Math. Struct. in Comp. Science (2009), *vol.* 19, *pp.* 565–599.
doi:10.1017/S0960129509007683 First published online 6 May 2009 Printed in the United Kingdom

On the expressive power of process interruption and compensation

MARIO BRAVETTI[†] and GIANLUIGI ZAVATTARO

Department of Computer Science, Università di Bologna
Email: {bravetti,zavattar}@cs.unibo.it

Received 27 June 2008; revised 16 February 2009

The investigation into the foundational aspects of linguistic mechanisms for programming long-running transactions (such as the *scope* operator of WS-BPEL) has recently renewed the interest in process algebraic operators that, due to the occurrence of a failure, *interrupt* the execution of one process, replacing it with another one called the *failure handler*. We investigate the decidability of termination problems for two simple fragments of CCS (one with recursion and one with replication) extended with one of two such operators, the *interrupt* operator of CSP and the *try-catch* operator for exception handling. More precisely, we consider the existential termination problem (existence of one terminated computation) and the universal termination problem (all computations terminate). We prove that, as far as the decidability of the considered problems is concerned, under replication there is no difference between interrupt and try-catch (universal termination is decidable while existential termination is not), while under recursion this is not the case (existential termination is undecidable while universal termination is decidable only for interrupt). As a consequence of our undecidability results, we show the existence of an expressiveness gap between a fragment of CCS and its extension with either the interrupt or the try-catch operator.

1. Introduction

The investigation into the foundational aspects of so-called service composition languages, see, for example, WS-BPEL (OASIS 2003) and WS-CDL (W3C 2004), has recently attracted the attention of the concurrency theory community. In particular, one of the main novelties of such languages is concerned with primitives for programming *long-running transactions*. These primitives permit, on the one hand, the *interruption* of processes when some unexpected failure occurs and, on the other hand, the activation of alternative processes named *failure handlers* that are responsible for compensating those activities that, even if completed, must be undone due to the occurred failure.

Several recent papers have proposed process calculi that include operators for process failure handling, for example, there have been, just to mention a few, StAC (Butler and Ferreira 2004), cJoin (Bruni *et al.* 2004), cCSP (Butler *et al.* 2003), π_t (Bocchi *et al.* 2003), SAGAS (Bruni *et al.* 2005), web-pi (Laneve and Zavattaro 2005), ORC (Misra and Cook 2007), SCC (Boreale *et al.* 2006), COWS (Lapadula *et al.* 2007), SOCK (Guidi *et al.* 2008)

[†] Research partially funded by EU Integrated Project Sensoria, contract number 016004.

and the Conversation Calculus (Vieira *et al.* 2008). This huge number of calculi that include mechanisms for interruption and failure handling is evidence of the limitations of the usual process calculi, which just include communication primitives, in providing adequate support for a formal investigation of long-running transactions, or of fault and compensation handling in languages for service composition.

In order to perform a formal investigation of these interruption operators, we have decided to concentrate on just two of them that we consider well established because they have been taken either from the tradition of process calculi or from popular programming languages: the *interrupt* operator of CSP (Hoare 1985) and the *try-catch* operator for exception handling in languages such as C++ or Java. The interrupt operator $P \triangle Q$ executes P until Q executes its first action; when Q starts executing, the process P is definitely interrupted. The try-catch operator $\texttt{try}\, P\, \texttt{catch}\, Q$ executes P, but if P performs a *throw* action it is definitely interrupted and Q is executed instead.

We have found these operators particularly useful because, even though very simple, they are expressive enough to model the typical operators for programming long-running transactions. For instance, we can consider an operator $\texttt{scope}_{\texttt{x}}(\texttt{P},\texttt{F},\texttt{C})$ corresponding to a simplified version of the *scope* construct of WS-BPEL. The meaning of this operator is as follows. The main activity P is executed. If a fault is raised by P, its execution is interrupted and the fault handler F is activated. If the main activity P completes, but an outer scope fails and calls for the compensation of the scope x, the compensation handler C is executed.

If we assume that the main activity P communicates internal failure with the action *throw* (we use a notation common in process calculi: an overlined action, for example, $\overline{a}$, is complementary to the corresponding non-overlined one, for example, action a, and complementary actions allow parallel processes to synchronise) and completion with $\overline{end}$, and the request for compensation corresponds with the action $\overline{x}$, we can model the behaviour of $\texttt{scope}_{\texttt{x}}(\texttt{P},\texttt{F},\texttt{C})$ with both the try-catch,

$$\texttt{try}\,\texttt{P}\,\texttt{catch}\,\texttt{F} \;\mid\; end.x.\texttt{C}$$

and the interrupt operator,

$$\texttt{P} \triangle (f.\texttt{F}) \;\mid\; \overline{throw}.\overline{f} \;\mid\; end.x.\texttt{C}$$

where the vertical bar means parallel composition.

These two operators appear to be very similar as they both allow for the combination of two processes P and Q, where the first one executes until the second one performs its first action. However, there is an interesting distinguishing feature: in the *try-catch* operator, the decision to interrupt the execution of P is taken inside P (by means of the execution of the *throw* action), while in the *interrupt* operator this decision is taken from Q (by executing any initial action). For instance, in the above example of modelling the $\texttt{scope}_{\texttt{x}}(\texttt{P},\texttt{F},\texttt{C})$ operator using the interrupt operator, we had to include an additional process $\overline{throw}.\overline{f}$, which captures the request for interruption coming from the main activity and the forwarding of it to the fault handler. Another difference between the try-catch and interrupt operators is that the former includes an implicit scoping mechanism, which has no counterpart in the interrupt operator. More precisely, the try-catch operator defines

https://doi.org/10.1017/S0960129509007683 Published online by Cambridge University Press

	Interrupt	Try-catch
Replication	$CCS_{!}^{\triangle}$ existential termination undecidable universal termination decidable	$CCS_{!}^{\mathtt{tc}}$ existential termination undecidable universal termination decidable
Recursion	$CCS_{rec}^{\triangle}$ existential termination undecidable universal termination decidable	$CCS_{rec}^{\mathtt{tc}}$ existential termination undecidable universal termination undecidable

Table 1. *Summary of the results*

a new scope for the special *throw* action, which is bound to a specific instance of the exception handler.

Starting from these intuitive and informal evaluations of the differences between these operators, we decided to perform a more rigorous and formal investigation. To this end, we have considered two restriction-free fragments of CCS (Milner 1989), one with replication and one with restriction, and we have extended them with either the interrupt or the try-catch operator to produce the four calculi $CCS_{!}^{\triangle}$, $CCS_{!}^{\mathtt{tc}}$, $CCS_{rec}^{\triangle}$ and $CCS_{rec}^{\mathtt{tc}}$, as shown in Table 1. We have decided to consider calculi without restriction, the standard explicit binder operator of CCS, in order that we can observe the impact of the implicit binder of try-catch. Moreover, we decided to consider replication and recursion separately because in CCS there is an interesting interplay between these operators and binders, as proved in Busi *et al.* (2003): in the case of replication, it is possible to compute, given a process P, an upper bound to the nesting depth of binders for all derivatives of P (that is, those processes that can be reached from P after a sequence of transitions). In CCS with recursion, however, this upper bound cannot be computed in general.

For each of the resulting calculi, we have investigated the decidability of the following termination problems: *existential termination* (that is, there exists a terminated computation) and *universal termination* (that is, all computations terminate). The results obtained are shown in Table 1. In order to prove that existential termination is undecidable in the calculi, we reduce the termination problem for Random Access Machines (RAMs) (Shepherdson and Sturgis 1963; Minsky 1967), which is a well-known Turing complete formalism, to the existential termination problem in $CCS_{!}^{\triangle}$ and $CCS_{!}^{\mathtt{tc}}$. As replication is a special case of recursion, we have that the same undecidability result also holds for $CCS_{rec}^{\triangle}$ and $CCS_{rec}^{\mathtt{tc}}$. For universal termination, we proceed as follows. We first prove that it is undecidable in $CCS_{rec}^{\mathtt{tc}}$ by reduction of the RAM termination problem to universal termination in that calculus. Then we separately prove that universal termination is decidable in $CCS_{!}^{\mathtt{tc}}$ and in $CCS_{rec}^{\triangle}$. As recursion is more general than replication, the latter result allows us to conclude that universal termination is decidable in $CCS_{!}^{\triangle}$ also.

The most significant technical contribution of this paper concerns the proof of decidability of universal termination in $CCS_{rec}^{\triangle}$. This is because, while proving decidability of universal termination in $CCS_{!}^{\mathtt{tc}}$ is done by resorting to the approach in Busi *et al.* (2003) based on the existence of an upper bound to the nesting depth of operators in the derivatives of a process, proving termination in $CCS_{rec}^{\triangle}$ requires us to deal with an

https://doi.org/10.1017/S0960129509007683 Published online by Cambridge University Press

unbounded nesting depth of the interrupt operators. For this reason, we need to use a completely different technique, which is based on devising a particular transformation of terms into *trees* (of unbounded depth) and considering an ordering on such trees. The particular transformation devised must be 'tuned' in such a way that the ordering obtained is, on the one hand, a well-quasi-ordering (and to prove this we exploit the Kruskal Tree theorem (Kruskal 1960)), but, on the other hand, strongly compatible with the operational semantics. Obtaining and proving the latter result is particularly intricate, and requires us also to slightly modify the operational semantics of the interrupt operator in a termination-preserving way and, technically, to introduce different kinds of trees on subterms and contexts in order to interpret transitions on trees.

Another interesting consequence of our undecidability results (in particular, the undecidability of existential termination) is the existence of an expressiveness gap between a fragment of CCS (without restriction and relabelling, and with guarded choice) and its extension with either interrupt or try-catch. In fact, we observe that existential termination is decidable for this calculus, but this is not the case for its extensions with either interrupt or try-catch. Thus, there exists no computable encoding of the considered interruption operators into this fragment that preserves at least existential termination.

The paper is structured as follows. In Section 2, we define the calculi under consideration. In Section 3, we prove the undecidability of existential termination in $CCS_{!}^{\triangle}$ and $CCS_{!}^{\mathtt{tc}}$. In Section 4, we prove the undecidability of universal termination in $CCS_{rec}^{\mathtt{tc}}$. In Section 5, we prove the decidability of universal termination for $CCS_{!}^{\mathtt{tc}}$ and $CCS_{rec}^{\triangle}$. In Section 6, we evaluate the impact of our results on the evaluation of the expressive power of the calculi under consideration, and, finally, in Section 7 we give some concluding remarks.

2. The calculi

We start by considering the fragment of CCS (Milner 1989) without recursion, restriction or relabelling (which we will call finite core CCS or just finite CCS). Then we present the two infinite extensions with either replication or recursion, the new *interrupt operator*, and, finally, the *try-catch* operator.

Definition 2.1 (finite core CCS). Let *Name*, ranged over by x, y, …, be a denumerable set of channel names. The class of finite core CCS processes is described by the following grammar:

$$P \quad ::= \quad \mathbf{0} \quad | \quad \alpha.P \quad | \quad P + P \quad | \quad P|P \qquad\qquad \alpha \quad ::= \quad \tau \quad | \quad x \quad | \quad \overline{x}.$$

The term $\mathbf{0}$ denotes the empty process, while the term $\alpha.P$ has the ability to perform the action α (which is either the unobservable τ action or a synchronisation on a channel x) and then behaves like P. Two forms of synchronisation are available: the output $\overline{x}$ and the input x. The sum construct $+$ is used to make a choice among the summands while parallel composition $|$ is used to run parallel programs. We use just α to denote the process $\alpha.\mathbf{0}$.

For input and output actions α, that is, $\alpha \neq \tau$, we write $\overline{\alpha}$ for the complement of α: that is, if $\alpha = x$, then $\overline{\alpha} = \overline{x}$ and if $\alpha = \overline{x}$, then $\overline{\alpha} = x$. We use $n(P)$ to denote the channel names

https://doi.org/10.1017/S0960129509007683 Published online by Cambridge University Press

$$\texttt{PRE}: \quad \alpha.P \xrightarrow{\alpha} P \qquad\qquad \texttt{PAR}: \quad \frac{P \xrightarrow{\alpha} P'}{P|Q \xrightarrow{\alpha} P'|Q}$$

$$\texttt{SUM}: \quad \frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'} \qquad\qquad \texttt{COM}: \quad \frac{P \xrightarrow{\alpha} P' \quad Q \xrightarrow{\overline{\alpha}} Q'}{P|Q \xrightarrow{\tau} P'|Q'}$$

Table 2. *The transition system for finite core CCS (symmetric rules of* PAR *and* SUM *omitted)*

that occur in P. The names in a label α, written $n(\alpha)$, is the set of names in α, that is, the empty set if $\alpha = \tau$ or the singleton $\{x\}$ if α is either x or $\overline{x}$.

Table 2 shows the set of transition rules for finite core CCS.

Definition 2.2 (CCS$_!$). The class of CCS$_!$ processes is defined by adding the production $P ::= \ !\alpha.P$ to the grammar of Definition 2.1.

The transition rule for replication is

$$!\alpha.P \xrightarrow{\alpha} P|!\alpha.P.$$

Definition 2.3 (CCS$_{rec}$). We assume a denumerable set of process variables, ranged over by X. The class of CCS$_{rec}$ processes is defined by adding the productions $P ::= X \mid recX.P$ to the grammar of Definition 2.1. In the process $recX.P$, $recX$ is a binder for the process variable X and P is the scope of the binder. We consider (weakly) guarded recursion, that is, in the process $recX.P$, each occurrence of X (which is free in P) occurs inside a subprocess of the form $\alpha.Q$.

The transition rule for recursion is

$$\frac{P\{recX.P/X\} \xrightarrow{\alpha} P'}{recX.P \xrightarrow{\alpha} P'}$$

where $P\{recX.P/X\}$ denotes the process obtained by substituting $recX.P$ for each free occurrence of X in P, that is, each occurrence of X that is not inside the scope of a binder $recX$. Note that CCS$_!$ is equivalent to a fragment of CCS$_{rec}$. In fact, the replication operator $!\alpha.P$ of CCS$_!$ is equivalent (according, for example, to the standard definition of strong bisimilarity) to the recursive process $recX.(\alpha.(P|X))$.

We now introduce the extensions having the new *process interruption* operator.

Definition 2.4 (CCS$_!^\triangle$ and CCS$_{rec}^\triangle$). The class of CCS$_!^\triangle$ and CCS$_{rec}^\triangle$ processes is defined by adding the production $P ::= P \triangle P$ to the grammars of Definition 2.2 and Definition 2.3, respectively.

https://doi.org/10.1017/S0960129509007683 Published online by Cambridge University Press

The transition rules for the interrupt operator are

$$\frac{P \xrightarrow{\alpha} P'}{P \triangle Q \xrightarrow{\alpha} P' \triangle Q} \qquad \frac{Q \xrightarrow{\alpha} Q'}{P \triangle Q \xrightarrow{\alpha} Q'}$$

We will now complete the list of calculi that we will consider by giving the definitions of the calculi extended by having the new *try-catch* operator.

Definition 2.5 ($\mathrm{CCS}_{!}^{\mathtt{tc}}$ and $\mathrm{CCS}_{rec}^{\mathtt{tc}}$). The class of $\mathrm{CCS}_{!}^{\mathtt{tc}}$ and $\mathrm{CCS}_{rec}^{\mathtt{tc}}$ processes is defined by adding the productions $P ::= \mathtt{try}\, P\, \mathtt{catch}\, P$ and $\alpha ::= \mathtt{throw}$ to the grammars of Definition 2.2 and Definition 2.3, respectively. The new action `throw` is used to model the raising of an exception.

The transition rules for the try-catch operator are

$$\frac{P \xrightarrow{\alpha} P' \quad \alpha \neq \mathtt{throw}}{\mathtt{try}\, P\, \mathtt{catch}\, Q \xrightarrow{\alpha} \mathtt{try}\, P'\, \mathtt{catch}\, Q} \qquad \frac{P \xrightarrow{\mathtt{throw}} P'}{\mathtt{try}\, P\, \mathtt{catch}\, Q \xrightarrow{\tau} Q}$$

We use $\prod_{i \in I} P_i$ to denote the parallel composition of the indexed processes P_i, and $\prod_n P$ to denote the parallel composition of n instances of the process P (if $n = 0$, then $\prod_n P$ denotes the empty process $\mathbf{0}$).

In the following we will only consider closed processes, that is, processes without free occurrences of process variables. Given a closed process Q, its internal runs $Q \longrightarrow Q_1 \longrightarrow Q_2 \longrightarrow \ldots$ coincide with sequences of τ labelled transitions, that is, $P \longrightarrow P'$ if and only if $P \xrightarrow{\tau} P'$. We use $\longrightarrow^+$ to denote the transitive closure of $\longrightarrow$, and $\longrightarrow^*$ to denote the reflexive and transitive closure of $\longrightarrow$.

A process Q is *dead* if there exists no Q' such that $Q \longrightarrow Q'$. We say that a process P *existentially terminates* if there exists P' such that $P \longrightarrow^* P'$ and P' is dead. We say that P *universally terminates* if all its internal runs terminate, that is, the process P cannot give rise to an infinite computation: formally, P *universally terminates* if and only if there exists no family $\{P_i\}_{i \in \mathbf{N}}$ such that $P_0 = P$ and $P_j \longrightarrow P_{j+1}$ for any j. Observe that *universal termination* implies *existential termination*, but the converse does not hold.

3. The undecidability of existential termination in $\mathrm{CCS}_{!}^{\triangle}$ and $\mathrm{CCS}_{!}^{\mathtt{tc}}$

We prove that $\mathrm{CCS}_{!}^{\triangle}$ and $\mathrm{CCS}_{!}^{\mathtt{tc}}$ are powerful enough to model, at least in a non-deterministic way, any Random Access Machine (RAM) (Shepherdson and Sturgis 1963; Minsky 1967), which is a well-known register-based Turing powerful formalism.

A RAM (denoted R in the following) is a computational model composed of a finite set of registers $r_1, \ldots, r_n$ that can hold arbitrary large natural numbers, together with a program composed of indexed instructions $(1 : I_1), \ldots, (m : I_m)$, that is a sequence of simple numbered instructions, such as arithmetical operations (on the contents of registers) or conditional jumps. An internal state of a RAM is given by $(i, c_1, \ldots, c_n)$ where i is the program counter indicating the next instruction to be executed, and $c_1, \ldots, c_n$

https://doi.org/10.1017/S0960129509007683 Published online by Cambridge University Press

are the current contents of the registers $r_1, \ldots, r_n$, respectively. Given a configuration $(i, c_1, \ldots, c_n)$, its computation proceeds by executing the instructions in sequence, unless a jump instruction is encountered. The execution stops when an instruction number higher than the length of the program is reached. Note that the computation of the RAM proceeds deterministically (it does not exhibit non-deterministic behaviours).

Without loss of generality, we assume that the registers contain the value 0 at both the beginning and end of the computation. In other words, the initial configuration is $(1, 0, \ldots, 0)$ and, if the RAM terminates, the final configuration is $(i, 0, \ldots, 0)$ with $i > m$ (that is, the instruction I_i is undefined). More formally, we use $(i, c_1, \ldots, c_n) \rightarrow_R (i', c'_1, \ldots, c'_n)$ to indicate the fact that the configuration of the RAM R changes from $(i, c_1, \ldots, c_n)$ to $(i', c'_1, \ldots, c'_n)$ after the execution of the ith instruction ($\rightarrow_R^*$ is the reflexive and transitive closure of $\rightarrow_R$).

The RAM instructions have two possible formats:

— $(i : Succ(r_j))$: add 1 to the contents of register r_j;

— $(i : DecJump(r_j, s))$: if the contents of register r_j is not zero, then decrease it by 1 and go to the next instruction, otherwise jump to instruction s.

Our encoding is non-deterministic because it introduces computations that do not follow the expected behaviour of the RAM being modelled. However, all these computations are infinite. This ensures that, given a RAM, its model has a terminating computation if and only if the RAM terminates. This proves that *existential termination* is undecidable.

In both this section and the next, which is devoted to the proof of the undecidability results, we reason up to a structural congruence $\equiv$ in order to rearrange the order of parallel composed processes and to abstract away from the terminated processes $\mathbf{0}$. We define $\equiv$ as the least congruence relation satisfying the usual axioms: $P|Q \equiv Q|P$, $P|(Q|R) \equiv (P|Q)|R$ and $P|\mathbf{0} \equiv P$.

Let R be a RAM with registers $r_1, \ldots, r_n$, and instructions $(1 : I_1), \ldots, (m : I_m)$. We model registers and instructions separately.

The program counter is modelled by a message $\overline{p_i}$ indicating that the ith instruction is the next to be executed. For each $1 \leqslant i \leqslant m$, we model the ith instruction $(i : I_i)$ of R by a process that is guarded by an input operation p_i. Once activated, the instruction performs its operation on the registers and then updates the program counter by producing $\overline{p_{i+1}}$ (or $\overline{p_s}$ in the case of a jump).

Formally, for any $1 \leqslant i \leqslant m$, the instruction $(i : I_i)$ is modelled by $[\![(i : I_i)]\!]$, which is a shorthand notation for the following processes:

$$[\![(i : I_i)]\!] \quad : \quad !p_i.(\overline{inc_j}.\overline{loop} \mid \overline{p_{i+1}}) \qquad \text{if } I_i = Succ(r_j)$$

$$[\![(i : I_i)]\!] \quad : \quad !p_i.\big(\ \tau.(\overline{loop} \mid \overline{dec_j}.loop.loop.\overline{p_{i+1}}) + \tau.\overline{zero_j}.ack.\overline{p_s}\ \big) \qquad \text{if } I_i = DecJump(r_j, s).$$

It is worth noting that every time an increment operation is performed, a process $\overline{loop}$ is spawned. This process will be removed by a corresponding decrement operation. The modelling of the $DecJump(r_j, s)$ instruction decides internally whether to decrement or to test the register for zero.

https://doi.org/10.1017/S0960129509007683 Published online by Cambridge University Press

In the case of decrement, if the register is empty, the instruction deadlocks because the register cannot actually be decremented. Nevertheless, before trying to decrement the register, a process $\overline{loop}$ is generated. As we will discuss in the following, the presence of this process prevents the encoding from terminating. If the decrement operation is actually executed, two instances of the process $\overline{loop}$ are removed, one instance corresponding to the one produced before the execution of the decrement, and one instance corresponding to a previous increment operation.

In the case of testing for zero, the corresponding register will have to be modified as we will discuss below. As this modification requires the execution of several actions, the instruction waits for an acknowledgment before producing the new program counter $\overline{p_s}$.

We now show how to model the registers using either the interrupt or try-catch operators. In both cases we exploit the following idea. Every time the register r_j is incremented, a dec_j process is spawned, which permits the subsequent execution of a corresponding decrement operation. In the case of testing the register r_j for zero, we will exploit either the interrupt or the try-catch operators in order to remove all the active processes dec_j, thus resetting the register. If the register is not empty when it is reset, the computation of the encoding no longer reproduces the RAM computation. However, this 'wrong' computation certainly does not terminate, thus we can conclude that we faithfully model at least the terminating computations. Divergence in the case of a 'wrong' reset is guaranteed by the fact that if the register is not empty, k instances of dec_j processes are removed with $k > 0$, and k instances of the process $\overline{loop}$ (previously produced by the corresponding k increment operations) will never be removed. As discussed above, the presence of $\overline{loop}$ processes prevents the encoding from terminating. This is guaranteed by considering, for example, the divergent process

$$LOOP \;:\; loop.(\bar{l} \mid !l.\bar{l}).$$

Formally, we model each register r_j, when it contains c_j, using one of the following processes, which we denote $[\![r_j = c_j]\!]^{\triangle}$ and $[\![r_j = c_j]\!]^{\mathtt{tc}}$:

$$[\![r_j = c_j]\!]^{\triangle} \;:\; \left(!inc_j.dec_j \mid \prod_{c_j} dec_j\right) \triangle \left(zero_j.\overline{nr_j}.\overline{ack}\right)$$

$$[\![r_j = c_j]\!]^{\mathtt{tc}} \;:\; \mathtt{try}\left(!inc_j.dec_j \mid \prod_{c_j} dec_j \mid zero_j.\mathtt{throw}\right)\ \mathtt{catch}\left(\overline{nr_j}.\overline{ack}\right).$$

It is worth observing that, when a test for zero is performed on the register r_j, an output operation $\overline{nr_j}$ is executed before sending the acknowledgment to the corresponding instruction. This action is used to activate a new instance of the process $[\![r_j = 0]\!]$, as the process modelling the register r_j is removed by the execution of either the interrupt or the try-catch operators. The activation of new instances of the process modelling the registers is obtained simply by considering, for each register r_j, (one of) the following two processes:

$$!nr_j.[\![r_j = 0]\!]^{\triangle} \qquad !nr_j.[\![r_j = 0]\!]^{\mathtt{tc}}.$$

We are now able to define formally our encoding of RAMs, as well as its properties.

Definition 3.1. Let R be a RAM with program instructions $(1 : I_1), \ldots, (m : I_m)$ and registers $r_1, \ldots, r_n$, and let Γ be either $\triangle$ or $\mathtt{tc}$. Given the configuration $(i, c_1, \ldots, c_n)$ of R,

https://doi.org/10.1017/S0960129509007683 Published online by Cambridge University Press

we define

$$[\![(i, c_1, \ldots, c_n)]\!]_R^\Gamma \;=\; \overline{p_i} \mid [\![(1 : I_1)]\!] \mid \ldots \mid [\![(m : I_m)]\!] \mid \prod_{\sum_{j=1}^{n} c_j} \overline{loop} \mid LOOP \mid [\![r_1 = c_1]\!]^\Gamma \mid \ldots \mid [\![r_n = c_n]\!]^\Gamma \mid !nr_1.[\![r_1 = 0]\!]^\Gamma \mid \ldots \mid !nr_n.[\![r_n = 0]\!]^\Gamma$$

to be the encoding of the RAM R in either $CCS_!^\triangle$ or $CCS_!^{\mathtt{tc}}$, where $\Gamma = \triangle$ or $\Gamma = \mathtt{tc}$, respectively. The processes $[\![(i : I_i)]\!]$, $LOOP$, and $[\![r_j = c_j]\!]^\Gamma$ are as defined above.

The following proposition states that every step of the computation of a RAM can be mimicked by the corresponding encoding. On the other hand, the encoding could introduce additional computations. The proposition also states that all these additional computations are infinite.

Proposition 3.2. Let R be a RAM with program instructions $(1 : I_1), \ldots, (m : I_m)$ and registers $r_1, \ldots, r_n$, and let Γ be either $\triangle$ or $\mathtt{tc}$. Given a configuration $(i, c_1, \ldots, c_n)$ of R, we have that, if $i > m$ and $c_j = 0$ for each j, with $1 \leqslant j \leqslant n$, then $[\![(i, c_1, \ldots, c_n)]\!]_R^\Gamma$ is a dead process, otherwise:

1 If $(i, c_1, \ldots, c_n) \rightarrow_R (i', c_1', \ldots, c_n')$, we have $[\![(i, c_1, \ldots, c_n)]\!]_R^\Gamma \rightarrow^+ [\![(i', c_1', \ldots, c_n')]\!]_R^\Gamma$.
2 If $[\![(i, c_1, \ldots, c_n)]\!]_R^\Gamma \longrightarrow Q_1 \longrightarrow Q_2 \longrightarrow \cdots \longrightarrow Q_l$ is a, possibly zero-length, internal run of $[\![(i, c_1, \ldots, c_n)]\!]_R^\Gamma$, then one of the following holds:
   — There exists k, with $1 \leqslant k \leqslant l$, such that we have $Q_k \equiv [\![(i', c_1', \ldots, c_n')]\!]_R^\Gamma$, with $(i, c_1, \ldots, c_n) \rightarrow_R (i', c_1', \ldots, c_n')$.
   — $Q_l \longrightarrow^+ [\![(i', c_1', \ldots, c_n')]\!]_R^\Gamma$, with $(i, c_1, \ldots, c_n) \rightarrow_R (i', c_1', \ldots, c_n')$.
   — Q_l does not existentially terminate.

Proof. First, if $i > m$ and $c_i = 0$, $\forall 1 \leqslant i \leqslant n$, then $[\![(i, c_1, \ldots, c_n)]\!]_R^\Gamma$ is obviously a dead process because no $\overline{loop}$ process is included among the parallel processes composing $[\![(i, c_1, \ldots, c_n)]\!]_R^\Gamma$ and all other processes are stuck on inputs that cannot be triggered.

If, instead, $i > m$ and there exists i, with $1 \leqslant i \leqslant n$, such that $c_i > 0$, then $[\![(i, c_1, \ldots, c_n)]\!]_R^\Gamma$ can only perform a unique reduction: the one originated by the synchronisation with the divergent process $LOOP$. Thus both statements 1 and 2 are trivially satisfied.

Otherwise, $i \leqslant m$, and we suppose $(i, c_1, \ldots, c_n) \rightarrow_R (i', c_1', \ldots, c_n')$. We have two cases:

— If I_i is a $Succ(r_j)$ instruction, the process $[\![(i, c_1, \ldots, c_n)]\!]_R^\Gamma$ may perform a reduction sequence composed of two reduction steps that lead to $[\![(i', c_1', \ldots, c_n')]\!]_R^\Gamma$: the first reduction is caused by the synchronisation on p_i and the second by the synchronisation on inc_j. Every process occurring in such a sequence has at most one alternative reduction: the synchronisation with the divergent process $LOOP$. Thus both statements 1 and 2 are satisfied.
— If I_i is a $DecJump(r_j, s)$ instruction, we have two subcases depending on whether $c_j = 0$ or $c_j > 0$.
  – If $c_j = 0$, then $[\![(i, c_1, \ldots, c_n)]\!]_R^\Gamma$ may perform a reduction sequence composed of six or seven reduction steps (six in the case of $\Gamma = \mathtt{tc}$ and seven in the case of $\Gamma = \triangle$) that leads to $[\![(i', c_1', \ldots, c_n')]\!]_R^\Gamma$: the first reduction is caused by the synchronisation on p_i; the second by the internal τ action in the right-hand side of the choice in

https://doi.org/10.1017/S0960129509007683 Published online by Cambridge University Press

the encoding of $DecJump(r_j, s)$; the third by the synchronisation on $zero_j$; then, if $\Gamma = \mathtt{tc}$, the fourth by the execution of *throw* inside the encoding of the r_j register, or if $\Gamma = \triangle$, the fourth and fifth by the synchronisation on z_j and $stop_j$ inside the encoding of the r_j register; and, finally, the last two reductions are caused by the synchronisations on nr_j and *ack*. Apart from the process reached by the first reduction step, every process occurring in such a sequence has at most one alternative reduction: the synchronisation with the divergent process $LOOP$. However, for the process reached by the first reduction step, there is, possibly, in addition to the above alternative reduction, another alternative reduction: the reduction caused by the internal τ action in the left-hand side of the choice in the encoding of $DecJump(r_j, s)$. Such a reduction leads to a process where an additional instance of the process $\overline{loop}$ is produced, and, since synchronisation on dec_j cannot occur (because $c_j = 0$), such a process can perform a unique reduction: the reduction originated by the synchronisation with the divergent process $LOOP$. Thus both statements 1 and 2 are satisfied.

– If $c_j > 0$, then $[\![(i, c_1, \ldots, c_n)]\!]_R^\Gamma$ may perform a reduction sequence composed of five reduction steps that leads to $[\![(i', c'_1, \ldots, c'_n)]\!]_R^\Gamma$: the first reduction is caused by the synchronisation on p_i; the second by the internal τ action in the left-hand side of the choice in the encoding of $DecJump(r_j, s)$; the third by the synchronisation on dec_j; and the fourth and fifth by the synchronisation on *loop* (two times). Apart from the process reached by the first reduction step, every process occurring in such a sequence has at most one alternative reduction: the synchronisation with the divergent process $LOOP$. However, for the process reached by the first reduction step, there is, possibly, in addition to the above alternative reduction, another alternative reduction: the reduction caused by the internal τ action in the right-hand side of the choice in the encoding of $DecJump(r_j, s)$. Such a reduction leads to a process that may perform a reduction sequence composed of two or three reduction steps (two in the case of $\Gamma = \mathtt{tc}$ and three in the case of $\Gamma = \triangle$) that leads to a process with the following structure of parallel components: the number of $\overline{loop}$ processes is strictly greater than the sum of the numbers of dec_k processes for all $1 \leqslant k \leqslant n$. Such a sequence is composed of an initial reduction caused by the synchronisation on $zero_j$ and then, in the case of $\Gamma = \mathtt{tc}$, a second reduction caused by the execution of *throw* inside the encoding of the r_j register, or, in the case of $\Gamma = \triangle$, a second and third reduction caused by the synchronisation on z_j and $stop_j$ inside the encoding of the r_j register. Note that in both cases, an interruption (or exception) is generated that, since $c_j > 0$, removes at least one occurrence of a dec_j process (without removing a corresponding number of occurrences of $\overline{loop}$ processes). It is easy to see that the process reached by such a sequence does not existentially terminate: the possibility of reaching a dead process depends crucially on the capability of removing all $\overline{loop}$ processes (the only thing that can prevent the $LOOP$ divergent process from being activated by synchronisation) and the removal of one $\overline{loop}$ process can only be done by the encoding after one dec_j process is correspondingly removed. Finally, every process

https://doi.org/10.1017/S0960129509007683 Published online by Cambridge University Press

occurring in the last reduction sequence has at most an alternative reduction: the synchronisation with the divergent process $LOOP$. Thus both statements 1 and 2 are satisfied. □

Thus, we have the following corollary.

Corollary 3.3. Let R be a RAM. The RAM R terminates if and only if $[\![(1,0,\ldots,0)]\!]_R^{\Gamma}$ existentially terminates (for both $\Gamma = \triangle$ and $\Gamma = \mathtt{tc}$).

Proof. If the RAM R terminates, it follows immediately, by induction on the length of the computation of the RAM, from Proposition 3.2 (first part plus statement 1) that $[\![(1,0,\ldots,0)]\!]_R^{\Gamma}$ can reach a dead state.

For the opposite implication, we show that if the RAM R does not terminate, then $[\![(1,0,\ldots,0)]\!]_R^{\Gamma}$ does not existentially terminate. If this were not the case, and we could reach a dead process from $[\![(1,0,\ldots,0)]\!]_R^{\Gamma}$, we can show that this would violate Proposition 3.2 (statement 2) by induction on the length of an assumed reduction sequence from the encoding of a configuration reached by the non-terminating RAM to the dead process. □

This proves that existential termination is undecidable in both $CCS_{!}^{\triangle}$ and $CCS_{!}^{\mathtt{tc}}$. As replication is a particular case of recursion, the same undecidability result also holds for $CCS_{rec}^{\triangle}$ and $CCS_{rec}^{\mathtt{tc}}$.

4. Undecidability of universal termination in $\mathrm{CCS}_{rec}^{\mathtt{tc}}$

In this section we prove that universal termination is also undecidable in $\mathrm{CCS}_{rec}^{\mathtt{tc}}$. This result follows from the existence of a deterministic encoding of RAMs satisfying the following stronger soundness property: a RAM terminates if and only if the corresponding encoding universally terminates.

The basic idea of the new modelling is to represent the number c_j, stored in the register r_j, using a process composed of c_j nested try-catch operators. This approach can be adopted in $\mathrm{CCS}_{rec}^{\mathtt{tc}}$ because standard recursion admits recursion in *depth* – it was not applicable in $\mathrm{CCS}_{!}^{\mathtt{tc}}$ because replication only supports recursion in *width*. By recursion in width, we mean that the recursively defined term can expand only in parallel as, for instance, in $recX.(P|X)$ (corresponding to the replicated process $!P$) where the variable X is an operand of the parallel composition operator. By recursion in depth, we mean that the recursively defined term also expands under other operators such as, for instance, in $recX.(\mathtt{try}\,(P|X)\,\mathtt{catch}\,Q)$ (corresponding to an unbounded nesting of try-catch operators).

Let R be a RAM with registers $r_1,\ldots,r_n$, and instructions $(1 : I_1),\ldots,(m : I_m)$. We will start by presenting the modelling of the instructions, which is similar to the encoding presented in the previous section. Note that here we do not need to assume that all the registers have value 0 in a terminating configuration. We encode each instruction $(i : I_i)$ using the process $[\![(i : I_i)]\!]$, which is a shorthand for the following process:

$$\begin{array}{lcll} [\![(i : I_i)]\!] & : & recX.p_i.(\overline{inc}_j.\overline{p_{i+1}}|X) & \text{if } I_i = Succ(r_j) \\ [\![(i : I_i)]\!] & : & recX.p_i.\big((\overline{zero}_j.\overline{p_s} \;+\; \overline{dec}_j.ack.\overline{p_{i+1}})|X\big) & \text{if } I_i = DecJump(r_j, s). \end{array}$$

https://doi.org/10.1017/S0960129509007683 Published online by Cambridge University Press

As in the previous section, the program counter is modelled by the process $\overline{p_i}$, which indicates that the next instruction to execute is $(i : I_i)$. The process $[\![(i : I_i)]\!]$ simply consumes the program counter process, then updates the registers (respectively, performs a test for zero), and, finally, produces the new program counter process $\overline{p_{i+1}}$ (respectively, $\overline{p_s}$). Notice that in the case of a decrement operation, the instruction process waits for an acknowledgment before producing the new program counter process. This is necessary because the register decrement requires the execution of several operations.

The register r_j, which we assume to be initially empty, is modelled by the process $[\![r_j = 0]\!]$, which is a shorthand for the process

$$[\![r_j = 0]\!] \quad : \quad recX.\big(zero_j.X \; + \; inc_j.\mathtt{try}\, R_j \,\mathtt{catch}\,(\overline{ack}.X)\big)$$

where, to simplify the notation, we have used the shorthand R_j defined by

$$R_j \quad : \quad recY.\big(dec_j.\mathtt{throw} \; + \; inc_j.\mathtt{try}\, Y \,\mathtt{catch}\,(\overline{ack}.Y)\big).$$

The process $[\![r_j = 0]\!]$ is able to react either to test for zero requests or increment operations. In the case of increment requests, a try-catch operator is activated. A recursive process is installed within this operator that reacts to either increment or decrement requests. In the case of an increment, an additional try-catch operator is activated (thus increasing the number of nested try-catch operators). In the case of a decrement, a failure is raised, which removes the active try-catch operator (thus decreasing the number of nested try-catch operators) and emits the acknowledgment required by the instruction process. When the register returns to being empty, the outer recursion reactivates the initial behaviour.

Formally, the register r_j with contents $c_j > 0$ is modelled by the following process composed of the nesting of c_j try-catch operators:

$$\begin{array}{rcl}
[\![r_j = c_j]\!] & : & \mathtt{try} \\
& & \quad \big(\mathtt{try} \\
& & \qquad \big(\cdots \\
& & \qquad \mathtt{try}\, R_j \,\mathtt{catch}\,(\overline{ack}.R_j) \\
& & \qquad \cdots\big) \\
& & \quad \mathtt{catch}\,(\overline{ack}.R_j)\big) \\
& & \mathtt{catch}\,(\overline{ack}.[\![r_j = 0]\!])
\end{array}$$

where R_j is as defined above. We are now able to define formally the encoding of RAMs in $CCS^{\mathtt{tc}}_{rec}$.

Definition 4.1. Let R be a RAM with program instructions $(1 : I_1), \ldots, (m : I_m)$ and registers $r_1, \ldots, r_n$. Given the configuration $(i, c_1, \ldots, c_n)$, we define the encoding of the RAM R in $CCS^{\mathtt{tc}}_{rec}$ to be

$$[\![(i, c_1, \ldots, c_n)]\!]_R \;=\; \overline{p_i} \mid [\![(1 : I_1)]\!] \mid \ldots \mid [\![(m : I_m)]\!] \mid [\![r_1 = c_1]\!] \mid \ldots \mid [\![r_n = c_n]\!].$$

The new encoding faithfully reproduces the behaviour of a RAM, as stated by the following proposition, in which we use the notion of a *deterministic* internal run, which is

https://doi.org/10.1017/S0960129509007683 Published online by Cambridge University Press

defined as follows: an internal run $P_0 \longrightarrow P_1 \longrightarrow \ldots \longrightarrow P_l$ is deterministic if for every process P_i, with $i < l$, P_{i+1} is the unique process Q such that $P_i \longrightarrow Q$.

Proposition 4.2. Let R be a RAM with program instructions $(1 : I_1), \ldots, (m : I_m)$ and registers $r_1, \ldots, r_n$. Given a configuration $(i, c_1, \ldots, c_n)$ of R, we have that, if $i > m$, then $[\![(i, c_1, \ldots, c_n)]\!]_R$ is a dead process, otherwise:

1 If $(i, c_1, \ldots, c_n) \rightarrow_R (i', c'_1, \ldots, c'_n)$, we have $[\![(i, c_1, \ldots, c_n)]\!]_R \rightarrow^+ [\![(i', c'_1, \ldots, c'_n)]\!]_R$.
2 There exists a non-zero length deterministic internal run

$$[\![(i, c_1, \ldots, c_n)]\!]_R^\Gamma \longrightarrow Q_1 \longrightarrow Q_2 \longrightarrow \cdots \longrightarrow [\![(i', c'_1, \ldots, c'_n)]\!]_R^\Gamma$$

such that $(i, c_1, \ldots, c_n) \rightarrow_R (i', c'_1, \ldots, c'_n)$.

Proof. First, if $i > m$, then $[\![(i, c_1, \ldots, c_n)]\!]_R$ is obviously a dead process because all processes (of which it is comprised through parallel composition) are stuck on inputs that cannot be triggered.

Otherwise, if $i \leqslant m$, suppose $(i, c_1, \ldots, c_n) \rightarrow_R (i', c'_1, \ldots, c'_n)$. There are two cases:

— If I_i is a $Succ(r_j)$ instruction, the process $[\![(i, c_1, \ldots, c_n)]\!]_R$ proceeds deterministically by performing a reduction sequence composed of two reduction steps that lead to $[\![(i', c'_1, \ldots, c'_n)]\!]_R$: the first reduction is caused by the synchronisation on p_i and the second by the synchronisation on inc_j. Thus both statements 1 and 2 are satisfied.
— If I_i is a $DecJump(r_j, s)$ instruction, there are two subcases depending on whether $c_j = 0$ or $c_j > 0$.
  – If $c_j = 0$, then $[\![(i, c_1, \ldots, c_n)]\!]_R$ proceeds deterministically by performing a reduction sequence composed of two reduction steps that lead to $[\![(i', c'_1, \ldots, c'_n)]\!]_R$: the first reduction is caused by the synchronisation on p_i and the second by the synchronisation on $zero_j$. Thus both statements 1 and 2 are satisfied.
  – If $c_j > 0$, then $[\![(i, c_1, \ldots, c_n)]\!]_R$ proceeds deterministically by performing a reduction sequence composed of four reduction steps that lead to $[\![(i', c'_1, \ldots, c'_n)]\!]_R$: the first reduction is caused by the synchronisation on p_i; the second by the synchronisation on dec_j; the third by the execution of *throw* inside the innermost *try-catch* clause in the encoding of the r_j register; and the fourth by the synchronisation on *ack*. Thus both statements 1 and 2 are satisfied. □

Thus, we have the following corollary.

Corollary 4.3. Let R be a RAM. The RAM R terminates if and only if $[\![(1, 0, \ldots, 0)]\!]_R$ universally terminates.

Proof. The proof of this corollary is the same as that of Corollary 4.3, but using Proposition 4.2 instead of Proposition 3.2. □

This proves that universal termination is undecidable in $CCS_{rec}^{\mathtt{tc}}$.

https://doi.org/10.1017/S0960129509007683 Published online by Cambridge University Press

5. Decidability of universal termination in $CCS_!^{tc}$ and $CCS_{rec}^{\triangle}$

In the RAM encoding presented in the previous section, natural numbers are represented by chains of nested try-catch operators, which are constructed by exploiting recursion. In this section we prove that both recursion and try-catch are strictly necessary. In fact, if we consider replication instead of recursion, or the interrupt operator instead of the try-catch operator, universal termination turns out to be decidable.

These results are based on the theory of well-structured transition systems (Finkel and Schnoebelen 2001). We start by recalling some basic definitions and results for well-structured transition systems that will be used in the following.

A *quasi-ordering*, also known as a pre-order, is a reflexive and transitive relation.

Definition 5.1. A *well-quasi-ordering* (wqo) is a quasi-ordering $\leqslant$ over a set $\mathscr{S}$ such that for any infinite sequence $s_0, s_1, s_2, \ldots$ in $\mathscr{S}$, there exist indices $i < j$ such that $s_i \leqslant s_j$.

Transition systems can be formally defined as follows.

Definition 5.2. A *transition system* is a structure $TS = (\mathscr{S}, \rightarrow)$, where $\mathscr{S}$ is a set of *states* and $\rightarrow \subseteq \mathscr{S} \times \mathscr{S}$ is a set of *transitions*. We write $Succ(s)$ to denote the set $\{s' \in \mathscr{S} \mid s \rightarrow s'\}$ of immediate successors of S. TS is *finitely branching* if all $Succ(s)$ are finite.

Well-structured transition systems, defined as follows, provide the key tool for deciding the properties of computations.

Definition 5.3. A *well-structured transition system with strong compatibility* is a transition system $TS = (\mathscr{S}, \rightarrow)$ equipped with a quasi-ordering $\leqslant$ on $\mathscr{S}$ such that the following two conditions hold:

1 **Well-quasi-ordering**: $\leqslant$ is a well-quasi-ordering.
2 **Strong compatibility**: $\leqslant$ is (upward) compatible with $\rightarrow$, that is, for all $s_1 \leqslant t_1$ and all transitions $s_1 \rightarrow s_2$, there exists a state t_2 such that $t_1 \rightarrow t_2$ and $s_2 \leqslant t_2$.

In the following we use the notation $(\mathscr{S}, \rightarrow, \leqslant)$ for transition systems equipped with a quasi-ordering $\leqslant$.

The following theorem (which is a special case of a result in Finkel and Schnoebelen (2001)) will be used to obtain our decidability results.

Theorem 5.4. Let $(\mathscr{S}, \rightarrow, \leqslant)$ be a finitely branching, well-structured transition system with strong compatibility, decidable $\leqslant$ and computable $Succ$. The existence of an infinite computation starting from a state $s \in \mathscr{S}$ is decidable.

Rather than prove the decidability of universal termination in $CCS_{rec}^{\triangle}$ using the original transition system, we will use one that is termination equivalent to it instead. The new transition system does not eliminate interrupt operators during the computation, so the nesting of interrupt operators can only grow, and never shrinks. As we will see, this transformation will be needed for proving that the ordering that we consider on processes is strongly compatible with the operational semantics. Formally, we define the

https://doi.org/10.1017/S0960129509007683 Published online by Cambridge University Press

new transition system $\overset{\alpha}{\longmapsto}$ for $CCS^{\triangle}_{rec}$ by considering the transition rules of Definition 2.3 (where $\overset{\alpha}{\longmapsto}$ is substituted for $\overset{\alpha}{\longrightarrow}$) plus the following rules

$$\frac{P \overset{\alpha}{\longmapsto} P'}{P \triangle Q \overset{\alpha}{\longmapsto} P' \triangle Q} \qquad \frac{Q \overset{\alpha}{\longmapsto} Q'}{P \triangle Q \overset{\alpha}{\longmapsto} Q' \triangle \mathbf{0}}$$

Notice that the first of the above rules is as in Definition 2.4, while the second is different because it does not remove the $\triangle$ operator.

As we did for the standard transition system, we assume that the reductions $\longmapsto$ of the new semantics corresponds to the τ-labelled transitions $\overset{\tau}{\longmapsto}$. For the new semantics also, we say that a process P universally terminates if and only if all its computations are finite, that is, it cannot give rise to an infinite sequence of reductions $\longmapsto$.

To prove the equivalence of the semantics of $CCS^{\triangle}_{rec}$ presented in Section 2 with the alternative semantics presented in this section with respect to termination, we need to define the following congruence between processes:

Definition 5.5. We define $\equiv_T$ as the least congruence relation satisfying the axiom

$$P \triangle \mathbf{0} \equiv_T P.$$

The equivalence result (equivalence with respect to termination) can be easily proved with the help of the following propositions.

Proposition 5.6. Let $P, Q \in CCS^{\triangle}_{rec}$ with $P \equiv_T Q$. If $P \overset{\alpha}{\longrightarrow} P'$, then there exists Q' such that $Q \overset{\alpha}{\longrightarrow} Q'$ and $P' \equiv_T Q'$.

Proof. The proof is by induction on the proof of the relation $P \equiv_T Q$. □

Proposition 5.7. Let $P, Q \in CCS^{\triangle}_{rec}$ with $P \equiv_T Q$. If $P \overset{\alpha}{\longmapsto} P'$, then there exists Q' such that $Q \overset{\alpha}{\longmapsto} Q'$ and $P' \equiv_T Q'$.

Proof. The proof is by induction on the proof of the relation $P \equiv_T Q$. □

Proposition 5.8. Let $P \in CCS^{\triangle}_{rec}$. If $P \overset{\alpha}{\longmapsto} P'$, then there exists P'' such that $P \overset{\alpha}{\longrightarrow} P''$ and $P' \equiv_T P''$.

Proof. The proof is by induction on the proof of the derivation $P \overset{\alpha}{\longmapsto} P'$. □

Proposition 5.9. Let $P \in CCS^{\triangle}_{rec}$. If $P \overset{\alpha}{\longrightarrow} P'$, then there exists P'' such that $P \overset{\alpha}{\longmapsto} P''$ and $P' \equiv_T P''$.

Proof. The proof is by induction on the proof of the derivation $P \overset{\alpha}{\longrightarrow} P'$. □

Corollary 5.10. Let $P \in CCS^{\triangle}_{rec}$. Then P universally terminates according to the semantics $\longrightarrow$ if and only if P universally terminates according to the new semantics $\longmapsto$.

We now prove the decidability of universal termination in $CCS^{\mathtt{tc}}_{!}$ and in $CCS^{\triangle}_{rec}$ separately in the following two subsections.

https://doi.org/10.1017/S0960129509007683 Published online by Cambridge University Press

5.1. *Universal termination is decidable in* $(CCS^{\mathtt{tc}}_{!}, \longrightarrow)$

The proof for $CCS^{\mathtt{tc}}_{!}$ is just a reformulation of the proof given in Busi *et al.* (2009) for decidability of universal termination in CCS without relabelling and with replication instead of recursion.

We first define a quasi-ordering on processes for $(CCS^{\mathtt{tc}}_{!}, \longrightarrow)$, which turns out to be a well-quasi-ordering compatible with $\longrightarrow$. Then, exploiting Theorem 5.4, we can show that universal termination is decidable.

Definition 5.11. Let $P \in CCS^{\mathtt{tc}}_{!}$. We use $Deriv(P)$ to denote the set of processes reachable from P through a sequence of reduction steps:

$$Deriv(P) = \{Q \mid P \longrightarrow^{*} Q\}.$$

To define the wqo on processes, we need the following structural congruence.

Definition 5.12. We define $\equiv$ as the least congruence relation satisfying the axioms

$$P|Q \equiv Q|P \qquad P|(Q|R) \equiv (P|Q)|R \qquad P|\mathbf{0} \equiv P.$$

We are now ready to define the quasi-ordering on processes.

Definition 5.13. Let $P, Q \in CCS^{\mathtt{tc}}_{!}$. We write $P \preceq Q$ if and only if there exist n, P', R, $P_1, \ldots, P_n$, $Q_1, \ldots, Q_n$, $S_1, \ldots, S_n$ such that $P \equiv P' | \prod_{i=1}^{n} \mathtt{try}\, P_i\, \mathtt{catch}\, S_i$ and $Q \equiv P'|R| \prod_{i=1}^{n} \mathtt{try}\, Q_i\, \mathtt{catch}\, S_i$, and $P_i \preceq Q_i$ for $i = 1, \ldots, n$.

The above definition can be seen as a definition by induction on the nesting depth of try-catch operators. In the base case, we have $n = 0$, thus $P \preceq Q$ if and only if Q contains the processes in P plus other processes in parallel. In the inductive case, $P \preceq Q$ if each process $\mathtt{try}\, P_i\, \mathtt{catch}\, S_i$ occurring in P has a corresponding process $\mathtt{try}\, Q_i\, \mathtt{catch}\, S_i$ with $P_i \preceq Q_i$.

Theorem 5.14. Let $P \in CCS^{\mathtt{tc}}_{!}$. Then the transition system $(Deriv(P), \longrightarrow, \preceq)$ is a finitely branching, well-structured transition system with strong compatibility, decidable $\preceq$ and computable *Succ*.

Proof. In the following we show precisely how to obtain the proof of this result from the analogous proof in Busi *et al.* (2009) for the decidability of universal termination in CCS without relabelling and with replication instead of recursion.

First we need to introduce some auxiliary definitions. Let $P \in CCS^{\mathtt{tc}}_{!}$. Using $d_{\mathtt{tc}}(P)$ to denote the maximum number of nested try-catch operators in process P, we have

$$\begin{aligned}
d_{\mathtt{tc}}(\mathbf{0}) &= 0 \\
d_{\mathtt{tc}}(\alpha.P) &= d_{\mathtt{tc}}(!\alpha.P) \\
&= d_{\mathtt{tc}}(P) \\
d_{\mathtt{tc}}(P + Q) &= d_{\mathtt{tc}}(P|Q) \\
&= max(\{d_{\mathtt{tc}}(P), d_{\mathtt{tc}}(Q)\}) \\
d_{\mathtt{tc}}(\mathtt{try}\, P\, \mathtt{catch}\, Q) &= max(\{1 + d_{\mathtt{tc}}(P), d_{\mathtt{tc}}(Q)\}).
\end{aligned}$$

https://doi.org/10.1017/S0960129509007683 Published online by Cambridge University Press

The set of sequential subprocesses of P is defined by

$$\begin{aligned} Sub(\mathbf{0}) &= \{\mathbf{0}\} \\ Sub(\alpha.P) &= \{\alpha.P\} \cup Sub(P) \\ Sub(!\alpha.P) &= \{!\alpha.P\} \cup Sub(P) \\ Sub(P + Q) &= \{P + Q\} \cup Sub(P) \cup Sub(Q) \\ Sub(P|Q) &= Sub(P) \cup Sub(Q) \\ Sub(\texttt{try}\, P\, \texttt{catch}\, Q) &= Sub(P) \cup Sub(Q). \end{aligned}$$

Finally, let $catch(P)$ be the set of the processes used as handlers of exceptions in try-catch operators occurring in P:

$$catch(P) = \{S \mid \exists Q : \texttt{try}\, Q\, \texttt{catch}\, S \;\; occurs\ in\ P\}.$$

The proof of the theorem is then performed by using exactly the same formal machinery as used in Busi *et al.* (2009) to prove their corresponding Theorem 8, which includes their Definition 17, Lemma 2, Propositions 10–13, Corollary 2 and Theorems 6 and 7, with the following replacements (in statements and proofs): all occurrences in Busi *et al.* (2009) of $CCS_!$ are replaced by $CCS_!^{\texttt{tc}}$; of $\equiv_w$ by $\equiv$; of $bn(P)$ by $catch(P)$; of d_v by $d_{\texttt{tc}}$; of $\longmapsto$ by $\longrightarrow$; of $(\nu x_i)Q$ by $\texttt{try}\, Q\, \texttt{catch}\, S_i$; of x_i by S_i; and, finally, in the proof of Busi *et al.* (2009, Theorem 8), we must replace the justification for the statement that the transition system of P is finitely branching by 'The fact that $(Deriv(P), \longrightarrow)$ is finitely branching follows from an inspection of the transition rules (in particular, of the operational rule for the $!\alpha.P$ operator presented in Section 2 of this paper)'. □

Corollary 5.15. Let $P \in CCS_!^{\texttt{tc}}$. The universal termination of process P is decidable.

5.2. *Universal termination is decidable in* $(CCS_{rec}^{\triangle}, \longmapsto)$

According to the ordering defined in Definition 5.13, we have $P \preceq Q$ if Q has the same structure of nesting of try-catch operators and at each point of this nesting, Q contains at least the same processes (plus some other processes in parallel). This is a well-quasi-ordering in the calculus with replication because, given P, it is possible to compute an upper bound to the number of nestings in any process in $Deriv(P)$. In the calculus with recursion, this upper bound does not exist as recursion permits the generation of nesting with unbounded depth (this is, for example, used in the deterministic RAM modelling of Section 4). For this reason, we need to move to a different ordering, which is inspired by the ordering on trees used in Kruskal (1960). This allows us to use the Kruskal Tree theorem, which states that the trees defined on a well-quasi-ordering form a well-quasi-ordering.

The remainder of this section is devoted to the definition of how to associate trees to processes of $CCS_{rec}^{\triangle}$, and how to extract from these trees an ordering for $(CCS_{rec}^{\triangle}, \longmapsto)$, which turns out to be a wqo.

https://doi.org/10.1017/S0960129509007683 Published online by Cambridge University Press

We take $\mathscr{E}$ to be the set of (open) terms of $CCS^{\triangle}_{rec}$ and $\mathscr{P}$ to be the set of $CCS^{\triangle}_{rec}$ processes, that is, closed terms. $\mathscr{P}_{seq}$ is the subset of $\mathscr{P}$ of terms P such that either $P = \mathbf{0}$ or $P = \alpha.P_1$ or $P = P_1 + P_2$ or $P = recX.P_1$, with $P_1, P_2 \in \mathscr{E}$. Let $\mathscr{P}_{int} = \{P \triangle Q \mid P, Q \in \mathscr{P}\}$.

Given a set E, we use E^* to denote the set of finite sequences of elements in E. We use ';' as a separator for elements of a set E when denoting a sequence $w \in E^*$, ϵ to denote the empty sequence and $len(w)$ to denote the length of a sequence w. Finally, we use w_i to denote the ith element in the sequence w (starting from 1) and $e \in w$ to stand for $e \in \{w_i \mid 1 \leqslant i \leqslant len(w)\}$.

Definition 5.16. Let $P \in \mathscr{P}$. We define the flattened parallel components of P, denoted $FPAR(P)$, as the sequence over $\mathscr{P}_{seq} \cup \mathscr{P}_{int}$ given by

$$FPAR(P_1|P_2) = FPAR(P_1); FPAR(P_2)$$
$$FPAR(P) = P \text{ if } P \in \mathscr{P}_{seq} \cup \mathscr{P}_{int}.$$

Given a sequence $w \in E^*$, we define the sequence $w' \in E'^*$ obtained by filtering w with respect to $E' \subseteq E$ as follows. For $1 \leqslant i \leqslant len(k)$, $w'_i = w_{k_i}$, where $k \in \{1, \ldots, len(w)\}^*$ is such that k is strictly increasing, that is, $j' > j$ implies $k_{j'} > k_j$, and, for all h, we have $w_h \in E'$ if and only if $h \in k$. In the following we use $FINT(P)$ to denote the sequence obtained by filtering $FPAR(P)$ with respect to $\mathscr{P}_{int}$ and $FSEQ(P)$ to denote the sequence obtained by filtering $FPAR(P)$ with respect to $\mathscr{P}_{seq}$.

In the following we map processes into ordered trees (with both a left to right ordering of children at every node and the usual child to parent ordering). We use $\mathbb{N}$ to denote the set of positive natural numbers, that is, $\mathbb{N} = \{1, 2, \ldots\}$.

Definition 5.17. A tree t over a set E is a partial function from $\mathbb{N}^*$ to E such that $dom(t)$ is finite, is closed with respect to sequence prefixing and is such that $\vec{n}; m \in dom(t)$ and $m' \leqslant m$, with $m' \in \mathbb{N}$, implies $\vec{n}; m' \in dom(t)$.

Example 5.18. $(\epsilon, l) \in t$ denotes the fact that the root of the tree has label $l \in E$ and $(1; 2, l) \in t$ denotes the fact that the second son of the first son of the root of the tree t has label $l \in E$.

Let $\mathscr{P}_{inth} = \{\triangle Q \mid Q \in \mathscr{P}\}$ be a set representing interruption handlers.

Definition 5.19. Let $P \in \mathscr{P}$. We define the tree of P, denoted $TREE(P)$, as the minimal tree $TREE(P)$ over $\mathscr{P}^*_{seq} \cup \mathscr{P}_{inth}$ (and minimal auxiliary tree $TREE^{odd}(P')$ over $\mathscr{P}^*_{seq} \cup \mathscr{P}_{inth}$, with $P' \in \mathscr{P}_{int}$) satisfying:

— $(\epsilon, FSEQ(P)) \in TREE(P)$
— $(\vec{n}, l) \in TREE^{odd}(FINT(P)_i)$ implies $(i; \vec{n}, l) \in TREE(P)$
— $(\epsilon, \triangle Q) \in TREE^{odd}(P' \triangle Q)$
— $(\vec{n}, l) \in TREE(P')$ implies $(1; \vec{n}, l) \in TREE^{odd}(P' \triangle Q)$.

Example 5.20. The tree t of the process

$$P = (a + b)|\big((recX.(a.X|c))\triangle R\big)|c|((a|c)\triangle S)$$

https://doi.org/10.1017/S0960129509007683 Published online by Cambridge University Press

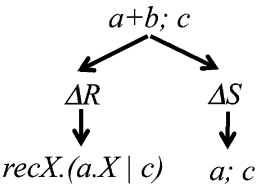

Fig. 1. Tree t associated with process P of Example 5.20.

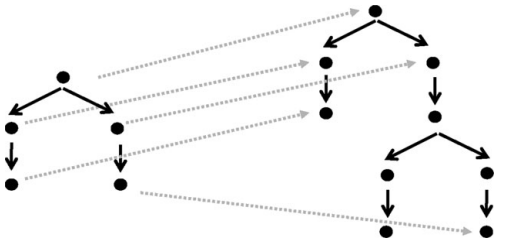

Fig. 2. The function φ from $dom(t)$ to $dom(t')$ of Example 5.22 strictly preserves order inside trees.

for some processes R and S is

$$t = \{(\epsilon, a + b; c), (1, \triangle R), (1; 1, recX.(a.X|c)), (2, \triangle S), (2; 1, a; c)\}.$$

The tree t is shown in Figure 1.

In the following, we define the ordering between processes using the ordering on trees used in Kruskal (1960) applied to the particular trees obtained from processes by our transformation procedure. In order to do this, we introduce the notion of an injective function that strictly preserves order inside trees: a possible formal way to express homeomorphic embedding between trees, which is used in Kruskal's Theorem (Kruskal 1960), and which we take from Simpson (1985).

We take $\leqslant_T$ to be the ancestor pre-order relation inside trees, which is defined by $\vec{n} \leqslant_T \vec{m}$ if and only if $\vec{m}$ is a prefix of $\vec{n}$ (or $\vec{m} = \vec{n}$). Moreover, we take $\wedge_T$ to be the minimal common ancestor of a pair of nodes, that is, $\vec{n}_1 \wedge_T \vec{n}_2 = min\{\vec{m} | \vec{n}_1 \leqslant_T \vec{m} \wedge \vec{n}_2 \leqslant_T \vec{m}\}$.

Definition 5.21. We say that an injective function φ from $dom(t)$ to $dom(t')$ strictly preserves order inside trees if and only if for every $\vec{n}, \vec{m} \in dom(t)$, we have:

— $\vec{n} \leqslant_T \vec{m}$ implies $\varphi(\vec{n}) \leqslant_T \varphi(\vec{m})$.
— $\varphi(\vec{n} \wedge_T \vec{m}) = \varphi(\vec{n}) \wedge_T \varphi(\vec{m})$.

Example 5.22. Consider tree t of Example 5.20 and a tree t' such that

$$dom(t') = \{\epsilon, 1, 1; 1, 2, 2; 1, 2; 1; 1, 2; 1; 1; 1, 2; 1; 2, 2; 1; 2; 1)\}.$$

The injective function

$$\varphi = \{(\epsilon, \epsilon), (1, 1), (1; 1, 1; 1), (2, 2), (2; 1, 2; 1; 2; 1)\},$$

which is shown in Figure 2, strictly preserves order inside trees.

https://doi.org/10.1017/S0960129509007683 Published online by Cambridge University Press

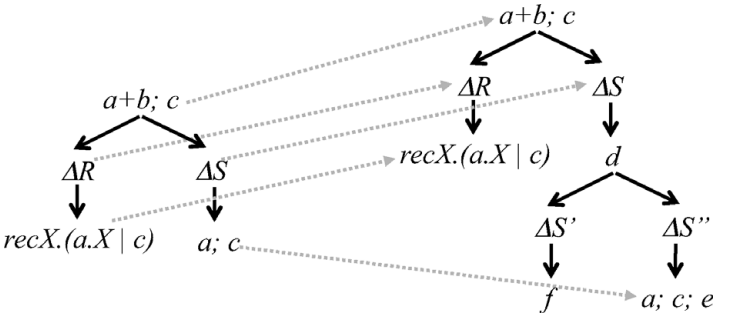

Fig. 3. Function φ shows that $P \preceq Q$ for P and Q of Example 5.24.

Definition 5.23. Let $P, Q \in \mathscr{P}$. $P \preceq Q$ if and only if there exists an injective function φ from $dom(TREE(P))$ to $dom(TREE(Q))$ such that φ strictly preserves order inside trees and for every $\vec{n} \in dom(\varphi)$, either:

— there exists $R \in \mathscr{P}$ such that $TREE(P)(\vec{n}) = TREE(Q)(\varphi(\vec{n})) = \triangle R$; or

— $TREE(P)(\vec{n}), TREE(Q)(\varphi(\vec{n})) \in \mathscr{P}^*_{seq}$ and, if $len(TREE(P)(\vec{n})) > 0$, there exists an injective function f from

$$\{1, \ldots, len(TREE(P)(\vec{n}))\}$$

to

$$\{1, \ldots, len(TREE(Q)(\varphi(\vec{n})))\}$$

such that for every $i \in dom(f)$, we have $TREE(P)(\vec{n})_i = TREE(Q)(\varphi(\vec{n}))_{f(i)}$.

Notice that $\preceq$ is a quasi-ordering in that it is obviously reflexive and one can immediately verify, taking into account the two conditions for the injective function in the above definition, that it is transitive.

Example 5.24. Consider the process

$$P = (a + b)|\big((recX.(a.X|c))\triangle R\big)|c|((a|c)\triangle S)$$

of Example 5.20 and its associated tree t. Also consider process

$$Q = (a + b)|\big((recX.(a.X|c))\triangle R\big)|c|\big((d|(f\triangle S''))|((a|c|e)\triangle S')\triangle S\big),$$

for some processes R, S, S' and S'', and its associated tree

$$t' = \{(\epsilon, a + b; c), (1, \triangle R), (1; 1, recX.(a.X|c)), (2, \triangle S), (2; 1, d), \\ (2; 1; 1, \triangle S'), (2; 1; 1; 1, f), (2; 1; 2, \triangle S''), (2; 1; 2; 1, a; c; e)\}.$$

The domain of t' is the same as in Example 5.22, hence the function

$$\varphi = \{(\epsilon, \epsilon), (1, 1), (1; 1, 1; 1), (2, 2), (2; 1, 2; 1; 2; 1)\}$$

strictly preserves order inside trees. It is easy to observe (see Figure 3) that φ also maps exception handlers into identical exception handlers and sequences of sequential terms into sequences of sequential terms that include a larger (multi)set of sequential terms, hence $P \preceq Q$.

https://doi.org/10.1017/S0960129509007683 Published online by Cambridge University Press

We now redefine on the transition system $(CCS^{\triangle}_{rec}, \longmapsto)$ the function $Deriv(P)$ that associates to a process the set of its derivatives.

Definition 5.25. Let $P \in CCS^{\triangle}_{rec}$. We use $Deriv(P)$ to denote the set of processes reachable from P through a sequence of reduction steps:

$$Deriv(P) = \{Q \mid P \longmapsto^{*} Q\}.$$

We are now ready to state our main result, which can be proved by simultaneously exploiting Higman's Theorem on sequences (Higman 1952) (also known as Higman's Lemma) and Kruskal's Theorem on trees (Kruskal 1960).

Theorem 5.26. Let $P \in CCS^{\triangle}_{rec}$. Then the transition system $(Deriv(P), \longmapsto, \preceq)$ is a finitely branching, well-structured transition system with strong compatibility, decidable $\preceq$ and computable $Succ$.

Proof. See Section 5.2.1. □

Corollary 5.27. Let $P \in CCS^{\triangle}_{rec}$. The termination of process P is decidable.

As replication is a particular case of recursion, the same decidability result also holds for $CCS^{\triangle}_{!}$.

5.2.1. *Proving Theorem 5.26* We first extend the definitions of sequential and interruption terms and of $FPAR(P)$ to open terms. $\mathscr{E}_{seq}$ is the subset of $\mathscr{E}$ of terms P such that either $P = \mathbf{0}$ or $P = \alpha.P_1$ or $P = P_1 + P_2$ or $P = recX.P_1$, with $P_1, P_2 \in \mathscr{E}$. Let $\mathscr{E}_{int} = \{P \triangle Q \mid P, Q \in \mathscr{E}\}$. We extend the definition of $FPAR(P)$ to open terms $P \in \mathscr{E}$ by replacing the second clause in the definition of $FPAR(P)$ with:

$$FPAR(P) = P \quad \text{if} \quad P \in \mathscr{E}_{seq} \cup \mathscr{E}_{int} \cup \{X | X \in Vars\}$$

where $Vars$ is the denumerable set of variables X in the syntax of $\mathscr{E}$ terms.

A context is a term P_X of $\mathscr{E}$ that includes a single occurrence of the free variable X (and possibly other free variables). A flat parallel context is a term P^i_X of $\mathscr{E}$ such that P^i_X is a context and the sequence $w \in (\mathscr{E}_{int} \cup \{X\})^*$ obtained by filtering $FPAR(P^i_X)$ with respect to $\mathscr{E}_{int} \cup \{X\}$ is such that $w_i = X$.

Definition 5.28. Let $P \in \mathscr{P}$. We define the context tree of P, denoted $CON_X(P)$, as the minimal tree $CON_X(P)$ over contexts P_X (and minimal auxiliary tree $CON^{odd}_X(P')$ over contexts P_X, with $P' \in \mathscr{P}_{int}$) satisfying:

— $(\epsilon, X) \in CON_X(P)$.
— $(\vec{n}, P'_X) \in CON^{odd}_X(P')$, with $P' \in \mathscr{P}_{int}$, implies

$$(i; \vec{n}, P^i_X\{P'_X/X\}) \in CON_X(P^i_X\{P'/X\}).$$

— $(\epsilon, X) \in CON^{odd}_X(P' \triangle Q)$.
— $(\vec{n}, P'_X) \in CON_X(P')$ implies $(1; \vec{n}, P'_X \triangle Q) \in CON^{odd}_X(P' \triangle Q)$.

https://doi.org/10.1017/S0960129509007683 Published online by Cambridge University Press

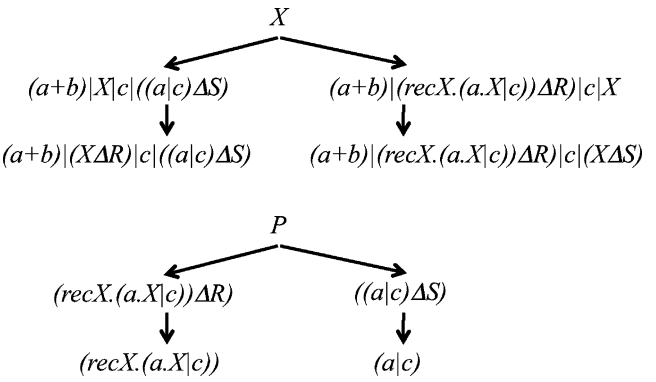

Fig. 4. $CON_X(P)$ and $SUBT(P)$ for the process P of Example 5.29, where

$$P = (a + b)|((recX.(a.X|c))\triangle R)|c|((a|c)\triangle S).$$

We define the subterm tree of P, denoted $SUBT(P)$, as the tree satisfying the following condition. $(\vec{n}, P') \in SUBT(P)$ if and only if there exists a context P_X such that $(\vec{n}, P_X) \in CON_X(P)$ and P' is the term such that $P_X\{P'/X\} = P$. Similarly, $(\vec{n}, P') \in SUBT^{odd}(P)$, with $P \in \mathscr{P}_{int}$, if and only if there exists a context P_X such that $(\vec{n}, P_X) \in CON_X^{odd}(P)$ and P' is the term such that $P_X\{P'/X\} = P$.

Example 5.29. Consider process

$$P = (a + b)|\big((recX.(a.X|c))\triangle R\big)|c|((a|c)\triangle S)$$

of Example 5.20. The tree $CON_X(P)$ is

$$\begin{aligned}&\{(\epsilon, X),\\ &(1, (a + b)|X|c|((a|c)\triangle S)),\\ &(1;1, (a + b)|\big(X\triangle R\big)|c|((a|c)\triangle S)),\\ &(2, (a + b)|\big((recX.(a.X|c))\triangle R\big)|c|X),\\ &(2;1, (a + b)|\big((recX.(a.X|c))\triangle R\big)|c|(X\triangle S))\}.\end{aligned}$$

The tree $SUBT(P)$ is

$$\{(\epsilon, P), (1, (recX.(a.X|c))\triangle R), (1;1, recX.(a.X|c)), (2, (a|c)\triangle S), (2;1, a|c)\}.$$

Both trees are shown in Figure 4.

In the following we will use $TREE^{\vec{n}}(P)$ to stand for $TREE(P)(\vec{n})$, $CON^{\vec{n}}(P)$ to stand for $CON(P)(\vec{n})$ and $SUBT^{\vec{n}}(P)$ to stand for $SUBT(P)(\vec{n})$.

Proposition 5.30.

— $(\vec{n}, l) \in TREE(P)$ if and only if $(\vec{n}, P') \in SUBT(P)$ and:
  – $l = FSEQ(P')$ if $len(\vec{n})$ is even or zero;
  – $l = \triangle Q$, with $P' = P''\triangle Q$ for some P'', if $len(\vec{n})$ is odd.

— $(\vec{n}, l) \in TREE^{odd}(P)$, with $P \in \mathscr{P}_{int}$, if and only if $(\vec{n}, P') \in SUBT^{odd}(P)$ and:

https://doi.org/10.1017/S0960129509007683 Published online by Cambridge University Press

- $l = \triangle Q$, with $P' = P''\triangle Q$ for some P'', if $len(\vec{n})$ is even or zero;
- $l = FSEQ(P')$ if $len(\vec{n})$ is odd.

Proof. First note that $FINT(P)_i = P'$ if and only if there exists a flat parallel context P^i_X such that $P^i_X\{P'/X\} = P$ and $P' \in \mathscr{P}_{int}$. Using this, we can rewrite the second clause in the definition of $TREE(P)$ as

$$(\vec{n}, l) \in TREE^{odd}(P'), \quad \text{with} \quad P' \in \mathscr{P}_{int}, \quad \text{implies} \quad (i; \vec{n}, l) \in TREE(P^i_X\{P'/X\}),$$

which makes it similar to the corresponding clause in the definition of $CON_X(P)$.

The statement is then trivially proved to hold by induction on $len(\vec{n})$, with $len(\vec{n}) = 0$ as the base case. □

In the following we use $rchn(t)$ to denote the number of children of the root of a (non-empty) tree t. We have $rchn(t) = max\{k | k \in dom(t)\}$.

Definition 5.31. Let $w, w' \in E^*$. We define the sequence obtained by inserting w' in w at position i, with $1 \leqslant i \leqslant len(w) + 1$, as the sequence w'' with length $len(w) + len(w')$ such that:

— $\forall 1 \leqslant n \leqslant i - 1.\, w''_n = w_n$.
— $\forall 1 \leqslant n \leqslant len(w').\, w''_{i-1+n} = w'_n$.
— $\forall i \leqslant n \leqslant len(w).\, w''_{len(w')+n} = w_n$.

Let t, t' be trees over E such that $(\epsilon, w) \in t$ and $(\epsilon, w') \in t'$ with $w, w' \in E'^* \subseteq E$. We define the tree obtained by inserting t' in t at position (i, j), with $1 \leqslant i \leqslant len(w) + 1$ and $1 \leqslant j \leqslant rchn(t) + 1$, as the minimal tree t'' such that:

— $(\epsilon, w'') \in t''$ where w'' is obtained by inserting w' in w at position i.
— $\forall \vec{n}, 1 \leqslant m \leqslant j - 1.\, (m; \vec{n}, l) \in t$ implies $(m; \vec{n}, l) \in t''$.
— $\forall \vec{n}, 1 \leqslant m \leqslant rchn(t').\, (m; \vec{n}, l) \in t'$ implies $(j - 1 + m; \vec{n}, l) \in t''$.
— $\forall \vec{n}, j \leqslant m \leqslant rchn(t).\, (m; \vec{n}, l) \in t$ implies $(rchn(t') + m; \vec{n}, l) \in t''$.

Finally, we define the sequence obtained from $w \in E^*$ by removing the ith element, with $1 \leqslant i \leqslant len(w)$, written $w - i$, as the sequence $w' \in E^*$ such that $len(w') = len(w) - 1$, $\forall 1 \leqslant n \leqslant i - 1.\, w'_n = w_n$ and $\forall i + 1 \leqslant n \leqslant len(w).\, w'_{n-1} = w_n$.

We now prove that $\preceq$ satisfies the strong compatibility property with respect to the transition system $(CCS^{\triangle}_{rec}, \longmapsto)$. The proof exploits the following lemma.

Lemma 5.32. $SUBT^{\vec{n}}(P) \overset{\alpha}{\longmapsto} P'$ implies $P \overset{\alpha}{\longmapsto} CON^{\vec{n}}_X(P)\{P'/X\}$.

Proof. We show by induction on $len(\vec{n})$, where 0 is the base case, that:

— $SUBT^{\vec{n}}(P) \overset{\alpha}{\longmapsto} P'$ implies $P \overset{\alpha}{\longmapsto} CON^{\vec{n}}_X(P)\{P'/X\}$.
— $SUBT^{odd}(P)(\vec{n}) \overset{\alpha}{\longmapsto} P'$ implies $P \overset{\alpha}{\longmapsto} CON^{odd}_X(P)(\vec{n})\{P'/X\}$.

The induction step is worked out as a trivial consequence of the fact that $P \overset{\alpha}{\longmapsto} P'$ implies $P\triangle Q \overset{\alpha}{\longmapsto} P'\triangle Q$ and $P|Q \overset{\alpha}{\longmapsto} P'|Q$. □

Theorem 5.33. Let $P, Q, P' \in \mathscr{P}$. If $P \overset{\alpha}{\longmapsto} P'$ and $P \preceq Q$, then there exists $Q' \in \mathscr{P}$ such that $Q \overset{\alpha}{\longmapsto} Q'$ and $P' \preceq Q'$.

https://doi.org/10.1017/S0960129509007683 Published online by Cambridge University Press

Proof. The proof is by induction on $depth(TREE(P))/2$, where for any tree t we take $depth(t) = max\{len(\vec{n}) | \vec{n} \in dom(t)\}$. It is easy to see that for every $P \in \mathscr{P}$, we have that $depth(TREE(P))$ is even (because $\vec{n} \in dom(TREE(P))$ and $len(\vec{n})$ odd implies $\vec{n};1 \in dom(TREE(P))$).

Therefore, we will prove the assertion for any P having a certain $depth(TREE(P))/2$, taking $depth(TREE(P))/2 = 0$ as the base case of the induction.

Since $P \preceq Q$, there exists an injective strictly order-preserving φ such that the labels of $TREE(P)$ and $TREE(Q)$ are correctly related. In particular, there exists an injection f from $\{1,\ldots,len(TREE^{\epsilon}(P))\}$ to $\{1,\ldots,len(TREE^{\varphi(\epsilon)}(Q))\}$ such that $TREE^{\epsilon}(P)_i = TREE^{\varphi(\epsilon)}(Q)_{f(i)}$. There are two cases:

(a) $P \stackrel{\alpha}{\longmapsto} P'$ is inferred from the move $P_1 \stackrel{\alpha}{\longmapsto} P_1'$ of a single process $P_1 \in FPAR(P)$; or
(b) $\alpha = \tau$ and $P \stackrel{\alpha}{\longmapsto} P'$ is inferred from the moves $P_1 \stackrel{\overline{a}}{\longmapsto} P_1'$ and $P_2 \stackrel{a}{\longmapsto} P_2'$ of two processes $P_1, P_2 \in FPAR(P)$.

In the following we develop (a), then show that (b) can be treated as a consequence of (a). There are three subcases for (a):

(1) $P_1 \in FSEQ(P)$.
(2) $P_1 = P_2 \triangle P_3$, for some P_2, P_3, and $P_1 \stackrel{\alpha}{\longmapsto} P_1'$ is inferred from a move $P_3 \stackrel{\alpha}{\longmapsto} P_1'$ of the process to the right-hand side of $\triangle$.
(3) $P_1 = P_2 \triangle P_3$, for some P_2, P_3, and $P_1 \stackrel{\alpha}{\longmapsto} P_1'$ is inferred from a move $P_2 \stackrel{\alpha}{\longmapsto} P_2'$ of the process to the left-hand side of $\triangle$.

Note that the second and third subcases can only arise if $depth(TREE(P))/2 > 0$. We give the proof for each of the subcases in turn:

(1) Assume that P_1 is the process at the jth position in the sequence $FSEQ(P)$ and that the first process of $\mathscr{P}_{int}$ to the right of P_1 in the sequence $FPAR(P)$ is at the kth position in the sequence $FINT(P)$.
First notice that $TREE(P')$ is obtained by inserting $TREE(P_1')$ in the tree

$$TREE(P) - \{(\epsilon, FSEQ(P))\} \cup \{(\epsilon, FSEQ(P) - j)\}$$

at position (j,k).
From the existence of the function f above, we derive $P_1 \in TREE^{\varphi(\epsilon)}(Q)$. Let $Q_1 = SUBT^{\varphi(\epsilon)}(Q)$. Assume that P_1 is the process at the j'th position in the sequence $FSEQ(Q_1)$ and that the first process of $\mathscr{P}_{int}$ to the right of P_1 in the sequence $FPAR(Q_1)$ is at the k'th position in the sequence $FINT(Q_1)$. We have $Q_1 \stackrel{\alpha}{\longmapsto} Q_1'$, where $TREE(Q_1')$ is obtained by inserting $TREE(P_1')$ in the tree

$$TREE(Q_1) - \{(\epsilon, FSEQ(Q_1))\} \cup \{(\epsilon, FSEQ(Q_1) - j)\}$$

at position (j',k').

https://doi.org/10.1017/S0960129509007683 Published online by Cambridge University Press

Using Lemma 5.32, we derive $Q \stackrel{\alpha}{\longmapsto} CON_X^{\varphi(\epsilon)}(Q)\{Q'_1/X\}$. Now consider

$$\begin{aligned}\varphi' \;=\; & \{(\epsilon, \varphi(\epsilon))\} \cup \\ & \{(m;\vec{n}, \varphi(\epsilon); z; \vec{n'}) | 1 \leqslant m \leqslant k-1 \wedge \\ & \;((\varphi(\epsilon); z; \vec{n'} = \varphi(m;\vec{n}) \wedge \varphi(m;\vec{n}) \leqslant k'-1) \vee \\ & \;(\varphi(\epsilon); z - (rchn(t'); \vec{n'} = \varphi(m;\vec{n}) \wedge \varphi(m;\vec{n}) \geqslant k'))\} \cup \\ & \{(k-1+m;\vec{n}, \varphi(\epsilon); k'-1+m;\vec{n}) | (m;\vec{n}) \in dom(t')\} \cup \\ & \{(rchn(t')+m;\vec{n}, \varphi(\epsilon); z; \vec{n'}) | k \leqslant m \leqslant rchn(t) \wedge \\ & \;((\varphi(\epsilon); z; \vec{n'} = \varphi(m;\vec{n}) \wedge \varphi(m;\vec{n}) \leqslant k'-1) \vee \\ & \;(\varphi(\epsilon); z - (rchn(t'); \vec{n'} = \varphi(m;\vec{n}) \wedge \varphi(m;\vec{n}) \geqslant k'))\},\end{aligned}$$

where $t = TREE(P)$ and $t' = TREE(P'_1)$. φ' is a strictly order-preserving injection such that the labels of P' and $Q' = CON_X^{\varphi(\epsilon)}(Q)\{Q'_1/X\}$ are correctly related, so $P' \preceq Q'$. In particular, the existence of an injection f' from $\{1,\ldots,len(TREE^{\epsilon}(P'))\}$ to $\{1,\ldots,len(TREE^{\varphi'(\epsilon)}(Q'))\}$ such that $TREE^{\epsilon}(P')_i = TREE^{\varphi'(\epsilon)}(Q')_{f'(i)}$ derives from the existence of the injection f between the sequences labelling the nodes ϵ and $\varphi(\epsilon) = \varphi'(\epsilon)$ of P and Q: f' is obtained from f by removing the pair (j, j') (corresponding to the removal of P_1 from both sequences) and then simply accounting for the insertion of the same sequence $FSEQ(P'_1)$ in both sides. Notice that the preservation of the minimal common ancestor property holds because, when, in $TREE(P')$, $i;1;\vec{n}$ nodes, with $k \leqslant i \leqslant k+rchn(TREE(P'))$, are involved and are considered together with nodes $j;1;\vec{m}$ for some $j < k \vee j > k + rchn(TREE(P')), \vec{m}$, the minimal common ancestor is the root ϵ. Moreover, the nodes $\varphi'(i;1;\vec{n}) = \varphi'(\epsilon); i-k+k';1;\vec{n}$ and $\varphi'(j)$ have $\varphi'(\epsilon)$ as a minimal common ancestor – this follows by the construction of $TREE(Q')$ from $TREE(Q)$ and because the injection φ preserves the ancestor pre-order.

(2) We assume that P_1 is the process at the ith position in the sequence $FINT(P)$.
We have $TREE^i(P) = TREE^{\varphi(i)}(Q) = \triangle P_3$, so $SUBT^{\varphi(i)}(Q) = Q_1 \triangle P_3$ for some $Q_1 \in \mathscr{P}$ and $SUBT^{\varphi(i)}(Q) \stackrel{\alpha}{\longmapsto} P'_3 \triangle \mathbf{0}$.
Using Lemma 5.32, we derive $Q \stackrel{\alpha}{\longmapsto} CON_X^{\varphi(i)}(Q)\{P'_1 \triangle \mathbf{0}/X\}$. Now consider

$$\varphi' = \varphi - \{(i;1;\vec{n},\vec{m}) | \vec{n},\vec{m} \in \mathbf{N}^*\} \cup \{(i;1;\vec{n}, \varphi(i);1;\vec{n}) | \vec{n} \in dom(TREE(P'_1))\}.$$

φ' is a strictly order-preserving injection such that the labels of P' and

$$Q' = CON_X^{\varphi(i)}(Q)\{P'_1 \triangle \mathbf{0}/X\}$$

are correctly related, so $P' \preceq Q'$. Notice that the preservation of the minimal common ancestor property holds because, when, in $TREE(P')$, $i;1;\vec{n}$ nodes are involved and are considered together with nodes $j;1;\vec{m}$ for some $j \neq i, \vec{m}$, the minimal common ancestor is the root ϵ. Since the nodes i, j have ϵ as a minimal common ancestor, $\varphi(i)$ and $\varphi(j)$ must have $\varphi(\epsilon)$ as a minimal common ancestor. Moreover, the injection φ preserves the ancestor pre-order and φ' differs from φ just over nodes $\varphi(i;1;\vec{n'})$ for some $\vec{n'}$, so $\varphi(i;1;\vec{n})$ and $\varphi(j;1;\vec{m})$ must have $\varphi'(\epsilon)$ as a minimal common ancestor.

(3) We assume that P_1 is the process at the ith position in the sequence $FINT(P)$.

https://doi.org/10.1017/S0960129509007683 Published online by Cambridge University Press

We have $\varphi' = \{(\vec{n},\vec{m}) | \varphi(i;1;\vec{n}) = \varphi(i;1);\vec{m}\}$ (which makes sense because φ preserves the ancestor pre-order) is a strictly order-preserving injection such that the labels of P_2 and $SUBT_{\varphi(i;1)}(Q)$ are correctly related; hence $P_2 \preceq SUBT_{\varphi(i;1)}(Q)$.
By applying the induction hypothesis, we can show that there exists $Q_1' \in \mathscr{P}$ such that $SUBT_{\varphi(i;1)}(Q) \stackrel{\alpha}{\longmapsto} Q_1'$ and $P_2' \preceq Q_1'$, so there exists a strictly order-preserving injection φ'' such that the labels of $TREE(P_2')$ and $TREE(Q_1')$ are correctly related.
Using Lemma 5.32, we can derive $Q \stackrel{\alpha}{\longmapsto} CON_X^{\varphi(i;1)}(Q)\{Q_1'/X\}$. Now consider

$$\varphi''' = \varphi - \{(i;1;\vec{n},\vec{m}) | \vec{n},\vec{m} \in \mathbb{N}^*\} \cup \{(i;1;\vec{n},\varphi(i;1);\vec{m}) | \varphi''(\vec{n}) = \vec{m}\}.$$

φ''' is a strictly order-preserving injection such that the labels of P' and $Q' = CON_X^{\varphi(i;1)}(Q)\{Q_1'/X\}$ are correctly related, so $P' \preceq Q'$. Notice that the preservation of the minimal common ancestor property holds because, when, in $TREE(P')$, $i;1;\vec{n}$ nodes are involved and are considered together with nodes $j;1;\vec{m}$ for some $j \neq i,\vec{m}$, the minimal common ancestor is the root ϵ. Since the nodes i, j have ϵ as a minimal common ancestor, $\varphi(i)$ and $\varphi(j)$ must have $\varphi(\epsilon)$ as a minimal common ancestor. Moreover, the injection φ preserves the ancestor pre-order and φ''' differs from φ just over nodes $\varphi(i;1;\vec{n'})$ for some $\vec{n'}$, so $\varphi(i;1;\vec{n})$ and $\varphi(j;1;\vec{m})$ must have $\varphi'''(\epsilon)$ as a minimal common ancestor.

Case (b), $\alpha = \tau$ and $P \stackrel{\alpha}{\longmapsto} P'$, is inferred from the moves $P_1 \stackrel{\overline{a}}{\longmapsto} P_1'$ and $P_2 \stackrel{a}{\longmapsto} P_2'$ of two processes P_1, P_2 at different positions in $FPAR(P)$. Suppose P_1 is at position i in $FPAR(P)$ and P_2 is at position j in $FPAR(P)$.

Consider the flat parallel context P_X^i such that $P_X^i\{P_1/X\} = P$ and P_X^j such that $P_X^j\{P_2/X\} = P$. We have $P \stackrel{\overline{a}}{\longmapsto} P_X^i\{P_1'/X\}$ and $P \stackrel{a}{\longmapsto} P_X^j\{P_2'/X\}$. Therefore, by the same proof as in the previous case, there exists Q'' and Q''' such that $Q \stackrel{\overline{a}}{\longmapsto} Q''$ and $Q \stackrel{a}{\longmapsto} Q'''$. Now observe that, no matter which of the three cases above applies for the inference of a move of P, in the above proof we identify Q_1 and Q_2 in $FPAR(SUBT^{\varphi(\epsilon)}(Q))$ such that $Q_1 \stackrel{\overline{a}}{\longmapsto} Q_1'$ and $Q_2 \stackrel{a}{\longmapsto} Q_2'$. Moreover, Q_1 and Q_2 are at different positions in $FPAR(SUBT^{\varphi(\epsilon)}(Q))$ because:

— If Q_1 and Q_2 are both $\mathscr{P}_{seq}$ terms (subcase 1 above), the injective function f yields Q_1 and Q_2 at different positions because P_1, P_2 are at different positions.
— If Q_1 and Q_2 are both $\mathscr{P}_{int}$ terms (subcases 2 and 3), and using k for the position of P_1 in $FINT(P)$ and h for the position of P_2 in $FINT(P)$ (obviously $k \neq h$), then Q_1 is the term at position k', where $k';\vec{n} = \varphi(k)$ for some $\vec{n}$, in $FINT(SUBT^{\varphi(\epsilon)}(Q))$ and Q_2 is the term at position h', where $h';\vec{m} = \varphi(h)$ for some $\vec{m}$, in $FINT(SUBT^{\varphi(\epsilon)}(Q))$ and we must have $k' \neq h'$ because otherwise φ would not preserve the minimal common ancestor of nodes k and h (which is the root ϵ).

In order to prove the assertion, we need to use another property of the terms Q' and injective functions φ' (and the related injective function f' relating $TREE^{\epsilon}(P')$ and $TREE^{\varphi(\epsilon)}(Q')$) showing that $P' \preceq Q'$ built in the previous case (subcases 1, 2 and 3). Consider a term in $FSEQ(P)$ that is not P_1 and let z be its position in $FSEQ(P)$. If m is the position that such a term assumes in $FSEQ(P_X^i\{P_1'/X\})$, then $f'(m)$ is the position that the $f(z)$th term in $FSEQ(SUBT^{\varphi(\epsilon)}(Q))$ (which is not Q_1 because f is injective) assumes

https://doi.org/10.1017/S0960129509007683 Published online by Cambridge University Press

in $FSEQ(Q_X^{i'}\{Q_1'/X\})$, where $Q_X^{i'}$ is such that $Q = CON_X^{\varphi(\epsilon)}(Q)\{Q_X^{i'}\{Q_1/X\}/X\}$. Similarly, consider a term in $FINT(P)$ that is not P_1 and let z be its position in $FINT(P)$. If m is the position that such a term assumes in $FINT(P_X^i\{P_1'/X\})$, then m' such that $m';\vec{n} = \varphi'(m)$, for some $\vec{n}$, is the position that the z'th term in $FINT(SUBT^{\varphi(\epsilon)}(Q))$, with $z';\vec{n} = \varphi(z)$ (which is not Q_1 because φ is injective and preserves the minimal common ancestor), assumes in $FINT(Q_X^{i'}\{Q_1'/X\})$.

Since the two processes P_1, P_2 are at different positions in $FPAR(P)$, then $P \stackrel{\overline{a}}{\longmapsto} P_X^i\{P_1'/X\} \stackrel{a}{\longmapsto} P'$, where the two moves are inferred from the moves $P_1 \stackrel{\overline{a}}{\longmapsto} P_1'$ and $P_2 \stackrel{a}{\longmapsto} P_2'$, respectively. From the first move, by building a corresponding move for Q to Q' and an injective function φ' (and the related injective function f' relating $TREE^{\epsilon}(P')$ and $TREE^{\varphi(\epsilon)}(Q')$) as in the previous case, we have

$$Q \stackrel{\overline{a}}{\longmapsto} Q' \equiv CON_X^{\varphi(\epsilon)}(Q)\{Q_X^{i'}\{Q_1'/X\}/X\},$$

which is inferred from a move $Q_1 \stackrel{\overline{a}}{\longmapsto} Q_1'$ of Q_1. Moreover, if $P_2 \in \mathscr{P}_{seq}$ and m is the position that term P_2 (which is inside P_X^i) assumes in $FSEQ(P_X^i\{P_1'/X\})$, then $f'(m)$ is the position that Q_2 (which is inside $Q_X^{i'}$) assumes in $FSEQ(Q_X^{i'}\{Q_1'/X\})$. If, instead, $P_2 \in \mathscr{P}_{int}$ and m is the position term P_2 (which is inside P_X^i) assumes in $FINT(P_X^i\{P_1'/X\})$, then m' such that $m';\vec{n} = \varphi'(m)$, for some $\vec{n}$, is the position that Q_2 (that is inside $Q_X^{i'}$) assumes in $FINT(Q_X^{i'}\{Q_1'/X\})$. From the second move, we have that we can similarly build a move from $CON_X^{\varphi(\epsilon)}(Q)\{Q_X^{i'}\{Q_1'/X\}/X\}$ to $CON_X^{\varphi(\epsilon)}(Q)\{Q''/X\}$, which is inferred from a move $Q_2 \stackrel{a}{\longmapsto} Q_2'$ of Q_2 (because of the correspondence between P_2 and Q_2 in φ' detailed above), and an injective function φ'' showing that $P' \preceq CON_X^{\varphi(\epsilon)}(Q)\{Q''/X\}$. Therefore, since Q_1 and Q_2 are in different positions in $FPAR(SUBT^{\varphi(\epsilon)}(Q))$, we have $SUBT^{\varphi(\epsilon)}(Q) \stackrel{\tau}{\longmapsto} Q''$, so, by Lemma 5.32, $Q \stackrel{\tau}{\longmapsto} CON_X^{\varphi(\epsilon)}(Q)\{Q''/X\}$. □

In order to prove that $\preceq$ is a wqo, we exploit both Higman's Theorem, which allows us to lift a wqo on a set $\mathscr{S}$ to a corresponding wqo on $\mathscr{S}^*$, that is, the set of finite sequences on $\mathscr{S}$, and Kruskal's Tree Theorem, which allows us to lift a wqo on a set $\mathscr{S}$ to a corresponding wqo on trees over $\mathscr{S}$.

Definition 5.34. Let $\mathscr{S}$ be a set and $\leqslant$ a wqo over $\mathscr{S}$. The relation $\leqslant_*$ over $\mathscr{S}^*$ is defined as follows. Let $t, u \in \mathscr{S}^*$, with $t = t_1 t_2 \ldots t_m$ and $u = u_1 u_2 \ldots u_n$. We define $t \leqslant_* u$ if and only if there exists an injection f from $\{1, 2, \ldots, m\}$ to $\{1, 2, \ldots, n\}$ such that $t_i \leqslant u_{f(i)}$ and $1 \leqslant f(1) < \ldots < f(m) \leqslant n$.

Note that the relation $\leqslant_*$ is a quasi-ordering over $\mathscr{S}^*$.

Theorem 5.35 (Higman). Let $\mathscr{S}$ be a set and $\leqslant$ be a wqo over $\mathscr{S}$. Then the relation $\leqslant_*$ is a wqo over $\mathscr{S}^*$.

Definition 5.36. Let $\mathscr{S}$ be a set and $\leqslant$ a wqo over $\mathscr{S}$. The relation $\leqslant^{tree}$ on the set of trees over $\mathscr{S}$ is defined as follows. Let t, u be trees over $\mathscr{S}$. We define $t \leqslant^{tree} u$ if and only if there exists a strictly order-preserving injection φ from $dom(t)$ to $dom(u)$ such that for every $\vec{n} \in dom(\varphi)$ we have $t(\vec{n}) \leqslant u(\varphi(\vec{n}))$.

https://doi.org/10.1017/S0960129509007683 Published online by Cambridge University Press

Note that the relation $\leqslant^{tree}$ is a quasi-ordering for the set of trees over $\mathscr{S}$.

Theorem 5.37 (Kruskal). Let $\mathscr{S}$ be a set and $\leqslant$ be a wqo over $\mathscr{S}$. Then the relation $\leqslant^{tree}$ is a wqo on the set of trees over $\mathscr{S}$.

Moreover, the following trivial proposition will be used to show that $\preceq$ is a wqo.

Proposition 5.38. Let $\mathscr{S}$ be a finite set. Then equality is a wqo over $\mathscr{S}$.

In order to apply Kruskal, we need first to define a wqo on the set of labels of the trees associated to the derivatives of a process P. To prove this result, we show that these labels are sequences of elements taken from a finite domain (then we can apply Proposition 5.38 and Theorem 5.35). We first introduce some auxiliary notation.

Given a process $P \in CCS^{\triangle}_{rec}$, we define the set $\mathscr{P}_{seq}(P)$ of the sequential subprocesses of P and the set $\mathscr{P}_{inth}(P)$ of the interruption handlers that are included in P as follows:

$$\mathscr{P}_{seq}(P) = \{Q \in \mathscr{P}_{seq} \mid Q \in ClSub(P)\}$$
$$\mathscr{P}_{inth}(P) = \{\triangle Q \in \mathscr{P}_{inth} \mid \exists R \in \mathscr{P} : R \triangle Q \in ClSub(P)\}$$

where $ClSub(P)$ is the set of the closures of the subterms of P, that is, $P' \in ClSub(P)$ if the closed term P' is obtained from a subterm P'' of P by replacing each free variable in P'' with the corresponding binding recursion inside P. For instance, if

$$P = recX.a.recY.(b.X + c.Y)$$

we have, for example, that

$$b.\big(recX.a.recY.(b.X + c.Y)\big) + c.recY.\big(b.\big(recX.a.recY.(b.X + c.Y)\big) + c.Y\big)$$

is in $ClSub(P)$ in that it is obtained as the closure of the subterm $b.X + c.Y$ of P. Note that, obviously (since we always consider variable definitions as they appear inside P, which does not change during the replacements), the order of variable replacement is not important given that we replace variables until we reach a closed term: if we replace variable definitions considering them from inner to outer scopes, that is, the definition of Y before the definition of X in the example above, we do not have to repeat any replacements for the same variable.

Since subterms of a term are finitely many and for each term there is exactly one closure of it, it follows immediately that both $\mathscr{P}_{seq}(P)$ and $\mathscr{P}_{inth}(P)$ are always finite.

Proposition 5.39. Given a process $P \in CCS^{\triangle}_{rec}$, the sets $\mathscr{P}_{seq}(P)$ and $\mathscr{P}_{inth}(P)$ are finite.

Proof. The proof is by induction on the structure of P. □

We now define the set $\mathscr{P}_P$ of those processes whose sequential subprocesses and interruption handlers either occur in P or are $\mathbf{0}$ and $\triangle\mathbf{0}$, respectively. The latter is needed to include terms derived from P due to the (modified) semantics of the $\triangle$ operator (see the termination preserving transformation of the semantics at the beginning of Section 5).

https://doi.org/10.1017/S0960129509007683 Published online by Cambridge University Press

Definition 5.40. Let P be a process of $CCS^{\triangle}_{rec}$. We use $\mathscr{P}_P$ to denote the set of $CCS^{\triangle}_{rec}$ processes defined by

$$\mathscr{P}_P \;=\; \{Q \in CCS^{\triangle}_{rec} \mid \mathscr{P}_{seq}(Q) \subseteq \big(\mathscr{P}_{seq}(P) \cup \{\mathbf{0}\}\big) \wedge \mathscr{P}_{inth}(Q) \subseteq \big(\mathscr{P}_{inth}(P) \cup \{\triangle\mathbf{0}\}\big)\}.$$

Note that for every P, the set $\mathscr{P}_P$ is infinite as there are no restrictions on the number of instances of the sequential subprocesses or of the interrupts. Nevertheless, $\preceq$ is a wqo on the set $\mathscr{P}_P$, as shown by the following theorem.

Theorem 5.41. Let $P \in CCS^{\triangle}_{rec}$. The relation $\preceq$ is a wqo over $\mathscr{P}_P$.

Proof. We prove the theorem by showing the existence of a wqo $\sqsubseteq$ such that for every $Q, R \in \mathscr{P}_P$, if $Q \sqsubseteq R$, then $Q \preceq R$ also.

The new relation $\sqsubseteq$ is defined as follows. Given $Q, R \in \mathscr{P}_P$, we define $Q \sqsubseteq R$ if and only if $TREE(Q)\ (=_*)^{tree}\ TREE(R)$ where $(=_*)^{tree}$ is a relation on trees obtained from the identity over $\mathscr{P}_{seq}(P) \cup \mathscr{P}_{inth}(P)$ lifted to sequences according to Definition 5.34, and then lifted to trees according to Definition 5.36. As $\mathscr{P}_{seq}(P) \cup \mathscr{P}_{inth}(P)$ is finite, $=_*$ is a wqo (Proposition 5.38 and Higman's Theorem (Theorem 5.35)). Thus, by Kruskal's Theorem (Theorem 5.37), we have that $(=_*)^{tree}$ is a wqo also.

We still need to show that $\sqsubseteq$ implies $\preceq$. To prove this, we show that given $Q \sqsubseteq R$, the conditions reported in the Definition 5.23 are also satisfied by Q and R, and thus $Q \preceq R$ also.

Consider $Q \sqsubseteq R$. Then there exists an order preserving injective function φ from $dom(TREE(Q))$ to $dom(TREE(R))$ such that for every $\vec{n} \in dom(\varphi)$, we have

$$TREE(Q)(\vec{n}) =_* TREE(R)(\varphi(\vec{n})).$$

There are two possible cases: either $TREE(Q)(\vec{n}) \in \mathscr{P}_{inth}$ or $TREE(Q)(\vec{n}) \in \mathscr{P}^*_{seq}$.

— In the first case, $TREE(Q)(\vec{n}) = TREE(R)(\varphi(\vec{n})) = \triangle S$ for some $S \in \mathscr{P}$ (so the first item of Definition 5.23 holds).

— In the second case, $TREE(Q)(\vec{n}) =_* TREE(R)(\varphi(\vec{n}))$, that is, there exists an injective function f from $\{1, \ldots, len(TREE(Q)(\vec{n}))\}$ to $\{1, \ldots, len(TREE(R)(\varphi(\vec{n})))\}$ such that

$$1 \leqslant f(1) < \ldots < f(len(TREE(Q)(\vec{n}))) \leqslant len(TREE(R)(\varphi(\vec{n})))$$

and

$$TREE(Q)(\vec{n})_i = TREE(R)(\varphi(\vec{n}))_{f(i)}$$

(so the second item of Definition 5.23, where we simply require that f is injective, holds). □

In order to prove that $(CCS^{\triangle}_{rec}, \longmapsto)$ is a well-structured transition system, we simply have to show that for every P the derivatives of P, that is, $Deriv(P)$, is a subset of $\mathscr{P}_P$. This follows from the following proposition.

Proposition 5.42. Let $P \in CCS^{\triangle}_{rec}$ and $Q \in \mathscr{P}_P$. If $Q \overset{\alpha}{\longmapsto} Q'$, then $Q' \in \mathscr{P}_P$.

Proof. The proof is by induction on the proof of transition $Q \overset{\alpha}{\longmapsto} Q'$. □

https://doi.org/10.1017/S0960129509007683 Published online by Cambridge University Press

Corollary 5.43. If $P \in CCS^{\triangle}_{rec}$, then $Deriv(P) \subseteq \mathscr{P}_P$.

We now complete the proof of decidability of termination in $(CCS^{\triangle}_{rec}, \longmapsto)$.

Proof of Theorem 5.26. The fact that $(Deriv(P), \longrightarrow)$ is finitely branching follows from an inspection of the transition rules taking into account the weak guardedness constraint in recursions. Strong compatibility was proved in Theorem 5.33. The fact that $\preceq$ is a wqo on $Deriv(P)$ is a consequence of Corollary 5.43 and Theorem 5.41. □

6. Interpretation of the results

We now discuss the computational power of the calculi considered in this paper. In the proof of the undecidability results we have considered RAMs, which constitute a Turing-complete formalism in the classical setting, that is, given a partial recursive function there is a corresponding RAM program that computes it. This means that whenever the function is defined, the corresponding RAM program is guaranteed to complete its computation yielding the correct value.

We have shown that it is possible to encode deterministically any RAM in $CCS^{\mathtt{tc}}_{rec}$, so we can conclude that the calculus is *Turing complete* according to the following criterion:

— Given a partial recursive function with a given input there is a corresponding process such that:
  – if the function is defined for the input, then all computations of the process terminate and make the corresponding output available (it is sufficient to count the nesting depth of the try-catch operator in the subprocess of the final process that represents the output register);
  – if the function is not defined for the input, then all computations of the process do not terminate.

For the other calculi discussed, the decidability of universal termination allows us to conclude that the calculi are not Turing complete according to the above criterion. Nevertheless, in the proof of the undecidability of existential termination in these calculi, we showed that RAMs can be encoded in such a way that at least the terminating computations respect RAM computations. We can conclude that these calculi satisfy a weaker criterion.

— Given a partial recursive function with a given input, there is a corresponding process such that:
  – if the function is defined for the input, then there exists at least one computation that terminates and, moreover, all computations that terminate make the corresponding output available;
  – if the function is not defined for the input, then all computations of the process do not terminate.

The difference with respect to the above criterion for Turing completeness is in the first item: when the function is defined, the corresponding process may have some computations that do not terminate. We can refer to this weaker criterion for Turing universality as

https://doi.org/10.1017/S0960129509007683 Published online by Cambridge University Press

weak Turing completeness. We can conclude that $CCS^{\mathtt{tc}}_{rec}$ is Turing complete while $CCS^{\triangle}_{!}$, $CCS^{\mathtt{tc}}_{!}$, and $CCS^{\triangle}_{rec}$ are only weakly Turing complete.

We now compare the computational strength of the interrupt and try-catch operators with respect to calculi that lack such operators. In particular, we consider a fragment of CCS (which we will refer to as *GCCS* in the following) that is obtained by removing restriction and relabelling, and by assuming that the operands of a choice are always prefixed processes, as in $\alpha_1.P_1 + \alpha_2.P_2$. It is well known that the processes of *GCCS* can be translated into strongly bisimilar finite Petri-nets (Goltz 1988). As existential and universal termination are decidable for finite Petri-nets (see, for example, Esparza and Nielsen (1994) for a survey of decidable properties in Petri-nets), and because strong bisimilarity preserves both existential and univeral termination (that is, an existentially (respectively, universally) terminating process cannot be strongly bisimilar to a non-existentially (respectively, non-universally) terminating process), we can conclude that both termination problems are decidable in *GCCS* (in fact, it is sufficient to check existential (respectively, universal) termination on the Petri-nets obtained by removing all transitions different from those representing τ actions). Thus, *GCCS* is not weakly Turing complete. As all the encodings of RAMs that we have presented in this paper exploit guarded choice only, we can conclude that if we extend *GCCS* with the interrupt operator we get a weakly Turing complete calculus, and if we extend *GCCS* with try-catch we get Turing completeness. In the light of this last observation, we can conclude that:

— there exists no computable encoding of the interrupt operator in *GCCS* that preserves existential termination;
— there exists no computable encoding of the try-catch operator in *GCCS* that preserves either existential or universal termination;
— there exists no computable encoding of the try-catch operator in *GCCS* extended with the interrupt operator that preserves universal termination.

The last item formalises an interesting impossibility result for the encodability of try-catch into the interrupt operator. As far as the inverse encoding is concerned, one might consider modelling the interrupt operator in terms of try-catch using an encoding like

$$[\![P \triangle Q]\!] = \begin{cases} \mathtt{try}\,(P|\mathtt{throw})\,\mathtt{catch}\,Q & \text{if } Q \text{ can execute at least one action} \\ P & \text{otherwise.} \end{cases}$$

It is easy to see that given a process P including the interrupt operator, the corresponding encoding reproduces all the computations of P. Thus, if P existentially terminates, then its encoding also existentially terminates. However, the encoding does not preserve existential termination because the opposite implication does not hold. For instance, consider the process

$$a.recX.(\tau.X) \mid b.\overline{c} \mid (\overline{a}.\overline{b}) \triangle c,$$

which does not existentially terminate. According to the approach followed by the previous encoding, this process is modelled by

$$a.recX.(\tau.X) \mid b.\overline{c} \mid \mathtt{try}\,(\overline{a}.\overline{b}|\mathtt{throw})\,\mathtt{catch}\,c,$$

https://doi.org/10.1017/S0960129509007683 Published online by Cambridge University Press

which, instead, does existentially terminates when the `throw` action is the first to be executed. We leave the investigation of a faithful encoding of the interrupt operator in terms of try-catch for future work.

7. Conclusion and related work

In this paper we have investigated the impact of the interrupt and the try-catch operators on the decidability of existential and universal termination in fragments of CCS with either replication or recursion. Table 1 in the Introduction summarised all the results proved in this paper, and we discussed the impact of these results in Section 6.

It is worth comparing the results proved in this paper with similar results presented in Busi *et al.* (2003; 2009). In those papers, the interplay between replication/recursion and restriction is studied: a fragment of CCS with restriction and replication is proved to be weakly Turing powerful (according to the criteria presented in the previous section), while the corresponding fragment with recursion is proved to be Turing complete. The main result proved in Busi *et al.* (2003; 2009) is the decidability of universal termination in CCS with replication instead of recursion. In those papers, general replication $!P$ is considered instead of the more constrained guarded replication $!a.P$ considered in the current paper. We can generalise the results we have proved in this paper for $CCS_!^{\triangle}$ and $CCS_!^{\mathtt{tc}}$ by considering general replication instead of guarded replication. The undecidability results hold trivially when we move from guarded to general replication. As far as the decidability results are concerned, we will now show how to modify the proof in Section 5.1 in order to prove the decidability of universal termination in the case of general replication also. Since we use the theory of well-structured transition systems, we need to consider a finitely branching transition system. To do this, we will carry out the proof on the alternative finitely branching transition system obtained by considering the following rules for replication:

$$\frac{P \stackrel{\alpha}{\longrightarrow} P'}{!P \stackrel{\alpha}{\longrightarrow} P' \mid !P} \qquad \frac{P \stackrel{\alpha}{\longrightarrow} P' \quad P \stackrel{\overline{\alpha}}{\longrightarrow} P''}{!P \stackrel{\tau}{\longrightarrow} P' \mid P'' \mid !P}$$

These are equivalent, with respect to universal termination (see the proof in Busi *et al.* (2003; 2009)), to the usual semantics:

$$\frac{P \mid !P \stackrel{\alpha}{\longrightarrow} P'}{!P \stackrel{\alpha}{\longrightarrow} P'}$$

As far as the try-catch operator is concerned, the proof in Section 5.1 applies equally to general replication (defined according to the finitely branching rules) if we replace $!\alpha.P$ by $!P$ in the definition of $d_{\mathtt{tc}}(\_)$ and $Sub(\_)$. As far as the interrupt operator is concerned, we also need to replace all occurrences of $\mathtt{try}\, P\, \mathtt{catch}\, Q$ by $P \triangle Q$.

The interplay between restriction and replication/recursion observed in Busi *et al.* (2003) and Busi *et al.* (2009) is similar to what we have proved in this paper for the interplay between the try-catch operator and replication/recursion. This proves a strong connection

https://doi.org/10.1017/S0960129509007683 Published online by Cambridge University Press

between restriction and try-catch, at least as far as computational power is concerned. Intuitively, this follows from the fact that, like restriction, the try-catch operator defines a new scope for the special `throw` action that is bound to a specific exception handler. On the other hand, the interrupt operator does not have the same computational power. In fact, the calculus with recursion and interrupt is only weakly Turing powerful. This follows from the fact that this operator does not provide a similar binding mechanism between the interrupt signals and the interruptible processes.

It is worth comparing our criterion for the evaluation of the expressive power with the criterion used by Palamidessi in Palamidessi (2003) to distinguish the expressive power of the synchronous and asynchronous π-calculus. Namely, in that paper, it was proved that there exists no modular embedding of the synchronous into the asynchronous π-calculus that preserves any reasonable semantics. When we prove that universal termination (respectively, existential termination) is undecidable in one calculus but not in another, we also prove that there exists no computable encoding (and thus also no modular embedding) of the first calculus into the second that preserves any semantics preserving universal termination (respectively, existential termination).

If we assume that the termination of one computation is observable (as is done, for instance, in process calculi with explicit termination (Baeten *et al.* 2008)), we have that any reasonable semantics (according to the notion of reasonable semantics presented in Palamidessi (2003)) preserves both universal and existential termination.

Finally, we should mention the investigation into the expressive power of the disrupt operator (which is similar to our interrupt operator) carried out by Baeten and Bergstra in the technical report Baeten and Bergstra (2000). They considered a different notion of expressive power: one calculus is more expressive than another if it generates a larger set of transition systems. We consider a stronger notion of expressive power: one calculus is more expressive than another if it supports a more faithful modelling of Turing complete formalisms.

Acknowledgments

We thank the anonymous referees and Catuscia Palamidessi (who is a member of the Editorial Board and followed the publication of this paper) for helpful suggestions that enabled us to improve the presentation. We also thank one of the reviewers for the idea behind the encoding of the interrupt operator in terms of try-catch discussed in Section 6.

References

Baeten, J. C. M., Basten, T. and Reniers, M. A. (2008) *Process algebra (equational theories of communicating processes)*, Cambridge Tracts in Theoretical Computer Science, Cambridge University Press.

Baeten, J. C. M. and Bergstra, J. A. (2000) Mode transfer in process algebra. Report CSR 00-01, Technische Universiteit Eindhoven. (This paper is an expanded and revised version of: Bergstra, J. (2000) A mode transfer operator in process algebra, Report P8808, Programming Research Group, University of Amsterdam, which is available at `http://alexandria.tue.nl/extra1/wskrap/publichtml/200010731.pdf`.)

Bocchi, L., Laneve, C. and Zavattaro, G. (2003) A calculus for long running transactions. In: FMOODS'03: Proceedings of the 6th IFIP International Conference on Formal Methods for Open Object-based Distributed Systems. *Springer-Verlag Lecture Notes in Computer Science* **2884** 124–138.

Boreale, M., Bruni, R., Caires, L., De Nicola, R., Lanese, I., Loreti, M., Martins, F., Montanari, U., Ravara, A., Sangiorgi, D., Vasconcelos, V. T. and Zavattaro, G. (2006) SCC: A Service Centered Calculus. In: WS-FM 2006: Proceedings of the Third International Workshop on Web Services and Formal Methods. *Springer-Verlag Lecture Notes in Computer Science* **4184** 38–57.

Bruni, R., Melgratti, H. C. and Montanari, U. (2004) Nested Commits for Mobile Calculi: Extending Join. In: *TCS 2004: IFIP 18th World Computer Congress, TC1 3rd International Conference on Theoretical Computer Science*, Kluwer 563–576.

Bruni, R., Melgratti, H. C. and Montanari, U. (2005) Theoretical foundations for compensations in flow composition languages. In: *POPL 2005: Proceedings of the 32nd Symposium on Principles of Programming Languages*, ACM Press 209–220.

Busi, N., Gabbrielli, M. and Zavattaro, G. (2003) Replication vs. Recursive Definitions in Channel Based Calculi. In: ICALP'03: Proceedings of 30th International Colloquium on Automata, Languages and Programming. *Springer-Verlag Lecture Notes in Computer Science* **2719** 133–144.

Busi, N., Gabbrielli, M. and Zavattaro, G. (2009) On the Expressive Power of Recursion, Replication and Iteration in Process Calculi. *Mathematical Structures in Computer Science* (to appear). (This is an extended version of Busi *et al.* (2003).)

Butler, M. and Ferreira, C. (2004) An operational semantics for StAC, a language for modelling long-running business transactions. In: COORDINATION'04: Proceedings of the 6th International Conference on Coordination Models and Languages. *Springer-Verlag Lecture Notes in Computer Science* **2949** 87–104.

Butler, M., Hoare, C. A. R. and Ferreira, C. (2003) A trace semantics for long-running transactions. *Springer-Verlag Lecture Notes in Computer Science* **3525** 133–150.

Esparza, J. and Nielsen, M. (1994) Decidability Issues for Petri Nets-a Survey. *Bulletin of the European Association for TCS* **52** 245–262.

Finkel, A. and Schnoebelen, P. (2001) Well-Structured Transition Systems Everywhere! *Theoretical Computer Science* **256** 63–92.

Goltz, U. (1988) On Representing CCS Programs by Finite Petri Nets. In: MFCS'88: Proceedings of Mathematical Foundations of Computer Science. *Springer-Verlag Lecture Notes in Computer Science* **324** 339–350.

Guidi, C., Lanese, I., Montesi, F. and Zavattaro, G. (2008) On the Interplay between Fault Handling and Request-Response Service Invocations. In: *ACSD'08: Proceedings of 8th International Conference on Application of Concurrency to System Design*, IEEE Computer Society press 190–199.

Higman, G. (1952) Ordering by divisibility in abstract algebras. *Proc. London Math. Soc.* **2** 236–366.

Hoare, C. A. R. (1985) *Communicating Sequential Processes*, Prentice-Hall.

Kruskal, J. B. (1960) Well-Quasi-Ordering, The Tree Theorem, and Vazsonyi's Conjecture. *Transactions of the American Mathematical Society* **95** (2) 210–225.

Laneve, C. and Zavattaro, G. (2005) Foundations of web transactions. In: FOSSACS 2005: Proceedings of the 8th International Conference on Foundations of Software Science and Computational Structures. *Springer-Verlag Lecture Notes in Computer Science* **3441** 282–298.

Lapadula, A., Pugliese, R. and Tiezzi, F. (2007) A Calculus for Orchestration of Web Services. In: ESOP 2007: Proceedings of 16th European Symposium on Programming. *Springer-Verlag Lecture Notes in Computer Science* **4421** 33–47.

Milner, R. (1989) *Communication and Concurrency*, Prentice-Hall.
Milner, R., Parrow, J. and Walker, D. (1992) A Calculus of Mobile Processes, Part I + II. *Information and Computation* **100** (1) 1–77.
Minsky, M. L. (1967) *Computation: finite and infinite machines*, Prentice-Hall.
Misra, J. and Cook, W. R. (2007) Computation Orchestration. *Journal of Software and System Modeling* **6** (1) 83–110.
OASIS (2003) WS-BPEL: Web Services Business Process Execution Language Version 2.0. Technical report, OASIS, 2003.
Palamidessi, C. (2003) Comparing the expressive power of the synchronous and asynchronous pi-calculus. *Mathematical Structures in Computer Science* **13** (5) 685–719. (A short version of this paper appeared in POPL'97.)
Shepherdson, J. C. and Sturgis, J. E. (1963) Computability of recursive functions. *Journal of the ACM* **10** 217–255.
Simpson, S. G. (1985) Nonprovability of certain combinatorial properties of finite trees. In: *Harvey Friedman's Research on the Foundations of Mathematics*, North-Holland 87–117.
Vieira, H. T., Caires, L. and Seco, J. C. (2008) The Conversation Calculus: A Model of Service-Oriented Computation. In: ESOP 2008: Proceedings of 17th European Symposium on Programming. *Springer-Verlag Lecture Notes in Computer Science* **4960** 269–283.
W3C (2004) WS-CDL: Web Services Choreography Description Language. Technical report, W3C.