5.2–5.5). We want to demonstrate that the set of transitions derived from the derivation strategy is the same as the set of transitions generated from our transition system. If so, we can say the derivation strategy is equivalent to the transition system.

The advantage of this equivalence result is that we can use either the derivation strategy or the transition system when working on the specific application of operational semantics. This will simplify effort in the application of the operational semantics.

In order to study this equivalence issue, we need to prove the following items for every process.

(1) If transition $\langle P, \alpha\rangle \xrightarrow{\beta} \langle P', \alpha'\rangle$ exists in the *transition system*, then it also exists in the *derivation strategy*.
(2) If transition $\langle P, \alpha\rangle \xrightarrow{\beta} \langle P', \alpha'\rangle$ exists in the *derivation strategy*, then it also exists in the *transition system*.

Item (1) above is correct because our transition system is derived from the derivation strategy. Now we consider (2) as a theorem to be proved.

Theorem 5.6. *If transition* $\langle P, \alpha\rangle \xrightarrow{\beta} \langle P', \alpha'\rangle$ *exists in the derivation strategy, then it also exists in the transition system.*

Proof. We can proceed with our proof using structural induction. Here we only give the proof for parallel composition. For simplicity, we give the proof for the following example case where both of the head normal forms of P and Q can be expressed as a guarded choice respectively.

$$\begin{aligned}
\text{Let}\quad \mathcal{HF}(P) &= []_{i\in I}\{[p_i]\, choice_{j\in J_i}(b_{ij}\&(x_{ij} := e_{ij})\, P_{ij})\} \\
&\quad\; [][]_{k\in K}\{@\hat{b}_k\, R_k\} \\
\mathcal{HF}(Q) &= []_{l\in L}\{[q_l]\, choice_{m\in M_l}(c_{lm}\&(y_{lm} := f_{lm})\, Q_{lm})\} \\
&\quad\; [][]_{n\in N}\{@\hat{c}_n\, T_n\}
\end{aligned}$$

Then

$$\begin{aligned}
\mathcal{HF}(P \parallel_r Q) &= []_{i\in I}\{[r \times p_i]\, choice_{j\in J_i}(b_{ij}\&(x_{ij} := e_{ij})\, \mathbf{par}(P_{ij}, Q, r))\} \\
&\quad [][]_{l\in L}\{[(1-r) \times q_l]\, choice_{m\in M_l}(c_{lm}\&(y_{lm} := f_{lm})\mathbf{par}(P, Q_{lm}, r))\} \\
&\quad [][]_{k\in K}\{@(\hat{b}_k \wedge \neg\hat{c})\, \mathbf{par}(R_k, Q, r)\} \\
&\quad [][]_{n\in N}\{@(\hat{c}_n \wedge \neg\hat{b})\, \mathbf{par}(P, T_n, r)\} \\
&\quad [][]_{k\in K\wedge n\in N}\{@(\hat{b}_k \wedge \hat{c}_n)\, \mathbf{par}(R_k, Q_n, r)\}
\end{aligned}$$

where, $\hat{b} = \vee_{k\in K}\, \hat{b}_k$ and $\hat{c} = \vee_{n\in N}\, \hat{c}_n$

In this case, from the derivation strategy, P can perform the following transitions:

(p-1) $\langle P,\ \sigma\rangle \xrightarrow{v} \langle R_k,\ \sigma\rangle$, if $\hat{b}_k(\sigma)$
(p-2) $\langle P,\ \sigma\rangle \xrightarrow{c}_{p_i} \langle P_{ij},\ \sigma[e_{ij}/x_{ij}]\rangle$, if $\neg\hat{b}(\sigma) \wedge b_{ij}(\sigma)$

On the other hand, from the derivation strategy, Q can also perform similar transitions, shown below:

(q-1) $\langle Q,\ \sigma\rangle \xrightarrow{v} \langle T_n,\ \sigma\rangle$, if $\hat{c}_n(\sigma)$
(q-2) $\langle Q,\ \sigma\rangle \xrightarrow{c}_{q_l} \langle Q_{lm},\ \sigma[f_{lm}/y_{lm}]\rangle$, if $\neg\hat{c}(\sigma) \wedge c_{lm}(\sigma)$

By structural induction, the above transitions for P and Q also exist in our transition. From $\mathcal{HF}(P \parallel_r Q)$ and the derivation strategy, $P \parallel_r Q$ can perform and only perform the following transitions, which are derived using the derivation strategy.

(l-1) $\langle P \parallel_r Q,\ \sigma\rangle \xrightarrow{v} \langle \mathbf{par}(R_k, Q, r),\ \sigma\rangle$, if $\hat{b}_k(\sigma) \wedge \neg\hat{c}(\sigma)$

(l-2) $\langle P \parallel_r Q,\ \sigma\rangle \xrightarrow{v} \langle \mathbf{par}(P, T_n, r),\ \sigma\rangle$, if $\hat{c}_n(\sigma) \wedge \neg\hat{b}(\sigma)$

(l-3) $\langle P \parallel_r Q,\ \sigma\rangle \xrightarrow{v} \langle \mathbf{par}(R_k, T_n, r),\ \sigma\rangle$, if $\hat{b}(\sigma) \wedge \hat{c}_n(\sigma)$

(l-4) $\langle P \parallel_r Q,\ \sigma\rangle \xrightarrow{c}_{r\times p_i} \langle \mathbf{par}(P_{ij}, Q, r),\ \sigma[e_{ij}/x_{ij}]\rangle$, if $\neg\hat{b}(\sigma) \wedge \neg\hat{c}(\sigma) \wedge b_{ij}(\sigma)$

(l-5) $\langle P \parallel_r Q,\ \sigma\rangle \xrightarrow{c}_{(1-r)\times q_l} \langle \mathbf{par}(P, Q_{lm}, r),\ \sigma[f_{lm}/y_{lm}]\rangle$, if $\neg\hat{b}(\sigma) \wedge \neg\hat{c}(\sigma) \wedge c_{lm}(\sigma)$

Now we want to demonstrate that the above transitions for $P \parallel_r Q$ also exist in the transition system. Here we give the proof for *(l-1)*. Others are similar.

$\langle P,\ \sigma\rangle \xrightarrow{v} \langle R_k,\ \sigma\rangle$ exists in the derivation strategy and $\neg\hat{c}(\sigma)$

$\Rightarrow$ {By Structural Induction and PL}

$\langle P,\ \sigma\rangle \xrightarrow{v} \langle R_k,\ \sigma\rangle$ exists in the transition system and

$stable(\langle Q, \sigma\rangle) \wedge stableE(\langle Q, \sigma\rangle)$

$\Rightarrow$ {Transition System}

$\langle P \parallel_r Q,\ \sigma\rangle \xrightarrow{v} \langle \mathbf{par}(R_k, Q, r),\ \sigma\rangle$ exists in the transition system □

Now we can present the main result of this section.

Theorem 5.7. *Regarding the derived operational semantics for our probabilistic language, the derivation strategy is equivalent to the transition system.*

This result demonstrates that the transitions from the derivation strategy are the same as those in the transition system. It also shows that our transition system (operational semantics) for our probabilistic language is complete with respect to the derivation strategy in Section 5.2.

6. Animation approaches to operational semantics

In Section 5.3, we have derived an operational semantics for *PTSC*, which forms a transition system of our language (Theorems 5.2–5.5). Now in Section 6.1, we explore the animation of operational semantics for *PTSC*, which is the executable version of the operational semantics. The correctness of the operational semantics can be checked by using various test results.

Meanwhile, in Section 5.2, we provided the derivation strategy for obtaining the operational semantics from the algebraic semantics. Now in Section 6.2 we explore the corresponding mechanical derivation, which can automatically derive the transitions of a program, as well as animate the execution of a program based on the derivation strategy. Using the simulated execution of the two animation approaches in Sections 6.1 and 6.2, the fact formulated by Theorem 5.7 can be shown through various test examples.

6.1. Animation of operational semantics for PTSC

6.1.1. Animation strategy and transition type

Next we start to explore the animation of the operational semantics for *PTSC*. With this aim, the transitions are expressed in the form below:

$$[P,\ Sigma]\ -[\beta]\rightarrow\ [P',\ Sigma']$$

where, configuration $[P,\ Sigma]$ can be expressed as a list in Prolog. Here, P stands for the program text that remains to be executed. *Sigma* is the current state of the program, which is represented in the form of list storing the values of program variables. $-[\beta]\rightarrow$ stands for the transition type.

As mentioned before (see page 13), the transitions for *PTSC* can be classified into four kinds. The first type can be encoded in Prolog as:

$$[P,\ Sigma]\ -['c', R]\rightarrow\ [P',\ Sigma'].$$

where, $'c'$ stands for the transition where the atomic action is caused by the program itself and R stands for the probability for program P to perform the execution.

For the other three kinds of transition, when encoded in Prolog, we use the notations $-[1]\rightarrow$, $-['tau']\rightarrow$ and $-['v']\rightarrow$ to stand for transitions $\xrightarrow{1}$, $\xrightarrow{\tau}$ and $\xrightarrow{v}$ respectively.

For animation, the transition for assignment $V := E$ can be expressed as below.

$$\frac{Sigma\_ = Sigma \otimes (V = E)}{[V = E, Sigma] -['c', 1]\rightarrow [epsilon, Sigma\_].}$$

Here, $\otimes$ stands for the overriding operator. $Sigma \otimes (V = E)$ stands for the new state, where the new value of V overrides the previous value of V by expression E.

Similarly, we can also provide the animation for the transitions of conditional, iteration, nondeterministic choice, and sequential composition. In the subsequent sections, we mainly focus on the animation for the rules of guarded choice and parallel composition.

6.1.2. Guarded choice

Guarded choice can perform assignment with the associated probability provided that its corresponding Boolean condition is satisfied. The corresponding transition $-['c', Pr]\rightarrow$ may appear in two types of guarded choice constructs. One is the guarded choice composed of a set of assignment-guarded components. For Pr *for EB then* $(V = E)$ \$ S, the program can execute assignment $V := E$ with probability Pr when the corresponding condition EB is evaluated true in the current state. The first two rules below handle this case recursively. The other one is the guarded choice composed of assignment-guarded components and event-guarded components. In this case, the event-guarded component has a relatively high priority, which means when both the conditions of an event-guarded component and that of an assignment-guarded component are true in the current state, we only allow the event to be fired and do not allow assignment to be performed. To handle this, we design the third rule and add the "$/-['v']\rightarrow$ " term in conditional clauses.

$$\frac{EB\ \$\ (Sigma)\ \wedge\ Sigma\_ = Sigma \otimes (V = E) \wedge\ [S', Sigma]\ /-['v']\rightarrow\ [\_, Sigma]}{[[[Pr\ for\ EB\ then\ (V = E)\ \$\ S]|S'], Sigma] -['c', Pr]\rightarrow [S, Sigma\_].}$$

$$\frac{EB\ \$\ (Sigma)\ \wedge\ [S', Sigma] -['c', Pr']\rightarrow [S_1, Sigma\_] \wedge\ [S', Sigma]\ /-['v']\rightarrow\ [\_, Sigma]}{[[[\_Pr\ for\ EB\ then\ (\_V = \_E)\ \$\ \_S] \mid S'], Sigma] -['c', Pr']\rightarrow [S_1, Sigma\_].}$$

$$\frac{[S', Sigma] -['v']\rightarrow [S_1, Sigma]}{[[[\_Pr\ for\ \_EB\ then\ (\_V = \_E)\ \$\ \_S]|S'], Sigma] -['v']\rightarrow [S_1, Sigma].}$$

Here, condition EB \$ $(Sigma)$ stands for the Boolean value of expression EB at the state $Sigma$ and the notation "$/-[\beta]\rightarrow$" indicates that transition $-[\beta]\rightarrow$ cannot be performed in the current state. For configuration $[\_, Sigma]$, this indicates that the program part can be of any form.

Now we consider the event firing transition. The corresponding transition $-['v']\rightarrow$ may appear in three types of guarded choice. The first type is the guarded choice composed of a set of event-guarded components. The second type is the guarded choice composed of assignment-guarded components and event-guarded components. The last rule above reveals this. The last type is the guarded choice composed of event-guarded components and time-delay component. For the first and third types, the transitions can be divided into the following cases.

$$\frac{EB\ \$\ (Sigma)}{[[[@EB\ \$\ S]|\_S'], Sigma] -['v']\rightarrow [S, Sigma].}$$

$$\frac{\sim EB\ \$\ (Sigma)\ \wedge\ [S', Sigma] -['v']\rightarrow [S_1', Sigma]}{[[[@EB\ \$\ \_S]|S'], Sigma] -['v']\rightarrow [S_1', Sigma].}$$

For a time-delay transition, as the execution of assignment is instantaneous, the second, third, and fifth types of guarded choice can perform a time-delay transition, among the five types of guarded choice. The corresponding transition $-[1]\rightarrow$ may appear in three cases. The first two transitions below model the time-delay transition for the second type of guarded choice (i.e., containing a set of event-guarded components). The fourth transition models the time-delay transition for a guarded choice containing only a time-delay component. The third transition models the fifth type of guarded choice, i.e., containing event-guarded components and a time-delay component. If no events can be enabled in the event-guarded choice, time will advance.

$$\frac{\sim EB\ \$\ (Sigma)}{[[[@EB\ \$\ S]], Sigma] -[1]\rightarrow [[[@EB\ \$\ S]], Sigma].}$$

$$\frac{\sim EB\ \$\ (Sigma)\ \wedge\ [S', Sigma] -[1]\rightarrow [S', Sigma]}{[[[@EB\ \$\ S]|S'], Sigma] -[1]\rightarrow [[[@EB\ \$\ S]|S'], Sigma].}$$

$$\frac{\sim EB\ \$\ (Sigma)\ \wedge\ [S', Sigma] \dashv[1]\mapsto [S_1, Sigma] \wedge S_1 \sim= S'}{[[[@EB\ \$\ \_S]|S'], Sigma] \dashv[1]\mapsto [S_1, Sigma].}$$

$$\frac{\textbf{true}}{[[[\#1\ \$\ S]], Sigma] \dashv[1]\mapsto [S, Sigma].}$$

6.1.3. Parallel composition

Next we consider probabilistic parallel composition. Once a parallel component has executed to termination, the terminated component will be eliminated and the whole parallel program will perform as the non-terminated parallel component.

First we consider the $['tau']$ transition. If one parallel component can perform a $['tau']$ transition and another parallel component cannot perform a $['tau']$ transition, the whole program can also perform a $['tau']$ transition. Here we only list the transitions for a parallel process when its left component performs the $['tau']$ transition.

$$\frac{[S_1, Sigma] \dashv['tau']\mapsto [epsilon, Sigma] \wedge [S_2, Sigma] \not\dashv['tau']\mapsto [\_, Sigma]}{[S_1 \parallel_R S_2, Sigma] \dashv['tau']\mapsto [S_2, Sigma].}$$

$$\frac{[S_1, Sigma] \dashv['tau']\mapsto [S_1', Sigma] \wedge [S_2, Sigma] \not\dashv['tau']\mapsto [\_, Sigma] \wedge S_1' \sim= epsilon}{[S_1 \parallel_R S_2, Sigma] \dashv['tau']\mapsto [S_1' \parallel_R S_2, Sigma].}$$

If both components make $['tau']$ transition, the whole process can make $['tau']$ transition and the subsequent behaviour is the parallel composition of the remaining components.

$$\frac{[S_1, Sigma] \dashv['tau']\mapsto [S_1', Sigma] \wedge [S_2, Sigma] \dashv['tau']\mapsto [S_2', Sigma] \wedge S_1' \sim= epsilon \wedge S_2' \sim= epsilon}{[S_1 \parallel_R S_2, Sigma] \dashv['tau']\mapsto [S_1' \parallel_R S_2', Sigma].}$$

If S_1 (or S_2, or both of them) reaches the terminating state after performing $['tau']$ transitions, the transition rule for $S_1 \parallel_R S_2$ is similar.

The following rule indicates the case of performing an atomic action with a certain probability in parallel composition. If process S_1 can perform an atomic action with probability X, the parallel process $S_1 \parallel_R S_2$ can also perform the same atomic action with probability $X \times R$. In parallel composition, we assume the $['tau']$ transition and $['v']$ transition have high priority. This is reflected in the transitions below.

$$\frac{[S_1, Sigma] \dashv['c', X]\mapsto [S_1', Sigma\_] \wedge S_1' \sim= epsilon \wedge [S_2, Sigma] \not\dashv['tau']\mapsto [\_, Sigma] \wedge [S_2, Sigma] \not\dashv['v']\mapsto [\_, Sigma]}{[S_1 \parallel_R S_2, Sigma] \dashv['c', X * R]\mapsto [S_1' \parallel_R S_2, Sigma\_].}$$

Here we only list the case that S_1 reaches the non-terminating state after performing the atomic action transition.

For the $['v']$ and $[1]$ transitions, the analysis for animation is similar.

Example 6.1. Let P be the process described in Example 4.1 (page 13). Now we consider the execution sequence using the operational semantics provided above. The query is posed in the form "*trackOP*$[P,\ Sigma]$.". The animation result is displayed as below. The process at step 0 is still process P itself. In each execution step, for example $[\ P,\ [b = 4, a = 2]\ ]$, the second element stands for the current state of the program variables, which is represented as a list. Here the notation $[b = 4, a = 2]$ stands for the state, where the value of variable b is 4 and the value of a is 2.

$? - trackOP[\ P,\ [b = 4, a = 2]\ ].$
$0 - -- > [\ P,\ [b = 4, a = 2]]$
$1 \dashv[v\ ]\mapsto [[[1\ for\ true\ then\ a = 10\ \$\ a = 11]] \parallel_{0.2} b = 6, [b = 4, a = 2]]$
$2 \dashv[c, 0.8]\mapsto [[[1\ for\ true\ then\ a = 10\ \$\ a = 11]], [b = 6, a = 2]]$
$3 \dashv[c, 1]\mapsto [a = 11, [b = 6, a = 10]]$
$4 \dashv[c,\ 1\ ]\mapsto [epsilon, [b = 6, a = 11]]$
$2 \dashv[c, 0.2]\mapsto [a = 11 \parallel_{0.2} b = 6, [b = 4, a = 10]]$
$3 \dashv[c, 0.2]\mapsto [b = 6, [b = 4, a = 11]]$
$4 \dashv[c,\ 1\ ]\mapsto [epsilon, [b = 6, a = 11]]$
$3 \dashv[c, 0.8]\mapsto [a = 11, [b = 6, a = 10]]$
$4 \dashv[c,\ 1\ ]\mapsto [epsilon, [b = 6, a = 11]]$

From the displayed result, we know that there are three execution sequences leading to the terminating state.

6.2. Animation of generating operational semantics from algebraic semantics

Section 5 considered the derivation of the operational semantics from the algebraic semantics for *PTSC*. A derivation strategy has been provided based on the head normal form. Now we consider the animation of the derivation strategy for deriving the operational semantics from the algebraic semantics.

Definition 6.2.

(1) If the head normal form of process P can be expressed as a guarded choice (i.e., it can be of form *assignGuardChoice*, *eventGuardChoice*, *timeDelay*, *assign_event* and *event_time*), then the transition rules for the process P are the same as the transition rules of its corresponding guarded choice.

(2) On the other hand, if the head normal form of process P cannot be described as one of the five types of guarded choice (in other words, the head normal form has a structure of summation), then the process P can first do $[' tau']$ transitions and reach to all the processes that are initially deterministic; these processes can be expressed as one of the five type guarded choices.

$$\frac{\sim pgc([X \mid L])}{[[X \mid L], Sigma] \dashv[' tau'] \mapsto [X, Sigma].}$$

$$\frac{L \sim= [\,] \wedge [L, Sigma] \dashv[' tau'] \mapsto [Y, Sigma]}{[[\_ \mid L], Sigma] \dashv[' tau'] \mapsto [Y, Sigma].}$$

Here, $\sim pgc([X \mid L])$ indicates that the head normal form of $[X \mid L]$ is not in the form of the five types of guarded choice, which means it is a summation.

For the transition rules of guarded choice (see page 13), these have already been formalized in Section 5.2. The second rule above stands for the τ transitions for a process if its head normal form is not in the form of the five types of guarded choice.

Now we have two ways to achieve operational semantics for *PTSC*. The first approach is the transition system itself (see Section 6.1), which directly provides the transition rules for each statement. The second approach described in this section is to derive the transition rules for each statement. The derivation strategy is based the head normal form of a process and those parallel expansion laws. This section applies the animation approach in showing that the above two approaches can achieve the same transitions for each statement. We use the example below to illustrate the animation of the operational semantics by the above two approaches.

Example 6.3. Let P be the program described in Example 4.1 (page 13) and Example 6.1 (page 20). In Example 6.1, we have already considered the execution sequence of program P using the operational semantics. The animation results have already been provided.

Now we consider the derivation of the operational semantics via algebraic semantics. The query is posted in the form "*trackHFOP*$[P,\ Sigma]$.". The process at step 0 is the head normal form of P, i.e., *NF* is the head normal form which has been explored in Example 5.1. For the derivation approach, there are also three execution sequences leading to the terminating state.

$? - trackHFOP[P,\ [b = 4, a = 2]].$

$0 - - - > [NF, [b = 4, a = 2]]$

$1 \dashv[v\,] \mapsto [[[1\ for\ true\ then\ a = 10\ \$\ a = 11]] \parallel_{0.2} b = 6, [b = 4, a = 2]]$

$2 \dashv[c, 0.8] \mapsto [[[1\ for\ true\ then\ a = 10\ \$\ a = 11]], [b = 6, a = 2]]$

$3 \dashv[c, 1] \mapsto [a = 11, [b = 6, a = 10]]$

$4 \dashv[c,\ 1\ ] \mapsto [epsilon, [b = 6, a = 11]]$

$2 \dashv[c, 0.2] \mapsto [a = 11 \parallel_{0.2} b = 6, [b = 4, a = 10]]$

$3 \dashv[c, 0.2] \mapsto [b = 6, [b = 4, a = 11]]$

$4 \dashv[c,\ 1\ ] \mapsto [epsilon, [b = 6, a = 11]]$

$3 \dashv[c, 0.8] \mapsto [a = 11, [b = 6, a = 10]]$

$4 \dashv[c,\ 1\ ] \mapsto [epsilon, [b = 6, a = 11]]$

The execution step for *NF* is based on the derivation strategy. Using the above execution result, we know that the derived operational semantics for program P is the same as the original operational semantics (Section 6.1).

7. Related work

Shared-variable concurrency (SVC) is used to model concurrency via multi-threaded programs. De Rover et al. have explored concurrency verification methods, including both compositional and non-compositional methods [10]. A shared-

variable concurrency language is introduced, where the concurrent execution is characterized by the interleaving model. Verification methods have been studied, including the assertion method, compositional proof methods, and the Hoare Logic method. However, their shared-variable language has not covered the reasoning about priority for scheduling one thread over another, or more general probabilistic choice.

Probability in sequential programming has been studied by Morgan and his colleagues [34–36]. They have explored abstraction and refinement for probabilistic processes using the weakest precondition (*wp*) approach. A set of algebraic laws has been studied for probabilistic programs. Hehner studied how probabilistic programming can be applied to the predicative style of programming [22–24], where the types of variables were interpreted as probability distributions expressed as functions and the applicability is limited to sequential programs. Another area where probabilistic aspects have been explored is for process algebra. Núñez extended Hennessy's "testing semantics" [25] for a variety of probabilistic processes [41–43]. Two processes are called testing equivalent if the probabilities with which they pass are the same. Seidel studied probabilistic communicating processes [47] by adding probability to Communicating Sequential Processes (CSP) [26]. Two semantic spaces have been explored, which act as the basis for a probabilistic variant of CSP. In [51], we proposed the model of *PTSC*, which integrates probability with time and shared-variable concurrency. Compared with the first two probabilistic explorations, our model includes concurrency. Compared with the above two probabilistic process algebraic approaches, our model has the time and shared-variable concurrency features.

Regarding the work of relating operational and algebraic semantics, Hoare and He have studied the derivation of operational semantics from algebraic semantics [27,28]. An operational semantics of CSP was derived, based on CSP's algebraic laws, following to a derivation strategy (called the action transition relation). An operational semantics of Dijkstra's Guarded Command Language (GCL) was also derived based on GCL's algebra, in accordance with a derivation strategy (called the step relation). However, the language does not contain probabilistic and event features, and the mechanical approach has not been studied.

Our work on linking the semantics for *PTSC* can be regarded as a further exploration of Unifying Theories of Programming (UTP) [28]. The UTP approach has been successfully applied in studying the semantics and algebraic laws of programming languages. Probabilistic Guarded Command Language is an extension of the Guarded Command Language with probabilistic choice. Its denotational semantics was formalized by He et al. [21] under the UTP framework. A set of algebraic laws was achieved based on the denotational semantics. New unification of probability with standard computation has been studied in which a nonzero chance of disaster is treated as a disaster [19]. However, the two probabilistic models are limited to sequential programs and do not have any timed feature.

For the semantic linking, several approaches have been investigated for the consistency between operational semantics and denotational semantics. Brookes has given a new denotational semantics for a shared-variable parallel programming language [5]. The denotational semantics is proved to be fully abstract with respect to the operational-based partial correctness behaviour. The consistency has also been investigated in the book *Control Flow Semantics* (*CFS*) [8]. *CFS* is devoted to studying the equivalence of operational semantics and denotational semantics for 27 languages using theory based on metric spaces. Hartog and his colleagues have studied the equivalence between operational and denotational semantics for a variety of probabilistic processes [11–14] using the *CFS* approach. Previously, we studied the derivation of denotational semantics from operational semantics for Verilog, based on the UTP approach [50]. The exploration showed that the derived denotational semantics is the same as the original defined denotational semantics. Our approach for the semantic linking in this paper is from a different angle. We have proposed the language *PTSC* and explored the derivation of an operational semantics from an algebraic semantics, which aims at consistency between the two semantics. Meanwhile, we also studied a mechanical logic programming approach for the semantic linking. The concept of head normal form has been applied in our work.

8. Conclusions

This paper has presented how an algebraic semantics links with an operational semantics for our proposed probabilistic language with time and shared-variable concurrency. The work is based on the unifying theories of programming, pioneered by Hoare and He [28]. This exploration includes both theoretical and practical approaches.

- We have provided algebraic laws. Our approach is new, where a process can be expressed as either a guarded choice, or the summation of a set of processes that are initially deterministic. Every guarded choice is composed of a set of guarded components. This approach with guarded choice gives us a way to sequentialize a process that also reflects the scheduling policy. This summation representation with guarded choice also gives meaning for the program.
- We have studied the derivation of the operational semantics for our language from its algebraic semantics. We have given the definition of the derivation strategy. Then a transition system (i.e., operational semantics) for our language can be derived via the derivation strategy. This gives us confidence for the soundness and consistency of the operational semantics with respect to the algebraic semantics.
- We have investigated the relationship between the derivation strategy and the derived operational semantics. We have proved that the derived operational semantics is equivalent to the derivation strategy. This tells us that we can use either the derivation strategy or the derived operational semantics for the application of the operational semantics. The result achieved here shows that our transition system (operational semantics) is complete with respect to head normal form.

Besides the above theoretical approach to the link between the operational semantics and algebraic semantics for *PTSC*, we also considered practical aspects of the link. We have explored the animation of the link between the two semantics for *PTSC*. The animated result supports the claim of the soundness and completeness of the operational semantics with respect to the algebraic laws from various test results. Our animation approach is based on the logic programming language Prolog.

- We explored the algebraic laws for *PTSC* using a mechanical approach. We mainly focused on the mechanical generation of the parallel expansion laws. Our approach is based on five types of guarded choice.
- We studied the mechanical generation of the head normal form for a program. The concept of head normal form has been used to aid the link between the algebraic semantics and the operational semantics for our considered language.
- We considered how to build the link between the operational semantics and algebraic semantics mechanically. Our approach is to implement the theoretical derivation strategy for deriving the operational semantics from the algebraic semantics. For the derived operational semantics as a whole system, we also investigated its animation.

For the future, we are continuing to work on the linking theories for various semantics of programming languages [28,49]. The denotational model for *PTSC* is much more challenging because of the three additional features. Similar theoretical and practical approaches would also be interesting for other computation models, for example, probabilistic web services [6,30,52]. Further, we are also interested in how our animation approach can be applied to system verification [1,2,31,32].

Acknowledgement

This work is supported in part by National Basic Research Program of China (No. 2011CB302904), National High Technology Research and Development Program of China (Nos. 2011AA010101 and 2007AA010302), National Natural Science Foundation of China (Nos. 61061130541 and 61021004), Macau Science and Technology Development PEARL and EAE projects (Nos. 041/2007/A3 and 072/2009/A3) and UK EPSRC Project EP/G042322.

Appendix

As mentioned in Section 3.3, there are fifteen parallel expansion laws. Five of them have already been presented in the main text (Sections 3.3 and 4.1). The rest are listed below.

(par-3-3) Let $P = []_{i\in I}\{[p_i]\, choice_{j\in J_i}(b_{ij}\&(x_{ij} := e_{ij})\, P_{ij})\}$ and $Q = []\{\#1\ R\}$
Then $P \parallel_r Q = []_{i\in I}\{[p_i]\, choice_{j\in J_i}(b_{ij}\&(x_{ij} := e_{ij})\, \mathbf{par}(P_{ij}, Q, r))\}$

(par-3-4) Let $P = []_{i\in I}\{[p_i]\, choice_{j\in J_i}(b_{ij}\&(x_{ij} := e_{ij})\, P_{ij})\}$ and
$Q = []_{k\in K}\{[q_k]\, choice_{l\in L_k}(b_{kl}\&(x_{kl} := e_{kl})\, Q_{kl})\}[][]_{m\in M}\{@c_m\ R_m\}$
Then $P \parallel_r Q = []_{i\in I}\{[r \times p_i]\, choice_{j\in J_i}(b_{ij}\&(x_{ij} := e_{ij})\, \mathbf{par}(P_{ij}, Q, r))\}$
$[][]_{k\in K}\{[(1-r) \times q_k]\, choice_{l\in L_k}(b_{kl}\&(x_{kl} := e_{kl})\, \mathbf{par}(P, Q_{kl}, r)\}$
$[][]_{m\in M}\{@c_k\ \mathbf{par}(P, R_m, r)\}$

(par-3-5) Let $P = []_{i\in I}\{[p_i]\, choice_{j\in J_i}(b_{ij}\&(x_{ij} := e_{ij})\, P_{ij})\}$ and $Q = []_{l\in L}\{@c_l\ Q_l\}[]\{\#1\ R\}$
Then $P \parallel_r Q = []_{i\in I}\{[p_i]\, choice_{j\in J_i}(b_{ij}\&(x_{ij} := e_{ij})\, \mathbf{par}(P_{ij}, Q, r))\}[][]_{l\in L}\{@c_l\ \mathbf{par}(P, Q_l, r)\}$

(par-3-7) Let $P = []_{i\in I}\{@b_i\ P_i\}$ and $Q = []\{\#1\ R\}$
Then $P \parallel_r Q = []_{i\in I}\{@b_i\ \mathbf{par}(P_i, Q, r)\}[]\{\#1\ \mathbf{par}(P, R, r)\}$

(par-3-8) Let $P = []_{i\in I}\{@b_i\ P_i\}$ and $Q = []_{j\in J}\{[q_j]\, choice_{k\in K_j}(b_{jk}\&(x_{jk} := e_{jk})\, Q_{jk})\}[][]_{l\in L}\{@c_l\ R_l\}$
Then $P \parallel_r Q = []_{j\in J}\{[q_j]\, choice_{k\in K_j}(b_{ij}\&(x_{jk} := e_{jk})\, \mathbf{par}(P_{jk}, Q, r))\}$
$[][]_{i\in I}\{@(b_i \wedge \neg c)\ \mathbf{par}(P_i, Q, r)\}[][]_{l\in L}\{@(c_l \wedge \neg b)\ \mathbf{par}(P, R_l, r)\}[][]_{i\in I\wedge l\in L}\{@(b_i \wedge c_l)\ \mathbf{par}(P_i, Q_l, r)\}$
where, $b = \vee_{i\in I}\, b_i$ and $c = \vee_{l\in L}\, c_l$

(par-3-9) Let $P = []_{i\in I}\{@b_i\ P_i\}$ and $Q = []_{j\in J}\{@c_j\ Q_j\}[]\{\#1\ R\}$
Then $P \parallel_r Q = []_{i\in I}\{@(b_i \wedge \neg c)\ \mathbf{par}(P_i, Q, r)\}[][]_{j\in J}\{@(c_j \wedge \neg b)\ \mathbf{par}(P, Q_j, r)\}$
$[][]_{i\in I\wedge j\in J}\{@(b_i \wedge c_j)\ \mathbf{par}(P_i, Q_j, r)\}[][]\{\#1\ \mathbf{par}(P, R, r)\}$
where, $b = \vee_{i\in I}\, b_i$ and $c = \vee_{j\in J}\, c_j$

(par-3-11) Let $P = []\{\#1\ T\}$ and $Q = []_{i\in I}\{[q_i]\, choice_{j\in J_i}(b_{jk}\&(x_{ij} := e_{ij})\, Q_{ij})\}[][]_{k\in K}\{@c_k\ R_k\}$
Then $P \parallel_r Q = []_{i\in I}\{[q_i]\, choice_{j\in J_i}(b_{ij}\&(x_{ij} := e_{ij})\, \mathbf{par}(P, Q_{ij}, r))\}[][]_{k\in K}\{@c_k\ \mathbf{par}(P, R_k, r)\}$

(par-3-12) Let $P = []\{\#1\ T\}$ and $Q = []_{i\in I}\{@b_i\ Q_i\}[]\{\#1\ R\}$
Then $P \parallel_r Q = []_{i\in I}\{@b_i\ \mathbf{par}(P, Q_i, r)\}[]\{\#1\ \mathbf{par}(T, R, r)\}$

(par-3-13) Let $P = []_{i\in I}\{[p_i]\, choice_{j\in J_i}(b_{ij}\&(x_{ij} := e_{ij})\, P_{ij})\}[][]_{k\in K}\{@b_k\, R_k\}$ and
$Q = []_{l\in L}\{[q_l]\, choice_{m\in M_l}(c_{lm}\&(x_{lm} := e_{lm})\, P_{lm})\}[][]_{n\in N}\{@c_n\, T_n\}$
Then $P \parallel_r Q = []_{i\in I}\{[r \times p_i]\, choice_{j\in J_i}(b_{ij}\&(x_{ij} := e_{ij})\, \mathbf{par}(P_{ij}, Q, r))\}$
$[][]_{l\in L}\{[(1-r) \times q_l]\, choice_{m\in M_l}(c_{lm}\&(x_{lm} := e_{lm})\, \mathbf{par}(P, Q_{lm}, r))\}$
$[][]_{k\in K}\{@(b_k \wedge \neg c)\, \mathbf{par}(R_k, Q, r)\}[][]_{n\in N}\{@(c_n \wedge \neg b)\, \mathbf{par}(R_k, Q, r)\}$
$[][]_{k\in K\wedge n\in N}\{@(b_k \wedge c_n)\, \mathbf{par}(R_k, Q_n, r)\}$
where, $b = \vee_{k\in K}\, b_k$ and $c = \vee_{n\in N}\, c_n$

(par-3-15) Let $P = []_{i\in I}\{@b_i\, P_i\}[]\{\#1\, R\}$ and $Q = []_{j\in J}\{@c_j\, Q_j\}[]\{\#1\, T\}$
Then $P \parallel_r Q = []_{k\in K}\{@(b_i \wedge \neg c)\, \mathbf{par}(P_i, Q, r)\}[][]_{j\in J}\{@(c_j \wedge \neg b)\, \mathbf{par}(P, Q_j, r)\}$
$[][]_{i\in I\wedge j\in J}\{@(b_i \wedge c_j)\, \mathbf{par}(P_i, Q_j, r)\}[][]\{\#1\, \mathbf{par}(R, T, r)\}$
where, $b = \vee_{i\in I}\, b_i$ and $c = \vee_{j\in J}\, c_j$

References

[1] K.R. Apt, Ten years of Hoare's logic: a survey — part 1, ACM Trans. Programming Lang. Syst. 3 (4) (1981) 431–483.
[2] K.R. Apt, Ten years of Hoare's logic: a survey part II: nondeterminism, Theoret. Comput. Sci. 28 (1984)
[3] J.P. Bowen, J. He, Q. Xu, An animatable operational semantics of the Verilog Hardware Description Language, in: Proc. ICFEM 2000: 3rd IEEE International Conference on Formal Engineering Methods, IEEE Computer Society Press, 2000, pp. 199–207.
[4] S. Brookes, A grainless semantics for parallel programs with shared mutable data, Electron. Notes Theor. Comput. Sci. 155 (2006) 277–307.
[5] S.D. Brookes, Full abstraction for a shared-variable parallel language, Inform. and Comput. 127 (2) (1996) 145–163.
[6] M. Butler, S. Ripon, Executable semantics for compensating CSP, in: Proc. EPEW 2005: International Workshop on Web Services and Formal Methods, Versailles, France, September 1–3, 2005, Lecture Notes in Computer Science, vol. 3670, Springer-Verlag, 2005, pp. 243–256.
[7] W.F. Clocksin, C.S. Mellish, Programming in Prolog, fifth ed., Springer-Verlag, 2003.
[8] J. de Bakker, E. de Vink, Control Flow Semantics, The MIT Press, 1996.
[9] F.S. de Boer, A sound and complete shared-variable concurrency model for multi-threaded Java programs, in: Proc. 9th IFIP WG 6.1 International Conference on Formal Methods for Open Object-based Distributed Systems, FMOODS'07, Springer-Verlag, 2007, pp 252–268.
[10] W.P. de Roever, F. de Boer, U. Hanneman, J. Hooman, Y. Lakhnech, M. Poel, J. Zwiers, Concurrency Verification: Introduction to Compositional and Noncompositional Methods, Cambridge University Press, 2001.
[11] J. den Hartog, Probabilistic extensions of semantic models, Ph.D. thesis, Vrije University, The Netherlands, 2002.
[12] J. den Hartog, E. de Vink, Mixing up nondeterminism and probability: a preliminary report, Electron. Notes Theoret. Comput. Sci. 22 (1999)
[13] J. den Hartog, E. de Vink, Verifying probabilistic programs using a Hoare like logic, Internat. J. Found. Comput. Sci. 40 (3) (2002) 315–340.
[14] J. den Hartog, E. de Vink, J. de Bakker, Metric semantics and full abstractness for action refinement and probabilistic choice, Electron. Notes Theoret. Comput. Sci. 40 (2001)
[15] E.W. Dijkstra, The structure of the THE-multiprogramming system, Commun. ACM 11 (1968) 341–346.
[16] P.B. Hansen, Structured multiprogramming, Commun. ACM 15 (1972) 574–578.
[17] J. He, Provably Correct Systems: Modelling of Communication Languages and Design of Optimized Compilers, The McGraw-Hill International Series in Software Engineering, 1994.
[18] J. He, An algebraic approach to the Verilog programming, in: Proc. 10th Anniversary Colloquium of UNU/IIST, Lisbon, Portugal, March 18–20, 2002, Lecture Notes in Computer Science, vol. 2757, Springer, 2003, pp 65–80.
[19] J. He, J.W. Sanders, Unifying probability, in: Proc. UTP 2006: Unifying Theories of Programming, First International Symposium, UTP 2006, Walworth Castle, County Durham, UK, February 5–7, 2006, Lecture Notes in Computer Science, vol. 4010, Springer, 2006, pp. 173–199.
[20] J. He, H. Zhu, Formalising Verilog, in: Proc. ICECS 2000: IEEE International Conference on Electronics, Circuits and Systems, IEEE Computer Society Press, 2000, pp. 412–415.
[21] J. He, K. Seidel, A. McIver, Probabilistic models for the guarded command language, Sci. Comput. Programming 28 (2–3) (1997) 171–192.
[22] E.C.R. Hehner, Predicative programming, part I, Commun. ACM 27 (2) (1984) 134–143.
[23] E.C.R. Hehner, Predicative programming, part II, Commun. ACM 27 (2) (1984) 144–151.
[24] E.C.R. Hehner, Probabilistic predicative programming, In: Proc. MPC 2004: 7th International Conference on Mathematics of Program Construction, Stirling, Scotland, UK, July 12–14, 2004, Lecture Notes in Computer Science, vol. 3125, Springer, 2004, pp. 169–185.
[25] M. Hennessy, Algebraic Theory of Processes, The MIT Press, 1988.
[26] C.A.R. Hoare, Communicating Sequential Processes, Prentice Hall International Series in Computer Science, 1985.
[27] C.A.R. Hoare, J. He, From algebra to operational semantics, Inform. Process. Lett. 45 (1993) 75–80.
[28] C.A.R. Hoare, J. He, Unifying Theories of Programming, Prentice Hall International Series in Computer Science, 1998.
[29] C.A.R. Hoare, I.J. Hayes, J. He, C. Morgan, A.W. Roscoe, J.W. Sanders, I.H. Sørensen, J.M. Spivey, B. Sufrin, Laws of programming, Commun. ACM 38 (8) (1987) 672–686.
[30] F. Leymann, Web Services Flow Language (WSFL 1.0), IBM, 2001. Available from: <http://www-3.ibm.com/software/solutions/webservices/pdf/WSDL.pdf>.
[31] Z. Manna, A. Pnueli, The Temporal Logic of Reactive and Concurrent Systems: Specification, Springer-Verlag, 1992.
[32] Z. Manna, A. Pnueli, Temporal Verification of Reactive Systems: Safety, Springer-Verlag, 1995.
[33] J. Manson, W. Pugh, S.V. Adve, The Java memory model, Principles Programming Lang. (POPL) (2005) 378–391.
[34] A. McIver, C. Morgan, Partial correctness for probabilistic demonic programs, Theoret. Comput. Sci. 266 (1–2) (2001) 513–541.
[35] A. McIver, C. Morgan, Abstraction, Refinement and Proof of Probability Systems. Monographs in Computer Science, Springer, 2004.
[36] A. McIver, C. Morgan, K. Seidel, Probabilistic predicate transformers, ACM Trans. Programming Lang. Syst. 18 (3) (1996) 325–353.
[37] R. Motwani, P. Raghavan, Randomized Algorithms, Cambridge University Press, 1995.
[38] U. Ndukwu, J.W. Sanders, Reason about a distributed probabilistic system, Tech. Rep. 401, UNU/IIST, P.O. Box 3058, Macau SAR, China, 2008.
[39] U. Ndukwu, J.W. Sanders, Reasoning about a distributed probabilistic system, in: Proc. CATS 2009: Fifteenth Australasian Symposium on Computing: The Australasian Theory, vol. 94, Australian Computer Society, Wellington, New Zealand, 2009, pp. 35–42.
[40] N. Nissanke, Realtime Systems, Prentice Hall International Series in Computer Science, 1997.
[41] M. Núñez, Algebraic theory of probabilistic processes, J. Logic Algebr. Programming 56 (2003) 117–177.
[42] M. Núñez, D. de Frutos-Escrig, Testing semantics for probabilistic LOTOS, in: Proc. FORTE'95: IFIP TC6 Eighth International Conference on Formal Description Techniques, Montreal, Canada, October 1995, IFIP Conference Proceedings, vol. 43, Chapman & Hall, 1996, pp. 367–382.
[43] M. Núñez, D. de Frutos-Escrig, L.F.L. Díaz, Acceptance trees for probabilistic processes, in: Proc. CONCUR'95: 6th International Conference on Concurrency, Philadelphia, PA, USA, August, 1995, Lecture Notes in Computer Science, vol. 962, Springer, 1995.

[44] S. Park, F. Pfenning, S. Thrun, A probabilistic language based upon sampling functions, in: Proc. POPL 2005: 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM, 2005, pp. 171–182.
[45] G. Plotkin, A structural approach to operational semantics, Tech. Rep. 19, University of Aahus, 1981 (also published in The Journal of Logic and Algebraic Programming, vols. 60–61, 2004, pp. 17–139).
[46] J.C. Reynolds, Toward a grainless semantics for shared-variable concurrency, in: Proc. FSTTCS 2004, Lecture Notes in Computer Science, vol. 3328, Springer-Verlag, 2004, pp. 35–48.
[47] K. Seidel, Probabilistic communicating processes, Theoret. Comput. Sci. 152 (2) (1995) 219–249.
[48] J. Stoy, Denotational Semantics: The Scott–Strachey Approach to Programming Language, MIT Press, 1977.
[49] H. Zhu, Linking the semantics of a multithreaded discrete event simulation language, Ph.D. thesis, London South Bank University, 2005.
[50] H. Zhu, J.P. Bowen, J. He, From operational semantics to denotational semantics for Verilog, in: Proc. CHARME 2001: 11th Advanced Research Working Conference on Correct Hardware Design and Verification Methods, Lecture Notes in Computer Science, vol. 2144, Springer-Verlag, 2001, pp. 449–464.
[51] H. Zhu, S. Qin, J. He, J.P. Bowen, Integrating probability with time and shared-variable concurrency, in: Proc. SEW-30: 30th NASA/IEEE Software Engineering Workshop, IEEE Computer Society, 2006, pp. 179–189.
[52] H. Zhu, J. He, G. Pu , J. Li, An operational approach to BPEL-like programming, in: Proc. SEW-31: 31st IEEE Software Engineering Workshop, Baltimore, USA, IEEE Computer Society Press, 2007, pp. 236–245.
[53] H. Zhu, S. Qin, J. He, J.P. Bowen, PTSC: Probability, time and shared-variable concurrency, Innov. Syst. Softw. Eng. NASA J. 5 (4) (2009) 271–284.