Math. Struct. in Comp. Science (2009), *vol.* 19, *pp.* 1191–1222. © Cambridge University Press 2009
doi:10.1017/S096012950999017X

On the expressive power of recursion, replication and iteration in process calculi

NADIA BUSI, MAURIZIO GABBRIELLI and
GIANLUIGI ZAVATTARO

Dip. di Scienze dell'Informazione, Univ. di Bologna, Mura A.Zamboni 7, 40127 Bologna, Italy
Email: {gabbri;zavattar}@cs.unibo.it

Received 23 June 2008; revised 17 July 2009

In memory of Nadia Busi

In this paper we investigate the expressive power of three alternative approaches to the definition of infinite behaviours in process calculi, namely, recursive definitions, replication and iteration. We prove several results discriminating between the calculi obtained from a core CCS by adding the three mechanisms mentioned above. These results are derived by considering the decidability of four basic properties: termination (that is, all computations are finite); convergence (that is, the existence of a finite computation); barb (that is, the ability to perform an action on a given channel) and weak bisimulation.
Our results, which are summarised in Table 1, show that the three calculi form a strict expressiveness hierarchy in that: all the properties mentioned are undecidable in CCS with recursion; only termination and barb are decidable in CCS with replication; all the properties are decidable in CCS with iteration.
As a corollary, we also obtain a strict expressiveness hierarchy with respect to weak bisimulation, since there exist weak bisimulation preserving encodings of iteration in replication and of replication in recursion, whereas there are no weak bisimulation preserving encodings in the other directions.

1. Introduction

Process calculi are continuously being updated with new languages, dialects and variants, which range from general frameworks such as CCS and π-calculus to formalisms tailored to a specific class of applications (biological ones are probably the most common currently). Given such a rich variety, it is important to make a formal comparison between the existing languages in order to understand precisely their relative expressive power. Several notions of expressive power are meaningful in this context: the classical notion based on the ability to express recursive functions can be further refined by considering, for example, compositionality properties for the encoding of one language in another, or the ability to express certain patterns of behaviours (which are typically connected to mobility).

An important aspect that, in general, may significantly affect the expressive power of a language is the mechanism adopted for extending finite processes to express infinite

https://doi.org/10.1017/S096012950999017X Published online by Cambridge University Press

behaviours: three classical mechanisms used in process calculi are *recursion*, *replication* and *iteration*. Recursion can be supported through the use of process constants: each process constant D has an associated (possibly recursive) definition $D \stackrel{def}{=} P$. Recursively defined process constants allow a 'depthwise' infinite behaviour, since process copies in this case can be nested to an arbitrary depth through constant application. On the other hand, the replication operator $!P$ allows one to create an unbounded number of parallel copies of a process P, thus providing a 'widthwise' infinite behaviour, since the copies are placed at the same level. Finally, the iteration operator P^* allows us to iterate the execution of a process P, that is, at the end of the execution of one copy of P another copy can be activated. In this case, a 'repetitive' infinite behaviour is supported, since the copies are executed one after the other.

It is well known that recursion and replication are equivalent (for any reasonable notion of expressive power) for full π-calculus (Milner *et al.* 1992), as the ability to communicate free names together with replication and restriction allows us to simulate process constants (replication can easily be simulated by recursive definitions, provided one admits enough constants). On the other hand, there is common agreement on the fact that recursion cannot be replaced by replication when no name mobility is allowed, as in the case of CCS, even though this result has not yet been formally proved.

In this paper we compare the expressive power of recursion, replication and iteration in the context of CCS. More precisely, we consider the three dialects CCS_D, $\text{CCS}_!$ and CCS_* obtained by adding recursion, replication and iteration, respectively, to a core CCS language corresponding to the finite fragment of CCS (Milner 1989) without relabelling. For these three dialects, we investigate the decidability of the following four relevant properties of processes: *termination*, that is, the non-existence of a divergent computation (or, equivalently, that all the computations are finite); *convergence*, that is, the existence of a computation that terminates; *barb*, that is, the ability to perform a synchronisation on a certain channel after a (possibly empty) internal computation; *weak bisimulation* (here we say that weak bisimulation is decidable if, given any pair of processes, the question as to whether they are weakly bisimilar to each other is decidable).

We will first prove that all these properties are undecidable in CCS_D. To do this, we make use of Random Access Machines (Shepherdson and Sturgis 1963), RAMs for short, which are a well-known deterministic Turing powerful formalism. The undecidability results are obtained by providing an encoding of RAMs in CCS_D. The encoding is deterministic, that is, it presents a unique possible computation that reflects the computation of the corresponding RAM. This proves immediately that termination and convergence are undecidable. By exploiting a slightly different (deterministic) encoding, we prove the undecidability of both barb and weak bisimulation: the idea is to extend the modelling of RAMs with an observable action that can be performed on program termination; in this way we reduce the problem of testing the termination of a RAM to the problem of detecting an observable behaviour.

Next we show that convergence and weak bisimulation are also undecidable in the language $\text{CCS}_!$, which includes replication. These results are obtained by providing a non-deterministic encoding of RAMs in $\text{CCS}_!$. The encoding is non-deterministic in the sense that computations that do not follow the expected behaviour of the modelled

https://doi.org/10.1017/S096012950999017X Published online by Cambridge University Press

	Recursion	Replication	Iteration
Termination	undecidable	decidable	decidable
Convergence	undecidable	undecidable	decidable
Barb	undecidable	decidable	decidable
Weak bisimulation	undecidable	undecidable	decidable

Table 1. *Summary of the results*

RAM can be introduced by the encoding, however, all these computations are infinite. This proves that a process modelling a RAM has a terminating computation, that is, converges, if and only if the corresponding RAM terminates. Thus, process convergence is undecidable for the calculus with replication. The non-deterministic modelling of RAMs under replication enables us to prove that weak bisimulation is undecidable also, using a technique similar to that described above for the calculus with recursion. Interestingly, while convergence and weak bisimulation are undecidable under replication, we prove that both termination and barb turn out to be decidable properties. These decidability results are obtained by resorting to the theory of well-structured transition systems (Finkel and Schnoebelen 2001). It is also worth noting that the decidability of process termination for the calculus with replication implies the impossibility of providing a termination preserving encoding of RAMs in $CCS_!$.

For the calculus with process iteration we show that all the properties are decidable. This is a consequence of the fact that the processes of this calculus are finite state. Intuitively, this follows from the fact that each iteration activates one copy at a time (thus only a predefined number of processes can be active at the same time) and all the copies share the same finite set of possible states.

The results of our investigation are summarised in Table 1.

As a final corollary, we also show that recursion, replication and iteration constitute a strict expressiveness hierarchy with respect to weak bisimulation. In fact, we show that there exist weak bisimulation preserving encodings of iteration in replication and of replication in recursion, but that there are no weak bisimulation preserving encodings in the other directions.

The rest of this paper is structured as follows. In Section 2 we present the syntax and semantics of the calculi under consideration. In Section 3 we give the deterministic RAM encoding and the undecidability results for the calculus with recursion. In Section 4, we first give the non-deterministic RAM encoding for the calculus with replication and prove the undecidability results. Then we prove the decidability of termination and barb for $CCS_!$. In Section 5, we prove that the processes of the calculus with process iteration are finite states. In Section 6 we present conclusions, discuss the weak-bisimulation preserving encoding and mention some relevant related work.

Preliminary versions of this paper appeared as Busi *et al.* (2003; 2004).

https://doi.org/10.1017/S096012950999017X Published online by Cambridge University Press

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

$$\texttt{RES}: \quad \frac{P \xrightarrow{\alpha} P'}{(\nu x)P \xrightarrow{\alpha} (\nu x)P'} \; x \notin n(\alpha) \qquad \texttt{COM}: \quad \frac{P \xrightarrow{\alpha} P' \quad Q \xrightarrow{\overline{\alpha}} Q'}{P|Q \xrightarrow{\tau} P'|Q'}$$

Table 2. *The transition system for finite core CCS (the symmetric rules for* PAR, SUM *and* COM *have been omitted to save space).*

2. The calculi

We start by considering the finite fragment of the core of CCS (which we will sometimes simply call CCS for brevity). We will then present the three infinite extensions.

Definition 1 (finite core CCS). Let *Name*, ranged over by $x, y, \ldots$, be a denumerable set of channel names. The class of finite core CCS processes is described by the following grammar:

$$\begin{array}{lcl} P & ::= & \mathbf{0} \mid \alpha.P \mid P+P \mid P|P \mid (\nu x)P \\ \alpha & ::= & \tau \mid x \mid \overline{x}. \end{array}$$

The term $\mathbf{0}$ denotes the empty process. 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 synchronising action are available, the output $\overline{x}$ or the input x. The sum construct $+$ is used to create a choice between the summands, and parallel composition $|$ is used to run parallel programs. Restriction $(\nu x)P$ makes the name x local in P. We denote the process $\alpha.\mathbf{0}$ simply with α, and the process $(\nu x_1)(\nu x_2)\ldots(\nu x_n)P$ with $(\nu \tilde{x})P$ where $\tilde{x}$ is the sequence of names $x_1, x_2, \ldots, x_n$.

For input and output actions, we write $\overline{\alpha}$ for the complementary of α; that is, if $\alpha = x$, then $\overline{\alpha} = \overline{x}$, and if $\alpha = \overline{x}$, then $\overline{\alpha} = x$. We write $fn(P)$ and $bn(P)$ for the *free names* and *bound names* of P, respectively. The *names* of P, written $n(P)$, is the union of the free and bound names of 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 presents the set of transition rules for finite CCS.

Definition 2 (CCS_D). We assume a denumerable set of constants, ranged over by D. The class of CCS_D processes is defined by adding the production $P ::= D$ to the grammar of Definition 1. It is assumed that each constant D has a unique defining equation of the form $D \stackrel{def}{=} P$.

https://doi.org/10.1017/S096012950999017X Published online by Cambridge University Press

$$\mathbf{0} \xrightarrow{\surd} \qquad P^* \xrightarrow{\surd} \qquad \frac{P \xrightarrow{\surd}}{(\nu x)P \xrightarrow{\surd}}$$

$$\frac{P \xrightarrow{\surd} \quad Q \xrightarrow{\surd}}{P|Q \xrightarrow{\surd}} \qquad \frac{P \xrightarrow{\surd} \quad Q \xrightarrow{\surd}}{P + Q \xrightarrow{\surd}} \qquad \frac{P \xrightarrow{\surd} \quad Q \xrightarrow{\surd}}{P;Q \xrightarrow{\surd}}$$

Table 3. *Definition of the termination predicate.*

The transition rule for constant is

$$\texttt{CONST}: \quad \frac{P \xrightarrow{\alpha} P' \qquad D \stackrel{def}{=} P}{D \xrightarrow{\alpha} P'}$$

It is worth noting that this rule for the semantics of constants was also adopted in Giambiagi *et al.* (2004). In that paper, it was observed that no α-conversion is to be considered in calculi with this form of constant definition, and this causes name captures and scoping to be dynamic.

Definition 3 (CCS$_!$). The class of CCS$_!$ processes is defined by adding the production $P ::= !P$ to the grammar of Definition 1.

The transition rule for replication is

$$\texttt{REPL}: \quad \frac{P \mid !P \xrightarrow{\alpha} P'}{!P \xrightarrow{\alpha} P'}$$

Definition 4 (CCS$_*$). The class of CCS$_*$ processes is defined by adding the production $P ::= P^*$ to the grammar of Definition 1.

Intuitively, the process P^* has the ability to iterate the behaviour of the process P an arbitrary (possibly zero) number of times. In order to describe the semantics of iteration formally, we explicitly represent the ending of process P using the predicate $P \xrightarrow{\surd}$. We also exploit an auxiliary operator $P;Q$ denoting the sequential composition of processes. Informally, given the process $P;Q$, the process Q can start only if $P \xrightarrow{\surd}$. Formally, the axioms and rules defining the predicate for $P \xrightarrow{\surd}$ are given in Table 3. The transition

https://doi.org/10.1017/S096012950999017X Published online by Cambridge University Press

rules for iteration are

$$\texttt{ITER}: \quad \frac{P \xrightarrow{\alpha} P'}{P^* \xrightarrow{\alpha} P'; P^*}$$

$$\texttt{SEQ1}: \quad \frac{P \xrightarrow{\alpha} P'}{P; Q \xrightarrow{\alpha} P'; Q} \qquad \texttt{SEQ2}: \quad \frac{P \xrightarrow{\surd} \quad Q \xrightarrow{\alpha} Q'}{P; Q \xrightarrow{\alpha} Q'}$$

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

Given a process Q, its internal runs $Q \longrightarrow Q_1 \longrightarrow Q_2 \longrightarrow \ldots$ are given by the sequences of τ labelled transitions, that is, those transitions that the process can perform without requiring interaction with the context. Formally, we have $P \longrightarrow P'$ if and only if $P \xrightarrow{\tau} P'$. We use $\longrightarrow^*$ to denote the reflexive and transitive closure of $\longrightarrow$. We use $Deriv(P)$ to denote the set of processes reachable from P with a sequence of reduction steps, formally $Deriv(P) = \{Q \mid P \longrightarrow^* Q\}$.

A process Q is *dead* if there exists no Q' such that $Q \longrightarrow Q'$. We say that a process P *converges* if there exists a dead process P' in $Deriv(P)$. We say that P *terminates* if all its internal runs terminate, that is, the process P cannot give rise to an infinite computation: formally, P *terminates* if and only if there exist no $\{P_i\}_{i \in \mathbb{N}}$ such that $P_0 = P$ and $P_j \longrightarrow P_{j+1}$ for every j. Note that *process termination* implies *process convergence* but the converse does not hold.

Barbs are used to observe whether a process has the ability to perform, possibly after an internal run, an observable action on a specific channel; formally $P \Downarrow x$ if and only if there exist P' and P'' such that $P \longrightarrow^* P' \xrightarrow{\alpha} P''$ and $n(\alpha) = \{x\}$.

Definition 5 (weak bisimulation). A binary, symmetric relation $\mathscr{R}$ on processes is a *weak bisimulation* if $(P, Q) \in \mathscr{R}$ implies that, if $P \xrightarrow{\alpha} P'$, then one of the following holds:

— there exist Q', Q'', Q''' such that $Q \longrightarrow^* Q' \xrightarrow{\alpha} Q'' \longrightarrow^* Q'''$ and $(P', Q''') \in \mathscr{R}$;

— $\alpha = \tau$ and there exists Q' such that $Q \longrightarrow^* Q'$ and $(P', Q') \in \mathscr{R}$.

Two processes P and Q are *weakly bisimilar*, written $P \approx Q$, if there exists a weak bisimulation $\mathscr{R}$ such that $(P, Q) \in \mathscr{R}$.

3. Results for CCS_D

A RAM (Shepherdson and Sturgis 1963), denoted by R in the following, is a computational model composed of a finite set of registers $r_1, \ldots, r_n$, which can hold arbitrarily large natural numbers, and a program consisting of indexed instructions $(1 : I_1), \ldots, (m : I_m)$, that is, a sequence of simple numbered instructions, like 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 $c_1, \ldots, c_n$ are the current contents of the registers $r_1, \ldots, r_n$, respectively.

https://doi.org/10.1017/S096012950999017X Published online by Cambridge University Press

Without loss of generality, we assume that the registers contain the value 0 at the beginning of the computation and that the execution of the program begins with the first instruction $(1 : I_1)$. In other words, the initial configuration is $(1, 0, \ldots, 0)$. The computation continues by executing the other 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. 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.

Minsky (1967) showed that the following two instructions are sufficient to model every recursive function:

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

3.1. *Undecidability of convergence and termination*

In the remainder of this section we will reason up to the structural congruence $\equiv_R$, which is used to remove terminated processes equal to $\mathbf{0}$ as well as unnecessary restrictions. Formally, $\equiv_R$ is the least congruence relation satisfying the axioms

$$P|\mathbf{0} \equiv_R P$$
$$(\nu x)P \equiv_R P \quad \text{if } x \notin fn(P).$$

The following proposition shows that processes congruent according to $\equiv_R$ have the same operational semantics, which allows us to reason up to this structural congruence.

Proposition 1. Let $P, Q \in \mathrm{CCS}_D$ with $P \equiv_R Q$. If $P \xrightarrow{\alpha} P'$, there exists Q' such that $Q \xrightarrow{\alpha} Q'$ and $P' \equiv_R Q'$.

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

We now show how to reduce RAM termination to either convergence or termination in CCS_D. Let R be a RAM with registers $r_1, \ldots, r_n$, and instructions $(1 : I_1), \ldots, (m : I_m)$. We model R as follows. For each $1 \leqslant i \leqslant m$, we model the ith instruction $(i : I_i)$ of R using a program constant $Inst_i$ defined as follows:

$$Inst_i \stackrel{def}{=} \overline{inc_j}.Inst_{i+1} \qquad \text{if } I_i = Succ(r_j)$$
$$Inst_i \stackrel{def}{=} \overline{dec_j}.ack.Inst_{i+1} + \overline{zero_j}.Inst_s \qquad \text{if } I_i = DecJump(r_j, s).$$

In the first case, the process $Inst_i$ simply increments the register r_j (by firing the $\overline{inc_j}$ prefix) and activates the subsequent instruction. In the second case, the process $Inst_i$ tries either to decrement or to test whether the register r_j is empty. Depending on the prefix fired ($\overline{dec_j}$ or $\overline{zero_j}$), the corresponding next instruction is activated ($Inst_{i+1}$ or $Inst_s$, respectively). In the case of decrement, the process waits for an acknowledgment *ack* before activating the next instruction; this is necessary in order that the next instruction is activated only after the register has actually been updated.

https://doi.org/10.1017/S096012950999017X Published online by Cambridge University Press

In order to model RAM termination, we also need to take under consideration the instruction indexes outside $\{1,\ldots,m\}$ that can be reached as a subsequent instruction of one of the defined instructions $(1 : I_1),\ldots,(m : I_m)$. For each of these indexes i, we consider a corresponding constant defined as follows:

$$Inst_i \stackrel{def}{=} \mathbf{0} \qquad \text{if } I_i \text{ undefined (that is, } i \notin \{1,\ldots,m\}).$$

As far as the modelling of the registers is concerned, we represent each register r_j, which is initially empty, using a constant Z_j. The constant Z_j is defined in terms of two other constants O_j and E_j:

$$\begin{aligned} Z_j &\stackrel{def}{=} zero_j.Z_j + inc_j.(\nu x)(O_j \mid x.\overline{ack}.Z_j) \\ O_j &\stackrel{def}{=} dec_j.\overline{x} + inc_j.(\nu y)(E_j \mid y.\overline{ack}.O_j) \\ E_j &\stackrel{def}{=} dec_j.\overline{y} + inc_j.(\nu x)(O_j \mid x.\overline{ack}.E_j). \end{aligned}$$

The idea behind this modelling of the registers is to exploit a chain of nested restrictions with a length corresponding to the content of the register. More precisely, the term Z_j represents the register when empty, while O_j and E_j model the register when its contents are either odd or even, respectively. Each time the register is incremented, the length of the chain of restrictions augments due to the creation of a new name. Observe that in order to avoid name collisions, the two names x and y are used alternately. The use of two different names also requires the use of the two different constants O_j and E_j.

Example 1. Consider a RAM with two registers r_1 and r_2 and the following program:

$$\begin{aligned} &(1 : Succ(r_1)) \\ &(2 : Succ(r_1)) \\ &(3 : DecJump(r_1, 5)) \\ &(4 : DecJump(r_2, 3)). \end{aligned}$$

Assuming that the registers are initially both empty, and that the computation starts from the instruction with index 1, the computation of the RAM above consists of two increments of r_1, followed by a first decrement of r_1, a jump from instruction 4 to 3, a second decrement of r_1, a jump from instruction 4 to 3, and the final jump from instruction 3 to the undefined instruction 5. As we will express formally below, we will model the above RAM using the process

$$P = Inst_1 \mid Z_1 \mid Z_2.$$

We have that P has the following 2 deterministic reduction steps corresponding to the two increment instructions:

$$\begin{aligned} P &\longrightarrow Inst_2 \mid (\nu x)(O_1 \mid x.\overline{ack}.Z_1) \mid Z_2 \\ &\longrightarrow Inst_3 \mid (\nu x)((\nu y)(E_1 \mid y.\overline{ack}.O_1) \mid x.\overline{ack}.Z_1) \mid Z_2 = Q. \end{aligned}$$

Note that at this point of the computation the first register contains the value 2, which is represented by the nesting of the two restrictions on the names x and y. The computation continues with the following 3 deterministic reduction steps corresponding to the first

https://doi.org/10.1017/S096012950999017X Published online by Cambridge University Press

decrement:

$$\begin{aligned} Q &\longrightarrow ack.Inst_4 \mid (vx)((vy)(\overline{y} \mid y.\overline{ack}.O_1) \mid x.\overline{ack}.Z_1) \mid Z_2 \\ &\longrightarrow ack.Inst_4 \mid (vx)((vy)(\mathbf{0} \mid \overline{ack}.O_1) \mid x.\overline{ack}.Z_1) \mid Z_2 \\ &\longrightarrow Inst_4 \mid (vx)((vy)(\mathbf{0} \mid O_1) \mid x.\overline{ack}.Z_1) \mid Z_2 \\ &\equiv_R Inst_4 \mid (vx)(O_1 \mid x.\overline{ack}.Z_1) \mid Z_2 \;=\; R. \end{aligned}$$

Note that at this point the first register contains the value 1, which is represented by the fact that the inner restriction on y can be removed by the structural congruence. The computation is then completed by the following 6 deterministic reduction steps:

$$\begin{aligned} R &\longrightarrow Inst_3 \mid (vx)(O_1 \mid x.\overline{ack}.Z_1) \mid Z_2 \\ &\longrightarrow ack.Inst_4 \mid (vx)(\overline{x} \mid x.\overline{ack}.Z_1) \mid Z_2 \\ &\longrightarrow ack.Inst_4 \mid (vx)(\mathbf{0} \mid \overline{ack}.Z_1) \mid Z_2 \\ &\longrightarrow Inst_4 \mid (vx)(\mathbf{0} \mid Z_1) \mid Z_2 \;\equiv_R Inst_4 \mid Z_1 \mid Z_2 \\ &\longrightarrow Inst_3 \mid Z_1 \mid Z_2 \\ &\longrightarrow Inst_5 \mid Z_1 \mid Z_2 \end{aligned}$$

where the last process is dead because $Inst_5 \stackrel{def}{=} \mathbf{0}$.

We are now ready to give a formal definition of the process $R_j^{c_j}$ used to model the register r_j when its content is c_j:

$$R_j^{c_j} \stackrel{def}{=} Z_j \qquad \text{if } c_j = 0$$

$$R_j^{c_j} \stackrel{def}{=} \underbrace{(vx)\big((vy)\big(\ldots(vx)}_{c_j \text{ restrictions}} \big(O_j \mid \underbrace{x.\overline{ack}.E_j\big)\ldots\big) \mid y.\overline{ack}.O_j\big) \mid x.\overline{ack}.Z_j\big)}_{c_j \text{ processes}} \quad \text{if } c_j > 0 \text{ is odd}$$

$$R_j^{c_j} \stackrel{def}{=} \underbrace{(vx)\big((vy)\big(\ldots(vy)}_{c_j \text{ restrictions}} \big(E_j \mid \underbrace{y.\overline{ack}.O_j\big)\ldots \mid y.\overline{ack}.O_j\big) \mid x.\overline{ack}.Z_j\big)}_{c_j \text{ processes}} \quad \text{if } c_j > 0 \text{ is even.}$$

Definition 6. 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 $(i, c_1, \ldots, c_n)$ of R, we define

$$[\![(i, c_1, \ldots, c_n)]\!]_D \;=\; Inst_i \mid R_1^{c_1} \mid \ldots \mid R_n^{c_n}$$

with $Inst_i$ and $R_j^{c_j}$ defined as above.

As we have already discussed, the computation carried out by the encoding of a RAM proceeds deterministically, and corresponds exactly to the computation of the RAM. Hence the encoding terminates if and only if the RAM under consideration terminates, as stated by the following theorem. The remainder of this subsection is dedicated to the formal proof of this correctness result.

We say that a process P performs a deterministic reduction step if it is not dead and if $P \longrightarrow P'$ and $P \longrightarrow P''$ then $P' \equiv_R P''$. A process performs a deterministic computation if all of its derivatives are either dead or perform a deterministic computation step.

We are now ready to state the correspondence result between computations carried out by a RAM and those carried out by its encoding.

https://doi.org/10.1017/S096012950999017X Published online by Cambridge University Press

Proposition 2. 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 $(i, c_1, \ldots, c_n)$, one of the following holds:

— $i \notin \{1, \ldots, m\}$ and $[\![(i, c_1, \ldots, c_n)]\!]_D$ is dead.
— $(i, c_1, \ldots, c_n) \rightarrow_R (i', c'_1, \ldots, c'_n)$ and $[\![(i, c_1, \ldots, c_n)]\!]_D$ perform either one or three deterministic reduction steps reaching a process $Q \equiv_R [\![(i', c'_1, \ldots, c'_n)]\!]_D$.

Proof. It is easy to see that $[\![(i, c_1, \ldots, c_n)]\!]_D$ is dead if and only if the instruction of index i is undefined (that is, $i \notin \{1, \ldots, m\}$), so the first item holds.

If the instruction with index i is defined, there exists $(i', c'_1, \ldots, c'_n)$ such that

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

This RAM computational step is an increment, a decrement or a test-for-zero action. In the case of an increment or test-for-zero, $[\![(i, c_1, \ldots, c_n)]\!]_D$ has a deterministic reduction step leading to $[\![(i', c'_1, \ldots, c'_n)]\!]_D$. In the case of a decrement, $[\![(i, c_1, \ldots, c_n)]\!]_D$ performs three deterministic steps before reaching $[\![(i', c'_1, \ldots, c'_n)]\!]_D$. So the second item holds. □

As a simple corollary of the above Proposition, we have that convergence is undecidable since a RAM terminates if and only if the corresponding encoding has a terminating computation.

Theorem 1. Let R be a RAM with program instructions $(1 : I_1), \ldots, (m : I_m)$ and registers $r_1, \ldots, r_n$ with the initial configuration $(1, 0, \ldots, 0)$. Then R terminates if and only if the process $[\![(1, 0, \ldots, 0)]\!]_D$ converges.

Proof. As a consequence of Proposition 2, we have

$$(1, 0, \ldots, 0) \rightarrow_R (i_1, c_1^1, \ldots, c_n^1) \rightarrow_R \ldots \rightarrow_R (i_k, c_1^k, \ldots, c_n^k)$$

if and only if

$$[\![(1, 0, \ldots, 0)]\!]_D \longrightarrow^+ [\![(i_1, c_1^1, \ldots, c_n^1)]\!]_D \longrightarrow^+ \ldots \longrightarrow^+ [\![(i_k, c_1^k, \ldots, c_n^k)]\!]_D.$$

This can be proved by induction on the length k of the computation.

Assume now that the RAM terminates. In this case we have

$$(1, 0, \ldots, 0) \rightarrow_R (i_1, c_1^1, \ldots, c_n^1) \rightarrow_R \ldots \rightarrow_R (i_k, c_1^k, \ldots, c_n^k)$$

with the instruction of index i_k undefined. By the first item of Proposition 2, we have $[\![(i_k, c_1^k, \ldots, c_n^k)]\!]_D$ is dead, so $[\![(1, 0, \ldots, 0)]\!]_D$ converges due to the computation

$$[\![(1, 0, \ldots, 0)]\!]_D \longrightarrow^+ [\![(i_1, c_1^1, \ldots, c_n^1)]\!]_D \longrightarrow^+ \ldots \longrightarrow^+ [\![(i_k, c_1^k, \ldots, c_n^k)]\!]_D.$$

We now assume that the RAM does not terminate, and prove by contradiction that $[\![(1, 0, \ldots, 0)]\!]_D$ does not converge. Therefore, in order to show the contradiction, we assume that $[\![(1, 0, \ldots, 0)]\!]_D$ does converge. Hence

$$[\![(1, 0, \ldots, 0)]\!]_D \longrightarrow^+ [\![(i_1, c_1^1, \ldots, c_n^1)]\!]_D \longrightarrow^+ \ldots \longrightarrow^+ [\![(i_k, c_1^k, \ldots, c_n^k)]\!]_D$$

https://doi.org/10.1017/S096012950999017X Published online by Cambridge University Press

with the instruction of index i_k undefined. By the first item of Proposition 2, the RAM under consideration terminates due to the computation

$$(1, 0, \ldots, 0) \rightarrow_R (i_1, c_1^1, \ldots, c_n^1) \rightarrow_R \ldots \rightarrow_R (i_k, c_1^k, \ldots, c_n^k).$$

This contradicts our initial assumption that the RAM does not terminate. □

3.2. *Undecidability of barb and weak bisimulation*

In this section we slightly modify the above modelling of RAMs in order to prove that barb and weak bisimulation are undecidable. We use essentially the same modelling, but, in order to keep the internal actions unobservable, we add outer restrictions on all the names used by the encoding, and we add a unique observable action $\overline{w}$ to communicate termination. In this way, we have that the modelling of a RAM R is weakly bisimilar to $\overline{w}$ (respectively, has a barb on the channel w) if and only if R terminates, thus proving that weak bisimulation (respectively, barb) is undecidable.

All the definitions of the encoding are as in the previous subsection apart from the definition of undefined instructions, which is now as follows:

$$Inst_i \stackrel{def}{=} \overline{w} \qquad \text{if } I_i \text{ undefined (that is, } i \notin \{1, \ldots, m\}).$$

Recall that instructions outside the range $\{1, \ldots, m\}$ can be reached only on program termination.

We can now prove the undecidability of weak barb.

Theorem 2. Let R be a RAM with program instructions $(1 : I_1), \ldots, (m : I_m)$ and registers $r_1, \ldots, r_n$. We have that R terminates if and only if $[\![(1, 0, \ldots, 0)]\!]_D \Downarrow w$.

Proof. As already discussed in the previous subsection, the computation carried out by the encoding of a RAM proceeds deterministically and corresponds exactly to the computation carried out by the RAM itself.

Assume now that the RAM R terminates. In this case the encoding has a deterministic internal run leading to a process that contains the encoding of an undefined instruction $Inst_i$, that is, the process $\overline{w}$. We can conclude that the encoding has a barb on w.

Assume now that the RAM R does not terminate. In this case the encoding has an infinite deterministic internal run, thus no process $Inst_i$, with corresponding instruction I_i undefined, can be produced. As only such processes can perform actions on the channel w, we can conclude that the encoding has no barb on w. □

The above theorem proves that barb is undecidable in CCS_D. As a consequence of this theorem, and as a consequence of the fact that the encoding has only one deterministic internal run, we also have that weak bisimulation is undecidable. In fact, it is sufficient to add outer restrictions to the encoding on all the names used by the encoding except w in order to obtain a process that is weakly equivalent to $\overline{w}$ if and only if the encoding has a barb on w. Formally, we have

$$[\![(1, 0, \ldots, 0)]\!]_D \Downarrow w$$
$$\text{if and only if}$$
$$(\nu\ inc_1, dec_1, zero_1, \ldots, inc_n, dec_n, zero_n, ack)\big([\![(1, 0, \ldots, 0)]\!]_D\big) \approx \overline{w}.$$

Hence, weak bisimulation is undecidable too.

https://doi.org/10.1017/S096012950999017X Published online by Cambridge University Press

4. Results for CCS$_!$

4.1. *Undecidability of convergence and weak bisimulation*

We prove that CCS$_!$ is powerful enough to model, at least in a non-deterministic way, any Random Access Machine. 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 modelling has a terminating computation if and only if the RAM terminates. This proves that *convergence* is undecidable.

Exploiting the encoding, we also prove that weak bisimulation is undecidable. The idea is to use just two observable actions, namely $\overline{w}$ and $\overline{w'}$. The former makes visible the fact that the program counter has reached an index outside the original range $\{1,\ldots,m\}$ of program instructions. The latter makes visible the activation of an incorrect infinite computation. In this way, we have that a correct terminating run of the encoding includes one and only one observable action $\overline{w}$, while all incorrect computations also include the other observable action $\overline{w'}$. Thus, if P is the encoding of a RAM R, then R terminates if and only if $P \approx \tau.P + \tau.\overline{w}$. This proves that *weak bisimulation* is undecidable.

In this section we reason up to an extended version of the structural congruence $\equiv_R$ that allows also for the reordering of processes in parallel compositions. The new definition also includes the axioms

$$P|Q \equiv_R Q|P \qquad\qquad P|(Q|R) \equiv_R (P|Q)|R.$$

Just as we proved that the previous version of $\equiv_R$ preserves the operational semantics of CCS$_D$, this new extended version of $\equiv_R$ preserves the operational semantics of the calculus with replication.

Proposition 3. Let $P, Q \in$ CCS$_!$ with $P \equiv_R Q$. If $P \xrightarrow{\alpha} P'$, there exists Q' such that $Q \xrightarrow{\alpha} Q'$ and $P' \equiv_R Q'$.

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

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 using a process $\overline{p}_i$ 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 using a replicated process, which is guarded by an input operation p_i. Once activated, the instruction performs its operation on the registers, then waits for an acknowledgment indicating that the operation has been performed, and then, finally, 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:

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

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

https://doi.org/10.1017/S096012950999017X Published online by Cambridge University Press

It is worth noting that a program counter message $\overline{p_i}$, with the index i outside the range $\{1,\ldots,m\}$, is produced on program termination. It is not restrictive to assume that this is the unique index $m+1$. For this index $m+1$, we assume the presence of a process $p_{m+1}.\overline{w}$ that can make termination observable on the channel w.

We model each register r_j, when it contains c_j, using the following process (which we will simply denote using $[\![r_j = c_j]\!]$ in the following):

$$\begin{array}{rcl}
[\![r_j = c_j]\!] & : & (\nu\ m,u)\ \big(\\
& & \quad \prod_{c_j} \overline{u} \mid \\
& & \quad inc_j.(\overline{m} \mid \overline{u}) + dec_j.(u.\overline{m} + \overline{jmp}.(u.DIV \mid \overline{nr_j})) \mid \\
& & \quad !m.(\overline{ack} \mid inc_j.(\overline{m} \mid \overline{u}) + dec_j.(u.\overline{m} + \overline{jmp}.(u.DIV \mid \overline{nr_j}))) \big) \mid \\
& & !nr_j.(\nu\ m,u)\ \big(\\
& & \quad \overline{m} \mid !m.(\overline{ack} \mid inc_j.(\overline{m} \mid \overline{u}) + dec_j.(u.\overline{m} + \overline{jmp}.(u.DIV \mid \overline{nr_j}))) \big)
\end{array}$$

where DIV is a process able to activate the following infinite observable computation:

$$DIV \ : \quad \overline{w'} \mid !w'.\overline{w'}.$$

Example 2. We consider a RAM with only one register r_j, which is initially empty, and the following program instructions:

$$\begin{array}{l}
(1 : Succ(r_1)) \\
(2 : DecJump(r_1, 2)).
\end{array}$$

The RAM computation simply consists of an increment followed by a decrement. The encoding of this RAM includes the encoding of the program counter, instructions and register:

$$\begin{array}{rcl}
P = & \overline{p_1} \mid !p_1.(\overline{inc_1} | ack.\overline{p_2}) \mid !p_2.(\overline{dec_1} \mid (ack.\overline{p_3} \ + \ jmp.ack.\overline{p_2})) \mid \\
& (\nu\ m,u)\ \big(\\
& \quad inc_1.(\overline{m} \mid \overline{u}) + dec_1.(u.\overline{m} + \overline{jmp}.(u.DIV \mid \overline{nr_1})) \mid \\
& \quad !m.(\overline{ack} \mid inc_1.(\overline{m} \mid \overline{u}) + dec_1.(u.\overline{m} + \overline{jmp}.(u.DIV \mid \overline{nr_1}))) \big) \mid \\
& !nr_1.(\nu\ m,u)\ \big(\\
& \quad \overline{m} \mid !m.(\overline{ack} \mid inc_1.(\overline{m} \mid \overline{u}) + dec_1.(u.\overline{m} + \overline{jmp}.(u.DIV \mid \overline{nr_1}))) \big).
\end{array}$$

This process has four deterministic reduction steps (corresponding to the execution of the increment instruction) leading to a process that is structurally equivalent to

$$\begin{array}{rcl}
Q = & \overline{p_2} \mid !p_1.(\overline{inc_1} | ack.\overline{p_2}) \mid !p_2.(\overline{dec_1} \mid (ack.\overline{p_3} \ + \ jmp.ack.\overline{p_2})) \mid \\
& (\nu\ m,u)\ \big(\overline{u} \mid \\
& \quad inc_1.(\overline{m} \mid \overline{u}) + dec_1.(u.\overline{m} + \overline{jmp}.(u.DIV \mid \overline{nr_1})) \mid \\
& \quad !m.(\overline{ack} \mid inc_1.(\overline{m} \mid \overline{u}) + dec_1.(u.\overline{m} + \overline{jmp}.(u.DIV \mid \overline{nr_1}))) \big) \mid \\
& !nr_1.(\nu\ m,u)\ \big(\\
& \quad \overline{m} \mid !m.(\overline{ack} \mid inc_1.(\overline{m} \mid \overline{u}) + dec_1.(u.\overline{m} + \overline{jmp}.(u.DIV \mid \overline{nr_1}))) \big).
\end{array}$$

https://doi.org/10.1017/S096012950999017X Published online by Cambridge University Press

Process Q has two deterministic reduction steps leading to a process structurally equivalent to

$$\begin{aligned} R = \; & (ack.\overline{p_3} \; + \; jmp.ack.\overline{p_2}) \mid !p_1.(\overline{inc_1}|ack.\overline{p_2}) \mid !p_2.(\overline{dec_1} \mid (ack.\overline{p_3} \; + \; jmp.ack.\overline{p_2})) \mid \\ & (\nu \; m,u) \; \big(\overline{u} \mid \\ & \quad (u.\overline{m} + \overline{jmp}.(u.DIV \mid \overline{nr_1})) \mid \\ & \quad !m.(\overline{ack} \mid inc_1.(\overline{m} \mid \overline{u}) + dec_1.(u.\overline{m} + \overline{jmp}.(u.DIV \mid \overline{nr_1}))) \; \big) \mid \\ & !nr_1.(\nu \; m,u) \; \big(\\ & \quad \overline{m} \mid !m.(\overline{ack} \mid inc_1.(\overline{m} \mid \overline{u}) + dec_1.(u.\overline{m} + \overline{jmp}.(u.DIV \mid \overline{nr_1}))) \; \big). \end{aligned}$$

Process R has two possible reduction steps: either a synchronisation on u or a synchronisation on *jump*. In the first case, the computation proceeds deterministically until a dead process with the program counter $\overline{p_3}$ is reached. In the second case, the following process is reached:

$$\begin{aligned} S = \; & ack.\overline{p_2} \mid !p_1.(\overline{inc_1}|ack.\overline{p_2}) \mid !p_2.(\overline{dec_1} \mid (ack.\overline{p_3} \; + \; jmp.ack.\overline{p_2})) \mid \\ & (\nu \; m,u) \; \big(\overline{u} \mid \\ & \quad (u.DIV \mid \overline{nr_1}) \mid \\ & \quad !m.(\overline{ack} \mid inc_1.(\overline{m} \mid \overline{u}) + dec_1.(u.\overline{m} + \overline{jmp}.(u.DIV \mid \overline{nr_1}))) \; \big) \mid \\ & !nr_1.(\nu \; m,u) \; \big(\\ & \quad \overline{m} \mid !m.(\overline{ack} \mid inc_1.(\overline{m} \mid \overline{u}) + dec_1.(u.\overline{m} + \overline{jmp}.(u.DIV \mid \overline{nr_1}))) \; \big). \end{aligned}$$

It is easy to see that the above process does not converge because it is not possible to remove the possibility of the processes $\overline{u}$ and $u.DIV$ synchronising. This synchronisation activates the divergent process DIV. This is made clear by assuming that such synchronisation is delayed for at least three reduction steps. In this case, the computation continues with three synchronisations on the channels nr_1, m and ack, and the following process is reached:

$$\begin{aligned} T = \; & \overline{p_2} \mid !p_1.(\overline{inc_1}|ack.\overline{p_2}) \mid !p_2.(\overline{dec_1} \mid (ack.\overline{p_3} \; + \; jmp.ack.\overline{p_2})) \mid \\ & (\nu \; m,u) \; \big(\overline{u} \mid u.DIV \mid \\ & \quad !m.(\overline{ack} \mid inc_1.(\overline{m} \mid \overline{u}) + dec_1.(u.\overline{m} + \overline{jmp}.(u.DIV \mid \overline{nr_1}))) \; \big) \mid \\ & (\nu \; m,u) \; \big(\\ & \quad inc_1.(\overline{m} \mid \overline{u}) + dec_1.(u.\overline{m} + \overline{jmp}.(u.DIV \mid \overline{nr_1})) \mid \\ & \quad !m.(\overline{ack} \mid inc_1.(\overline{m} \mid \overline{u}) + dec_1.(u.\overline{m} + \overline{jmp}.(u.DIV \mid \overline{nr_1}))) \; \big) \mid \\ & !nr_1.(\nu \; m,u) \; \big(\\ & \quad \overline{m} \mid !m.(\overline{ack} \mid inc_1.(\overline{m} \mid \overline{u}) + dec_1.(u.\overline{m} + \overline{jmp}.(u.DIV \mid \overline{nr_1}))) \; \big). \end{aligned}$$

The processes $\overline{u}$ and $u.DIV$ occurring on the second line of the definition above cannot interact with any other processes, thus the unique action they can perform is a synchronisation that activates the divergent process DIV.

Note that the content c_j of the register is modelled by the parallel composition of a corresponding number of processes $(\overline{u} \mid d.u.\overline{m})$: the term $\overline{u}$ represents a unit inside the register, while $d.u.\overline{m}$ is an auxiliary term that is responsible for removing the unit when the register is decremented. The register is blocked until either the prefix inc_j or the prefix dec_j fires. In the former case, the effect is that a new instance of the process $(\overline{u} \mid d.u.\overline{m})$ is spawned (and the $\overline{ack}$ is produced after re-activating the process able to

https://doi.org/10.1017/S096012950999017X Published online by Cambridge University Press

perform either the inc_j or the dec_j actions). In the latter case, the computation proceeds non-deterministically. Either the register is actually decremented (activating one of the processes $d.u.\overline{m}$), or the decision to jump is taken. In this case, a possibly diverging process $u.DIV$ is spawned and a new register instance is activated executing $\overline{nr_j}$. The new instance of the register guarantees that if the register was not empty at the time of the jump, the process $u.DIV$ eventually executes the u action (performing a synchronisation with one of the still available units $\overline{u}$) and starts its infinite computation.

It is worth noting that even in correct computations (that is, those corresponding to the RAM computation), some garbage is left when a jump instruction is executed. The garbage is due to the activation of a new instance of the register. The garbage processes (one for each register) are as follows:

$$\begin{array}{lll} G_j & : & (\nu\ m,u)\ \big(\\ & & \quad u.DIV\ \mid \\ & & \quad !m.(\overline{ack}\ \mid\ inc_j.(\overline{m}\ \mid\ \overline{u}) + dec_j.(u.\overline{m} + \overline{jmp}.(u.DIV\ \mid\ \overline{nr_j}))) \big) \end{array}$$

Note that the a garbage process is dead as it is composed of processes prefixed by input actions on restricted names.

We are now ready to define the RAM encoding formally.

Definition 7. 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 $(i, c_1, \ldots, c_n)$ of R, we define

$$\begin{array}{l} [\![(i, c_1, \ldots, c_n)]\!]_R\ = \\ (\nu\ p_1, \ldots, p_m, p_{m+1}, inc_1, dec_1, nr_1, \ldots, inc_n, dec_n, nr_n, ack, jmp) \\ (\ \overline{p_i}\ \mid\ [\![(1 : I_1)]\!]\ \mid \ldots \mid\ [\![(m : I_m)]\!]\ \mid\ p_{m+1}.\overline{w}\ \mid \\ \quad [\![r_1 = c_1]\!]\ \mid \ldots \mid\ [\![r_n = c_n]\!]\ \mid\ \prod_{k_1} G_1\ \mid \ldots \mid\ \prod_{k_n} G_n\ \ ) \end{array}$$

where the modelling of program instructions $[\![(i : I_i)]\!]$ and registers $[\![r_j = c_j]\!]$, and the garbage processes $G_1, \ldots, G_n$ have been defined above, and $k_1 \ldots k_n$ are natural numbers. Observe that, due to the presence of $k_1, \ldots, k_n$, the target of the encoding is not a unique process but a class of processes that differ only in the amount of garbage.

We are now ready to state the correspondence result between the computations carried out by a RAM and those carried out by its encoding.

Proposition 4. 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 $(i, c_1, \ldots, c_n)$, one of the following holds:

— $i \notin \{1, \ldots, m\}$ and each process in $[\![(i, c_1, \ldots, c_n)]\!]_R$ has a deterministic reduction step leading to a dead process.

— I_i is an increment instruction, $(i, c_1, \ldots, c_n) \rightarrow_R (i', c'_1, \ldots, c'_n)$, and each process in $[\![(i, c_1, \ldots, c_n)]\!]_R$ performs four deterministic reduction steps reaching a process Q that is structurally congruent with respect to $\equiv_R$ to some process in $[\![(i', c'_1, \ldots, c'_n)]\!]_R$.

— I_i is a decrement instruction, $(i, c_1, \ldots, c_n) \rightarrow_R (i', c'_1, \ldots, c'_n)$, and each process in $[\![(i, c_1, \ldots, c_n)]\!]_R$ performs two deterministic reduction steps reaching a process Q for which the following holds: Q is not dead and for each Q' such that $Q \longrightarrow Q'$, either Q' has a sequence of deterministic reduction steps (of length either 2 or 3) to a process Q''

https://doi.org/10.1017/S096012950999017X Published online by Cambridge University Press

that is structurally congruent with respect to $\equiv_R$ to some process in $[\![(i', c'_1, \ldots, c'_n)]\!]_R$, or Q' does not converge.

Proof. The proof for the first two items is immediate. In the third case, we observe that either the computation of the encodings in $[\![(i, c_1, \ldots, c_n)]\!]_R$ faithfully reproduces the RAM computation (and in this case the first part holds) or a 'wrong jump' (that is, a synchronisation on the channel *jmp* is executed even if the tested register c_j is not empty) is executed. In this case, the process

$$(\nu\ m, u)\ \big(\ \textstyle\prod_{c_j} \overline{u} \mid u.DIV \mid \overline{nr_j})) \mid \\ !m.(\overline{ack} \mid inc_j.(\overline{m} \mid \overline{u}) + dec_j.(u.\overline{m} + \overline{jmp}.(u.DIV \mid \overline{nr_j}))) \ \big)$$

appears at top level (that is, it is not guarded by any prefix). Hence, as $c_j > 0$, any process including the above process at top level does not terminate. □

As a simple corollary of the above proposition, we have that convergence is undecidable since a RAM terminates if and only if the corresponding encoding has a terminating computation.

Theorem 3. Let R be a RAM having the program $(1 : I_1), \ldots, (m : I_m)$ and the state $(i, c_1, \ldots, c_n)$, and let the process P be in $[\![(i, c_1, \ldots, c_n)]\!]_R$. Then $(i, c_1, \ldots, c_n)$ terminates if and only if P converges.

Proof. The proof is similar to the proof of Theorem 1, except that Proposition 4 is used instead of Proposition 2. □

This proves that convergence is undecidable in $\mathrm{CCS}_!$.

It is worth noting that unlike the RAM encoding presented for CCS_D, we cannot also conclude that termination is undecidable. In fact, the processes produced by the new encoding are non-deterministic, as additional infinite computations are also added to processes that encode a terminating RAM. Hence, for these processes it is no longer true that convergence and termination coincide.

We conclude this section by proving that weak bisimulation is undecidable also.

Theorem 4. Let R be a RAM having the program $(1 : I_1), \ldots, (m : I_m)$ and the state $(i, c_1, \ldots, c_n)$, and let the process P be in $[\![(i, c_1, \ldots, c_n)]\!]_R$. Then $(i, c_1, \ldots, c_n)$ terminates if and only if $P \approx \tau.P + \tau.\overline{w}$.

Proof. Assume that $(i, c_1, \ldots, c_n)$ does not terminate. Consider one of the processes P in $[\![(i, c_1, \ldots, c_n)]\!]_R$. By Theorem 3, no P can converge. In order to prove that $P \not\approx \tau.P + \tau.\overline{w}$, we proceed by contradiction and assume that $P \approx \tau.P + \tau.\overline{w}$. In this case, as $\tau.P + \tau.\overline{w} \xrightarrow{\tau} \overline{w} \xrightarrow{\overline{w}} \mathbf{0}$, we also have that $P \longrightarrow^* P' \xrightarrow{\overline{w}} P'' \longrightarrow^* P'''$ with $P''' \approx \mathbf{0}$. The only way that P' can perform a transition labelled with $\overline{w}$ is by activating the process $p_{m+1}.\overline{w}$. Moreover, as P cannot converge, P' (which is one of its derivatives) cannot converge either. It is easy to see that any infinite computation starting from P' cannot include reductions executed by an instruction (in fact, the program counter process $\overline{p_i}$ is definitively consumed by the input action p_{m+1}). Thus, all computations are due to the presence in P' of some DIV process. It is easy to see that such a process is also present in P''. Hence all derivatives

https://doi.org/10.1017/S096012950999017X Published online by Cambridge University Press

of P'', thus also P''', can perform actions on the observable channel w' that cannot be mimicked by $\mathbf{0}$. Hence, $P''' \not\approx \mathbf{0}$, which is in contradiction with what we observed above.

Assume now that $(i, c_1, \ldots, c_n)$ terminates. Consider a process P in $[\![(i, c_1, \ldots, c_n)]\!]_R$. We prove that $P \approx \tau.P + \tau.\overline{w}$ by showing that the relation

$$\begin{aligned}\mathscr{R} = Id \;\cup\; & \{(P, \tau.P + \tau.\overline{w})\} \\ & \cup \{(Q, \overline{w}) \mid Q \equiv_R Q'' \text{ with } Q' \longrightarrow Q'' \text{ and } Q' \in [\![(m+1, c_1, \ldots, c_n)]\!]_R \\ & \qquad \text{for some } c_1, \ldots, c_n\} \\ & \cup \{(Q, \mathbf{0}) \mid Q \equiv_R Q''' \text{ with } Q' \longrightarrow Q'' \xrightarrow{\overline{w}} Q''' \text{ and } Q' \in [\![(m+1, c_1, \ldots, c_n)]\!]_R \\ & \qquad \text{for some } c_1, \ldots, c_n\}\end{aligned}$$

(where Id is the identity relation) is a weak bisimulation.

We first consider the pair $(P, \tau.P + \tau.\overline{w})$. Any transition executable by P can be mimicked by the $\tau.P$ branch. The process $\tau.P + \tau.\overline{w}$ can perform two kinds of τ transitions. The first can be mimicked by P with no transition. The second can be mimicked by P by executing the whole terminating computation (which exists by Proposition 4) leading to a process structurally equivalent to a process in $[\![(m+1, c_1, \ldots, c_n)]\!]_R$. This process can perform an additional reduction (that is, the synchronisation on the channel p_{m+1}) and reach one of those terms Q such that, by the second line of the definition of $\mathscr{R}$, we have $(Q, \overline{w})$ is in $\mathscr{R}$.

We now consider one of the pairs $(Q, \overline{w})$ that are in $\mathscr{R}$ due to the second line in the definition. Both processes have a unique outgoing transition labelled with $\overline{w}$, leading to a process Q' and the empty process $\mathbf{0}$, respectively, such that, by the last line of the definition of $\mathscr{R}$, we have $(Q', \mathbf{0})$ is in $\mathscr{R}$.

We complete the proof by simply observing that, given one of the pairs $(Q, \mathbf{0})$ that are in $\mathscr{R}$ due to the last line in the definition, we have that both processes have no outgoing transitions. □

This proves that weak bisimulation is undecidable in $\text{CCS}_!$ also.

4.2. *Decidability of termination*

We have already observed that the undecidability result proved in Theorem 3 applies to convergence but not to termination. In this subsection we prove that, unlike the case for the calculus with general recursion, in $\text{CCS}_!$ termination is indeed decidable. This result is based on the theory of well-structured transition systems, so we start this section by recalling some basic definitions and results from Finkel and Schnoebelen (2001), which will be used in the following. Then, since these results are valid for finitely branching transition systems, we provide an alternative finitely branching semantics for $\text{CCS}_!$, which is equivalent with respect to termination to the one presented in Section 2. This allows us to prove that termination is decidable for $\text{CCS}_!$ processes by defining a suitable well-structured transition system for $\text{CCS}_!$.

4.3. *Well-structured transition system*

The following results and definitions are from Finkel and Schnoebelen (2001) unless otherwise specified. Recall that a *quasi-order* (or, equivalently, preorder) is a reflexive and transitive relation.

https://doi.org/10.1017/S096012950999017X Published online by Cambridge University Press

Definition 8 (well-quasi-order). A *well-quasi-order* (wqo) is a quasi-order $\leqslant$ over a set X such that, for any infinite sequence $x_0, x_1, x_2, \ldots$ in X, there exist indexes $i < j$ such that $x_i \leqslant x_j$.

Note that if $\leqslant$ is a wqo, then any infinite sequence $x_0, x_1, x_2, \ldots$ contains an infinite increasing subsequence $x_{i_0}, x_{i_1}, x_{i_2}, \ldots$ (with $i_0 < i_1 < i_2 < \ldots$). Thus well-quasi-orders exclude the possibility of having infinite strictly decreasing sequences.

We also need a formal definition of (finitely branching) transition systems. This can be given as follows. Here and in the following we use $\rightarrow^+$ to denote the transitive closure of the relation $\rightarrow$, and $\rightarrow^*$ for its reflexive and transitive closure.

Definition 9. A *transition system* is a structure $TS = (S, \rightarrow)$, where S is a set of *states* and $\rightarrow \subseteq S \times S$ is a set of *transitions*. We define $Succ(s)$ as the set $\{s' \in S \mid s \rightarrow s'\}$ of immediate successors of S. We say that TS is *finitely branching* if, for each $s \in S$, $Succ(s)$ is finite.

We also define $Pred(s)$ as the set $\{s' \in S \mid s' \rightarrow s\}$ of immediate predecessors of s, and use $Pred^*(s)$ to denote the set $\{s' \in S \mid s' \rightarrow^* s\}$ (of predecessors of s).

The functions $Succ$, $Pred$ and $Pred^*$ will also be used on sets by assuming that in this case they are defined by the point-wise extension of the above definitions.

The key tool for deciding several properties of computations is the notion of a well-structured transition system. This is a transition systems equipped with a well-quasi-order on states that is (upward) compatible with the transition relation. Here we will use a strong version of compatibility using the following definition.

Definition 10 (well-structured transition system with strong compatibility). A *well-structured transition system with strong compatibility* is a transition system $TS = (S, \rightarrow)$ equipped with a quasi-order $\leqslant$ on S such that the following two conditions hold:

1 $\leqslant$ is a well-quasi-order.
2 $\leqslant$ is strongly (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 $s_2 \leqslant t_2$ holds.

The following theorem is a special case of Finkel and Schnoebelen (2001, Theorem 4.6), and will be used to prove our decidability result.

Theorem 5. Let $TS = (S, \rightarrow, \leqslant)$ be a finitely branching, well-structured transition system with strong compatibility, decidable $\leqslant$ and computable $Succ$. Then the existence of an infinite computation starting from a state $s \in S$ is decidable.

We will also need a result due to Higman, which allows us to extend a well-quasi-order from a set S to the set of the finite sequences on S. To be more precise, given a set S, we use S^* to denote the set of finite sequences built by using elements in S. We can define a quasi order on S^* as follows.

Definition 11. Let S be a set and $\leqslant$ be a quasi order over S. The relation $\leqslant_*$ over S^* is defined as follows. Let $t, u \in S^*$, with $t = t_1 t_2 \ldots t_m$ and $u = u_1 u_2 \ldots u_n$. We have that

https://doi.org/10.1017/S096012950999017X Published online by Cambridge University Press

$t \leqslant_* u$ if and only if there exists an injection f from $\{1, 2, \ldots, m\}$ into $\{1, 2, \ldots, n\}$ such that $t_i \leqslant u_{f(i)}$ and $i \leqslant f(i)$ for $i = 1, \ldots, m$.

The relation $\leqslant_*$ is clearly a quasi order over S^*. It is also a wqo since we have the following result.

Lemma 1 (Higman). Let S be a set and $\leqslant$ be a wqo over S. Then the relation$\leqslant_*$ is a wqo over S^*.

Finally, we will also use the following proposition, whose proof is immediate.

Proposition 5. Let S be a finite set. Then the equality is a wqo over S.

4.3.1. *A finitely branching transition system for CCS*$_!$. Because of the rule for replication, the transition systems for CCS$_!$ defined in Section 2 is not finitely branching. As previously mentioned, Theorem 5 only applies to finitely branching transition systems. Hence, in order to use it, we need to define a new finitely branching transition system for CCS$_!$, which is equivalent to the one presented in Section 2 with respect to termination. This new transition system can be obtained by reformulating the rule for replication as follows.

We define a new transition relation $\overset{\alpha}{\mapsto}$ over CCS$_!$ processes as the least relation satisfying all the axioms and rules of Table 2 (where $\overset{\alpha}{\longrightarrow}$ is substituted by $\overset{\alpha}{\mapsto}$) plus the following rules REPL1 and REPL2:

$$\texttt{REPL1}: \frac{P \overset{\alpha}{\mapsto} P'}{!P \overset{\alpha}{\mapsto} P' \mid !P} \qquad \texttt{REPL2}: \frac{P \overset{\alpha}{\mapsto} P' \quad P \overset{\overline{\alpha}}{\mapsto} P''}{!P \overset{\tau}{\mapsto} P' \mid P'' \mid !P}$$

Rule REPL of Section 2 caused an infinitely branching transition system because in the premise one could use an unbound number of copies of the process P. So, for example, from $(\alpha.P)!$ one could obtain the transitions

$$\begin{aligned} &!(\alpha.P) \overset{\alpha}{\longrightarrow} P \mid (\alpha.P)!, \\ &!(\alpha.P) \overset{\alpha}{\longrightarrow} P \mid P \mid !(\alpha.P) \\ &\vdots \end{aligned}$$

This behaviour is ruled out when using REPL1, where the premise does not involve the !. Rule REPL2 is added because some computations depend on the possibility of synchronising two copies of P generated by $!P$.

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

In order to prove the equivalence with respect to termination of the semantics of CCS$_!$ presented in Section 2 and the alternative semantics presented here, we use the following congruence on processes, which allows us to simplify the proofs.

https://doi.org/10.1017/S096012950999017X Published online by Cambridge University Press

Definition 12. We define $\equiv_T$ as the least congruence relation satisfying the following axioms:

$$\begin{aligned} P|Q &\equiv_T Q|P \\ P|(Q|R) &\equiv_T (P|Q)|R \\ P|\mathbf{0} &\equiv_T P \\ P|!P &\equiv_T !P. \end{aligned}$$

The following proposition shows that congruent processes are equivalent with respect to termination when considering the $\longrightarrow$ transitions.

Proposition 6. Let $P, Q \in \mathrm{CCS}_!$ with $P \equiv_T Q$. If $P \xrightarrow{\alpha} P'$, there exists Q' such that $Q \xrightarrow{\alpha} Q'$ and $P' \equiv_T Q'$.

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

The same also holds in the case of $\mapsto$ transitions, as shown by the following.

Proposition 7. Let $P, Q \in \mathrm{CCS}_!$ with $P \equiv_T Q$. If $P \overset{\alpha}{\mapsto} P'$, there exists Q' such that $Q \overset{\alpha}{\mapsto} Q'$ and $P' \equiv_T Q'$.

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

We have the following two propositions, which show that $\longrightarrow$ and $\mapsto$ induce the same derivations up to $\equiv_T$ congruence.

Proposition 8. Let $P \in \mathrm{CCS}_!$. If $P \xrightarrow{\alpha} P'$, there exists P'' such that $P \overset{\alpha}{\mapsto} P''$ and $P' \equiv_T P''$.

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

Proposition 9. Let $P \in \mathrm{CCS}_!$. If $P \overset{\alpha}{\mapsto} P'$, there exists P'' such that $P \xrightarrow{\alpha} P''$ and $P' \equiv_T P''$.

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

From these four propositions we can immediately obtain the result stating the equivalence with respect to termination.

Corollary 1. Let $P \in \mathrm{CCS}_!$. Then P terminates according to the semantics (defined by) $\longrightarrow$ if and only if P terminates according to the new semantics $\mapsto$.

4.3.2. *Termination is decidable in CCS*$_!$. The final, and the most complicated, step in proving that termination is decidable for $(\mathrm{CCS}_!, \longrightarrow)$ is to equip the transition system $(\mathrm{CCS}_!, \mapsto)$ with a well-quasi-order compatible with $\mapsto$. The desired result then follows from Theorem 5 and Corollary 1.

To define the well-quasi-order, we first introduce another congruence on processes, which is simpler than $\equiv_T$ and turns out to be compatible with $\mapsto$. In fact, unlike the case in the previous section, here the congruence is only needed to simplify the definition of the quasi-order so that we do not need to take the replication operator into account.

https://doi.org/10.1017/S096012950999017X Published online by Cambridge University Press

Definition 13. We define $\equiv_w$ as the least congruence relation satisfying the following axioms:

$$\begin{aligned} P|Q &\equiv_w Q|P \\ P|(Q|R) &\equiv_w (P|Q)|R \\ P|\mathbf{0} &\equiv_w P. \end{aligned}$$

We can now define the relation $\preceq$, which we will now prove to be a well-quasi-order.

Definition 14. Let $P, Q \in \mathrm{CCS}_!$. We write $P \preceq Q$ if and only if there exist n, $x_1, \ldots, x_n$, P', R, $P_1, \ldots, P_n$, $Q_1, \ldots, Q_n$ such that

$$\begin{aligned} P &\equiv_w P' | \prod_{i=1}^{n} (\nu x_i) P_i \\ Q &\equiv_w P' | R | \prod_{i=1}^{n} (\nu x_i) Q_i, \end{aligned}$$

and $P_i \preceq Q_i$ for $i = 1, \ldots, n$.

Intuitively, $P \preceq Q$ holds if Q can be obtained, up to $\equiv_w$, from P by adding some parallel processes, while preserving the nesting structure given by restrictions. In order to show that the relation $\preceq$ is indeed a quasi order, we need some more notation and a preliminary lemma.

First we define the maximum number $d_\nu(P)$ of nested restrictions in a process P.

Definition 15. Let $P \in \mathrm{CCS}_!$. We define $d_\nu(P)$ inductively as follows:

$$\begin{aligned} d_\nu(\alpha.P) &= d_\nu(P) \\ d_\nu(P + Q) &= max(\{d_\nu(P), d_\nu(Q)\}) \\ d_\nu(P|Q) &= max(\{d_\nu(P), d_\nu(Q)\}) \\ d_\nu((\nu x)P) &= 1 + d_\nu(P) \\ d_\nu(!P) &= d_\nu(P). \end{aligned}$$

We also need a notation for indicating all the sequential and bang subprocesses of P.

Definition 16. Let $P \in \mathrm{CCS}_!$. The set $Sub(P)$ containing all the sequential and bang subprocesses of P is defined inductively as follows:

$$\begin{aligned} Sub(\alpha.P) &= \{\alpha.P\} \cup Sub(P) \\ Sub(P + Q) &= \{P + Q\} \cup Sub(P) \cup Sub(Q) \\ Sub(P|Q) &= Sub(P) \cup Sub(Q) \\ Sub((\nu x)P) &= Sub(P) \\ Sub(!P) &= \{!P\} \cup Sub(P). \end{aligned}$$

Finally, we use $\mathscr{P}_{P,n}$ to denote the set of all those $\mathrm{CCS}_!$ processes whose nesting level of restrictions is no greater than n and such that their sequential subprocesses, bang

https://doi.org/10.1017/S096012950999017X Published online by Cambridge University Press

subprocesses and bound names are contained in the corresponding elements of P. More precisely, we have the following definition.

Definition 17 ($\mathscr{P}_{P,n}$). Let n be a natural number and P be a process. We define $\mathscr{P}_{P,n}$ by

$$\mathscr{P}_{P,n} = \{Q \in \mathrm{CCS}_! \mid Sub(Q) \subseteq Sub(P) \wedge bn(Q) \subseteq bn(P) \wedge d_v(Q) \leqslant n\}.$$

The notion of $\mathscr{P}_{P,n}$ is important because processes contained in $\mathscr{P}_{P,n}$ can be written in a sort of normal form (up to $\equiv_w$), which allows us to simplify the proofs. This is the content of the following lemma.

Lemma 2. Let $P \in \mathrm{CCS}_!$, $n \leqslant d_v(P)$ and $Q \in \mathscr{P}_{P,n}$. Suppose that $|bn(P)| = m$ and $bn(P) = \{x_1, \ldots, x_m\}$. Then there exist $l, k_1, \ldots, k_m$ such that

$$Q \equiv_w \prod_{i=1}^{l} Q_i \mid \prod_{j=1}^{m} (\prod_{h=1}^{k_j} (\nu x_j) R_{j,h})$$

for some

$Q_i \in Sub(P)$ for $i = 1, \ldots, l$

$R_{j,h} \in \mathscr{P}_{P,n-1}$ for $j = 1, \ldots, m$ (and corresponding $h = 1, \ldots k_j$).

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

Using this normal form, we can prove that $\preceq$ is a quasi order.

Proposition 10. The relation $\preceq$ is a quasi order over $\mathrm{CCS}_!$ processes.

Proof. Transitivity of $\preceq$ is a consequence of the fact that if $P \preceq Q$, by the definition of $\preceq$ and Lemma 2, it follows that

$$P \equiv_w \prod_{i=1}^{l} P_i \mid \prod_{j=1}^{m} \prod_{h=1}^{k_j} (\nu x_j) R_{j,h}$$

and

$$Q \equiv_w \prod_{i=1}^{l+l'} P_i \mid \prod_{j=1}^{m} \prod_{h=1}^{k_j+k'_j} (\nu x_j) R'_{j,h}$$

with $P_i \in Sub(P|Q)$ for $i = 1, \ldots, l + l'$ and $R_{j,h} \preceq R'_{j,h}$ for $j = 1, \ldots, m$ (and corresponding $h = 1, \ldots, k_j$). □

To prove that $\preceq$ is a well-quasi-order, we need two more preliminary results. The first, which is rather obvious, states that the set of sequential and bang subprocesses of a process is finite.

Proposition 11. Given a process $P \in \mathrm{CCS}_!$ the set $Sub(P)$ is finite.

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

The second preliminary result shows that when we perform a derivation step, we obtain a process that is no more 'complicated' than the original one. This is an important feature

https://doi.org/10.1017/S096012950999017X Published online by Cambridge University Press

of $\mathrm{CCS}_!$, which is due to the absence of recursive definitions. Essentially, this is the key property that allows us to obtain the decidability of termination.

Proposition 12. Let $P \in \mathrm{CCS}_!$ and $Q \in \mathscr{P}_{P,n}$. If $Q \overset{\alpha}{\mapsto} Q'$, then $Q' \in \mathscr{P}_{P,n}$.

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

An immediate consequence of the above proposition is that all the processes reachable from P through a sequence of reduction steps belong to $\mathscr{P}_{P,n}$, where n is the maximum level of nesting of P. We state this fact as an explicit corollary since we will need it later. So, we use $Deriv(P)$ to denote the processes reachable from P through a sequence of reduction steps.

Definition 18. Let $P \in \mathrm{CCS}_!$. Then we define

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

Corollary 2. Let $P \in \mathrm{CCS}_!$. Then $Deriv(P) \subseteq \mathscr{P}_{P,d_v(P)}$ holds.

Proof. The proof is immediate from Proposition 12. □

Now we are ready to prove that $\preceq$ is a well-quasi-ordering. The key idea is as follows. We use Lemma 2 to transform each derivative of P in $1 + m$ (finite) sequences, where m is the cardinality of $bn(P)$. The first sequence is over $Sub(P)$, which is a finite set, whereas the other sequences are over processes that are 'simpler' than P in the sense that the nesting level of restrictions in those processes is strictly smaller than $d_v(P)$. The result is proved by induction on the nesting level of restrictions and using Higman's lemma.

Theorem 6. Let $P \in \mathrm{CCS}_!$ and $n \geqslant 0$. The relation $\preceq$ is a wqo over $\mathscr{P}_{P,n}$.

Proof. The proof is by induction on n.

$n = 0$:

For the base case, take an infinite sequence $P_1, P_2, \ldots, P_i, \ldots$, with $P_i \in \mathscr{P}_{P,0}$ for $i > 0$. By Lemma 2, for any i we have $P_i \equiv_w \prod_{j=1}^{n_i} P_{i,j}$, with $P_{i,j} \in Sub(P)$. Hence, we have an infinite sequence of elements of $Sub(P)^*$. As $Sub(P)$ is finite (by Proposition 11), by Proposition 5 and Higman's Lemma (Lemma 1), we have that $=_*$ is a wqo over $Sub(P)^*$ (the relation $=_*$ on sequences is defined according to Definition 11, where we consider equality as the quasi order on the starting set).

It is easy to see that if $P_{i,1}P_{i,2}\ldots P_{i,n_i} =_* P_{k,1}P_{k,2}\ldots P_{k,n_k}$, then $P_i \preceq P_k$.

$n > 0$:

For the inductive step, take an infinite sequence $P_1, P_2, \ldots, P_i, \ldots$, with $P_i \in \mathscr{P}_{P,n}$ for any $i > 0$. By Lemma 2, there exists m such that for any i we have

$$P_i \equiv_w \prod_{j=1}^{n_i} P_{i,j} \mid \prod_{j=1}^{m} \prod_{h=1}^{k_{i,j}} (\nu x_j) R_{i,j,h}$$

with $P_{i,j} \in Sub(P)$ and $R_{i,j,h} \in \mathscr{P}_{P,n-1}$.

https://doi.org/10.1017/S096012950999017X Published online by Cambridge University Press

Hence, each P_i can be seen as composed of $m+1$ finite sequences:

$$\begin{array}{l} P_{i,1} \dots P_{i,n_i} \\ R_{i,1,1} \dots R_{i,1,k_{i,1}} \\ \vdots \\ R_{i,m,1} \dots R_{i,m,k_{i,m}}. \end{array}$$

Note that the first sequence is composed of elements from the finite set $Sub(P)$, whereas the other m sequences are composed of elements in $\mathscr{P}_{P,n-1}$. We know from the base case that $=_*$ is a wqo over $Sub(P)^*$.

By the induction hypothesis, we have that $\preceq$ is a wqo on $\mathscr{P}_{P,n-1}$. Hence, by Higman's Lemma, we have that $\preceq_*$ is a wqo on $\mathscr{P}^*_{P,n-1}$.

We start by extracting an infinite subsequence from $P_1 \dots P_i \dots$ making the finite sequences $P_{i,1} \dots P_{i,n_i}$ increasing with respect to $=_*$. Then we extract an infinite subsequence from the subsequence obtained in the previous step, which makes the finite sequences $R_{i,1,1} \dots R_{i,1,k_{i,1}}$ increasing with respect to $\preceq_*$, and so on.

At the end of the process, we obtain an infinite subsequence of $P_1 \dots P_i \dots$ that is ordered with respect to $\preceq$. □

As the final step in obtaining our decidability result, we need to show that the relation $\preceq$ of Definition 14 is strongly compatible with $\overset{\alpha}{\mapsto}$. This is the content of Theorem 7 below, which uses the following proposition.

Proposition 13. Let $P, Q \in \mathrm{CCS}_!$. If $P \equiv_w Q$ and $Q \overset{\alpha}{\mapsto} Q'$, there exists P' such that $P \overset{\alpha}{\mapsto} P'$ and $P' \equiv_w Q'$.

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

Theorem 7. Let $P, Q, P' \in \mathrm{CCS}_!$. If $P \overset{\alpha}{\mapsto} P'$ and $P \preceq Q$, there exists Q' such that $Q \overset{\alpha}{\mapsto} Q'$ and $P' \preceq Q'$.

Proof. The proof is by induction on $d_v(P)$.

By the definition of $\preceq$, we have $P \equiv_w \overline{P} | \prod_{i=1}^n (\nu x_i) P_i$ and $Q \equiv_w \overline{P} | R | \prod_{i=1}^n (\nu x_i) Q_i$, with $P_i \preceq Q_i$ for $i = 1, \dots, n$.

First note that $d_v(P_i) < d_v(P)$ for $i = 1, \dots, n$.

As $P \overset{\alpha}{\mapsto} P'$, by Proposition 13, we also have $\overline{P} | \prod_{i=1}^n (\nu x_i) P_i \overset{\alpha}{\mapsto} P''$ with $P'' \equiv_w P'$. The proof then proceeds by case analysis on the last rule applied in the proof of the transition $\overline{P} | \prod_{i=1}^n (\nu x_i) P_i \overset{\alpha}{\mapsto} P''$. □

We can now state the main result of this section.

Theorem 8. Let $P \in \mathrm{CCS}_!$. Then the transition system $(Deriv(P), \mapsto, \preceq)$ is a finitely branching well-structured transition system with strong compatibility, decidable $\preceq$ and computable *Succ*.

Proof. The fact that $(Deriv(P), \mapsto)$ is finitely branching follows from an inspection of the transition rules (in particular, `REPL1` and `REPL2`). The fact that $\preceq$ is a well-quasi-order

https://doi.org/10.1017/S096012950999017X Published online by Cambridge University Press

on $Deriv(P)$ is a consequence of Corollary 2 and Theorem 6 (taking $n = d_v(P)$). Strong compatibility was proved in Theorem 7. □

Corollary 3. Let $P \in \mathrm{CCS}_!$. The termination of process P is decidable.

Proof. The proof is immediate from Theorem 5, Corollary 1 and Theorem 8. □

4.4. *Decidability of barb*

In this section we show that the ability of a process to perform, possibly after some internal moves, an observable action on a given channel is a decidable property in the calculus with replication. As previously mentioned, this property will be called barb, for short.

This result is also based on the theory of well-structured transition systems, but we will need some additional definitions and results from Finkel and Schnoebelen (2001).

Recall that given a quasi-order $\leqslant$ over X, an *upward-closed set* is a subset $I \subseteq X$ such that $\forall x, y \in X : (x \in I \wedge x \leqslant y) \Rightarrow y \in I$. Given $x \in X$, we define its upward closure as $\uparrow x = \{y \in X \mid x \leqslant y\}$. This notion can be extended to sets in the obvious way: given a set $Y \subseteq X$, we define its upward closure as $\uparrow Y = \bigcup_{y \in Y} \uparrow y$.

Definition 19 (Finite basis). A *finite basis* of an upward-closed set I is a finite set B such that $I = \bigcup_{x \in B} \uparrow x$.

In our case the notion of basis is particularly important when considering the basis of the predecessor of a state in a transition system. More precisely, we are interested in an effective pred-basis as defined below. Recall from Definition 9 that $Pred(S)$ denotes the immediate predecessors of a set of states S, while $Pred^*(S)$ are the predecessors.

Definition 20 (Effective pred-basis). A well-structured transition system has an *effective pred-basis* if there exists an algorithm such that, for any any state $s \in S$, it returns the set $pb(s)$ that is a finite basis of $\uparrow Pred(\uparrow s)$.

The following proposition is a special case of Finkel and Schnoebelen (2001, Proposition 3.5).

Proposition 14. Let $TS = (S, \rightarrow, \leqslant)$ be a finitely branching, well-structured transition system with strong compatibility, decidable $\leqslant$ and effective pred-basis. It is possible to compute a finite basis of $Pred^*(I)$ for any upward-closed set I given via a finite basis.

This proposition can be used to prove that $P \Downarrow x$ is decidable by exploiting the following idea. Clearly, a process can perform an action α (in any number of steps) if and only if such a process is a predecessor of a process that can perform α immediately (that is, in one step). If one can show that the set S consisting of those processes that can immediately perform α is upward closed, the previous result allows us to compute effectively a finite pred-basis of $Pred^*(S)$ and therefore to decide whether a process Q belongs to $Pred^*(S)$. To this end, in fact, it is sufficient to decide whether in the (finite) basis there exists a process that is smaller than Q (this is possible because the quasi-order $\leqslant$ is decidable).

In order to develop this idea, we first need to perform three steps.

https://doi.org/10.1017/S096012950999017X Published online by Cambridge University Press

First we have to define a finitely branching, well-structured transition system with strong compatibility, decidable $\preceq$ and computable *Succ*. Such a system was obtained in the previous subsection, so we simply use that system here[†].

Next we have to show that the notion of barb does not change when considering the new relation $\mapsto$ (used in the finitely branching semantics) instead of the old one $\longrightarrow$. This is the content of the following proposition, where we assume that barbs in the new semantics are defined in the obvious way: we say that $P \Downarrow x$ if and only if there exists P', P'', α such that $P \mapsto^* P' \stackrel{\alpha}{\mapsto} P''$ and $n(\alpha) = \{x\}$.

Proposition 15. Let $P \in \mathrm{CCS}_!$. Then $P \Downarrow x$ if and only if $P \Downarrow x$.

Proof. The proof is analogous to that of Corollary 1. □

The third, and most substantial, step needed to prove that $P \Downarrow x$ (and therefore $P \Downarrow x$) is decidable consists of the definition of an effective pred-basis for our finitely branching, well-structured transition system. This will be done in the next subsection.

4.4.1. *An effective pred-basis for* $(\mathscr{P}_{P,d_v(P)}, \mapsto, \preceq)$. We start by defining the set of processes $Now_\alpha(P)$ that can immediately perform the labelled action α and are constructed using sequential and bang subprocesses of P (in the sense made precise by Definition 17).

Definition 21. Let $P \in \mathrm{CCS}_!$. The set of processes $Now_\alpha(P)$ is defined as $\{Q \in \mathscr{P}_{P,d_v(P)} \mid Q \stackrel{\alpha}{\mapsto}\}$.

Next we show that this set is upward-closed.

Proposition 16. Let $P \in \mathrm{CCS}_!$. Then $Now_\alpha(P) =\uparrow Now_\alpha(P)$ holds.

Proof. The proof is immediate from the definition of $\preceq$. □

We can now provide a finite basis for the previously defined set. This is possible because the set of sequential and bang subprocesses of a process is finite.

Definition 22. Let $P \in \mathrm{CCS}_!$. The set $fbNow_\alpha(P)$ is defined as follows:

$$fbNow_\alpha(P) = \{(\nu x_1 \ldots x_m)Q \mid \begin{array}{l} Q \in Sub(P), \\ m \leqslant d_v(P), \\ x_1 \ldots x_m \subseteq bn(P), \\ Q \stackrel{\alpha}{\mapsto}, n(\alpha) \notin \{x_1, \ldots, x_m\}\}. \end{array}$$

[†] To be more precise, in the transition system here we have to consider $\mathscr{P}_{P,d_v(P)}$ rather than $Deriv(P)$, because $Deriv(P)$ does not include all the possible states that we are dealing with (for example, in the basis). However, the proofs are the same.

https://doi.org/10.1017/S096012950999017X Published online by Cambridge University Press

Proposition 17. Let $P \in \mathrm{CCS}_!$ and $\alpha \neq \tau$. Then the set $fbNow_\alpha(P)$ is a finite basis of $Now_\alpha(P)$.

Proof. The set $fbNow_\alpha(P)$ is finite by construction and by Proposition 11. Moreover, it is also a basis, since if $R \in Now_\alpha(P)$, we can show by induction on the structure of R that there exists $Q \in fbNow_\alpha(P)$ such that $Q \preceq R$. □

Using the above definitions and results, we can provide a method to construct a finite basis for the set of predecessors of a given process with respect to a transition $\overset{\alpha}{\mapsto}$. The idea is to first consider sequential and bang processes (for which, as we will show, it is possible to compute a pred-basis directly) and then consider restricted and parallel processes (for which we proceed by induction on their syntactical structure). In order to compute the pred-basis directly for a sequential process Q, we simply observe that given any transition $S \overset{\alpha}{\mapsto} S'$ such that $Q \preceq S'$, we have that either Q was already present (at top level) in S or not. In the first case a pred-basis can be obtained by application of Proposition 17 by simply considering the processes $Q|R$ with $R \in fbNow_\alpha(P)$. In the second case, the process Q appears at top level only in S', which means that it is 'activated' by one sequential or bang process, let us call it R. The activation occurs either because R performs a transition labelled with α directly or because it synchronises with a parallel process (in the case $\alpha = \tau$). In the first case, it is enough to consider the sequential or bang subprocesses of P that are able to activate Q by performing a transition labelled with α; in the second case, it is necessary to consider sequential or bang subprocesses of P that are able to activate Q after performing a synchronisation with another process in parallel.

Definition 23. Let $P \in \mathrm{CCS}_!$. Given a process $Q \in \mathscr{P}_{P,d_v(P)}$, we define

$$\begin{aligned} basic_\alpha(Q) = {} & \{Q|R \mid R \in fbNow_\alpha(P)\} \cup \\ & \{R \in Sub(P) \mid \exists R' : R \overset{\alpha}{\mapsto} R' \wedge Q \preceq R'\} \cup \\ & synchbasic_\alpha(Q) \end{aligned}$$

where:

$$\begin{aligned} synchbasic_\tau(Q) = \{Q_1|Q_2 \mid {} & Q_1 \in Sub(P) \wedge Q_2 \in fbNow_\alpha(P) \wedge \\ & \exists Q_1' : Q_1 \overset{\overline{\alpha}}{\mapsto} Q_1' \wedge Q \preceq Q_1' \} \\ synchbasic_\alpha(Q) = \varnothing \quad & \text{if } \alpha \neq \tau. \end{aligned}$$

The pred-basis $pb_\alpha(Q)$ of a process Q with respect to α is defined by induction on the structure of the process as follows:

$$\begin{aligned} pb_\alpha(\mathbf{0}) &= basic_\alpha(\mathbf{0}) \\ pb_\alpha(\alpha'.Q) &= basic_\alpha(\alpha'.Q) \\ pb_\alpha(Q_1 + Q_2) &= basic_\alpha(Q_1 + Q_2) \\ pb_\alpha((\nu x)Q) &= basic_\alpha((\nu x)Q') \\ & \quad \cup \\ & \quad \{(\nu x)Q' \mid Q' \in pb_\alpha(Q) \wedge x \neq n(\alpha)\} \end{aligned}$$

https://doi.org/10.1017/S096012950999017X Published online by Cambridge University Press

$$
\begin{aligned}
pb_\alpha(Q_1|Q_2) &= basic_\alpha(Q_1|Q_2) \\
&\quad \cup \\
&\quad \{Q_1'|Q_2 \mid Q_1' \in pb_\alpha(Q_1)\} \\
&\quad \cup \\
&\quad \{Q_1|Q_2' \mid Q_2' \in pb_\alpha(Q_2)\} \\
&\quad \cup \\
&\quad sync_\alpha(Q_1, Q_2) \\
pb_\alpha(!Q) &= basic_\alpha(!Q) \\
sync_\tau(Q_1, Q_2) &= \{Q_1'|Q_2' \mid \exists \alpha \in n(P) : Q_1' \in pb_\alpha(Q_1) \wedge Q_2' \in pb_{\overline{\alpha}}(Q_2')\} \\
sync_\alpha(Q_1, Q_2) &= \varnothing \qquad \text{if } \alpha \neq \tau.
\end{aligned}
$$

This definition provides a procedure to compute a pred-basis of a process as shown by the following Lemma.

Lemma 3. Let $P \in \mathrm{CCS}_!$ and $Q \in \mathscr{P}_{P,d_v(P)}$. Then $pb_\tau(Q)$ is a computable finite basis of $\uparrow Pred(\uparrow Q)$.

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

Using the previous Lemma, we can obtain the following Theorem.

Theorem 9. Let $P \in \mathrm{CCS}_!$. Then the transition system $(\mathscr{P}_{P,d_v(P)}, \mapsto, \preceq)$ is a well-structured transition system with strong compatibility, decidable $\preceq$ and effective pred-basis.

Proof. The first part is the content of Theorem 8 (note that in Theorem 8, $Deriv(P)$ rather than $\mathscr{P}_{P,d_v(P)}$ was used, but the proof of such a theorem also works when we use $\mathscr{P}_{P,d_v(P)}$). The second part follows from Lemma 3. □

We can now prove the main result of this section following the argument previously illustrated. First we characterise barbs in terms of predecessors.

Proposition 18. Let $P \in \mathrm{CCS}_!$. $P \Downarrow x$ if and only if $P \in Pred^*(Now_x(P))$ or $P \in Pred^*(Now_{\overline{x}}(P))$.

Proof. The proof is immediate from the definitions of $P \Downarrow x$ and $Pred^*$, and by Definition 21. □

Corollary 4. Let $P \in \mathrm{CCS}_!$. Then $P \downarrow x$ is decidable.

Proof. First observe that from Theorem 9 and Proposition 14, we can compute a finite basis of $Pred^*(I)$ for any upward-closed set I specified by means of a finite basis. Next note that the set $Now_\alpha(P)$ of processe that can immediately perform a (non-silent) move α is upward-closed (Proposition 16), and we have provided a finite basis for it (Proposition 17). Hence, since $\preceq$ is decidable, we can decide whether a given process Q belongs to $Pred^*(Now_\alpha(P))$ by verifying if there exists a process in the finite basis (of $Pred^*(Now_\alpha(P))$) that is smaller than Q. This and Proposition 18 imply that $P \Downarrow x$ is decidable and therefore the theorem follows from Proposition 15. □

https://doi.org/10.1017/S096012950999017X Published online by Cambridge University Press

5. Decidability results for CCS$_*$

We show that the set of processes reachable from a given process P is finite. Hence, all the properties considered in this paper are decidable in CCS$_*$.

Definition 24. $Reach(P)$ is the set of terms reachable from P through a sequence of transitions: $Reach(P) = \{Q \mid \exists n \geqslant 0, \alpha_1, \ldots, \alpha_n \text{ s.t. } P \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_n} Q\}$.

We provide an upper bound to the number of reachable processes.

Definition 25. The function *size* on CCS$_*$ processes is defined as follows:

$$\begin{aligned} size(\mathbf{0}) &= 1 \\ size(\alpha.P) &= 1 + size(P) \\ size(P + Q) &= 1 + size(P) + size(Q) \\ size(P|Q) &= size(P) \times size(Q) \\ size((\nu x)P) &= size(P) + 1 \\ size(P^*) &= size(P) + 1. \end{aligned}$$

In order to prove that *size* is actually an upper bound to the number of processes reachable from a given process, we need the following Lemma.

Lemma 4. Let $P \in \text{CCS}_*$. For any $n > 0$, we have that if $P^* \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_n} Q$ or $R; P^* \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_n} Q$ with $R \in Reach(P)$, then $Q = Q'; P^*$ with $Q' \in Reach(P)$.

Proof. We proceed by induction on n.

$n = 1$:

We divide the base case into two parts. In the first part we consider $P^* \xrightarrow{\alpha_1} Q$. In this case the transition is implied by the rule `ITER`, thus $P \xrightarrow{\alpha_1} Q'$ (hence $Q' \in Reach(P)$) and $Q = Q'; P^*$. In the second part we consider $R; P^* \xrightarrow{\alpha_1} Q$ (with $R \in Reach(P)$). In this case the transition is implied by either the rule `SEQ1` or `SEQ2`. If it is `SEQ1`, it is easy to see that the lemma holds. If it is `SEQ2`, we have that $P^* \xrightarrow{\alpha_1} Q$, hence we can reason as in the first part.

$n > 1$:

By the induction hypothesis,

$$P^* \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_{n-1}} T \xrightarrow{\alpha_n} Q$$

where $T = Q'; P^*$ with $Q' \in Reach(P)$. If we consider the last transition, $Q'; P^* \xrightarrow{\alpha_n} Q$, we can prove that the lemma holds by simply resorting to the second part of the base case. □

Proposition 19. Let $P \in \text{CCS}_*$. Then $|Reach(P)| \leqslant size(P)$.

Proof. We use induction on the structure of P. The unique non-trivial case is $P = Q^*$ in the induction step. In this case we use the above Lemma, which allows us to conclude that $Reach(Q^*) = \{Q^*\} \cup \{R; Q^* \mid R \in Reach(Q)\}$. □

https://doi.org/10.1017/S096012950999017X Published online by Cambridge University Press

As a trivial corollary, we have that the set of processes reachable from any process in CCS_* is finite.

Corollary 5. Let $P \in \mathrm{CCS}_*$. The set $Reach(P)$ is finite.

As a consequence of the above corollary, we get that termination, convergence and barb are decidable. In fact, in a finite labelled transition system: termination can be checked by verifying the absence of sequences of τ-labelled transitions starting from the initial state that includes a loop; convergence can be checked by verifying the presence of a sequence of τ-labelled transitions starting form the initial state and leading to a state without outgoing τ-labelled transitions; and a barb on w can be checked by verifying the presence of a sequence of τ-labelled transitions starting from the initial state and leading to a state with an outgoing transition labelled with w or $\overline{w}$. Moreover, we also have that weak bisimulation is decidable as we can test weak bisimilarity of two finite labelled transition systems using, for example, the algorithm proposed in Kanellakis and Smolka (1990).

6. Conclusions and related work

In this paper we have investigated the expressive power of three different constructs for modelling infinite behaviours in process calculi. More precisely, we have considered a finite fragment of CCS that we have extended with recursive definitions, replication and iteration, respectively. We have considered four different properties for processes: process termination (that is, all runs are finite); process convergence (that is, there exists a finite completed run); barb (a process has the ability to perform visible actions on a specific channel); and weak bisimulation between processes. For each of these properties we have proved whether they are decidable or not in the three considered calculi; the results were presented in Table 1 in the Introduction.

As a consequence of the results we have proved in this paper, there exists a strict hierarchy of expressiveness with respect to weak bisimulation for the three infinite operators considered. In fact, there exist encodings of replication in recursion, and of iteration in replication that preserve weak bisimulation, while the converse does not hold.

To encode replication using recursive definitions, we consider an encoding $[\![\ ]\!]$ that is homomorphic except for replication:

$$[\![!P]\!] \;=\; D \qquad \text{with } D \stackrel{def}{=} [\![P]\!]|D.$$

In order to model iteration using replication, it is simply necessary to spawn replicas only on termination of the previous one. This can be done by detecting the termination of the execution of one iteration before activating the subsequent one. The idea is to use new fresh channel names in such a way that when a process terminates, it communicates termination on the corresponding channel, which then triggers the subsequent iteration. Formally, we consider the encoding $[\![P]\!] = (\nu x)([\![P]\!]_x)$, where $[\![\ ]\!]_x$ is an encoding function indexed on a fresh name x (that is, $x \notin fn(P)$). The name x indicates the channel on

https://doi.org/10.1017/S096012950999017X Published online by Cambridge University Press

which termination should be communicated. The indexed encoding is defined as follows:

$$\begin{aligned}
[\![\mathbf{0}]\!]_x &= \overline{x} \\
[\![\alpha.P]\!]_x &= \alpha.[\![P]\!]_x \\
[\![P+Q]\!]_x &= [\![P]\!]_x + [\![Q]\!]_x \\
[\![P|Q]\!]_x &= (\nu\, y,z)(y.z.\overline{x} \mid [\![P]\!]_y \mid [\![Q]\!]_z) \qquad y,z \notin fn(P) \cup fn(Q) \\
[\![(\nu y)P]\!]_x &= (\nu y)[\![P]\!]_x \\
[\![P^*]\!]_x &= (\nu y)(\overline{y} \mid !y.([\![P]\!]_y + \overline{x})) \qquad y \notin fn(P).
\end{aligned}$$

The encodings in the opposite direction do not exist. Replication cannot be encoded in terms of iteration because weak bisimulation is decidable only under iteration; recursion cannot be encoded in terms of replication because barb is decidable only under replication and weak bisimulation preserves barbs.

6.1. *Related work*

In Giambiagi *et al.* (2004), Giambagi, Schneider and Valencia considered other infinite operators in the setting of CCS, namely recursive expressions with static binding and parameterless constants with dynamic binding. The former is proved to be as expressive as replication, while the latter is proved to be as expressive as constants with parameters. In their approach, two calculi are equally expressive if there exists a weak bisimulation preserving encoding of one calculus in the other, and *vice versa*. Giambiagi *et al.* (2004) left the existence of a weak bisimulation preserving encoding from recursion to replication as an open problem. In Busi *et al.* (2004), one of the two conference papers on which this paper is based, we closed this open problem by proving that there can be no such encoding.

A comparison of different mechanisms for describing infinite behaviours is reported in Nielsen *et al.* (2002), where the expressive power of several timed concurrent constraint languages is investigated. In particular, one of the results in that paper shows that the language with replication is strictly less expressive than the language with recursive definitions of processes (when process constants have parameters). Because of the very different underlying computational model, the proof techniques exploited in that paper cannot be applied directly in the context of CCS.

The undecidability of weak bisimulation has been proved by Srba (Srba 2003) for PA processes. PA is a minimal process algebra comprising sequential and parallel composition, and recursion. Even though PA is minimal, Srba's result cannot be applied in our setting because PA considers sequential composition, which is more general than our prefix operator. The converse does not hold either (that is, we cannot apply our undecidability results directly to PA) because we consider the restriction operator, which is not part of PA.

Acknowledgments

We would like to thank Lucia Acciai for pointing out that the definition given in a preliminary version of this paper for the function $basic_\alpha$ in Definition 23 was incomplete as it did not manage the case that is now handled by the auxiliary $synchbasic_\alpha$ function.

https://doi.org/10.1017/S096012950999017X Published online by Cambridge University Press

We would also like to thank the anonymous referees for their comments, which helped us improve the presentation of this paper.

References

Busi, N., Gabbrielli, M. and Zavattaro, G. (2003) Replication vs. Recursive Definitions in Channel Based Calculi. In: Proc. ICALP'03. *Springer-Verlag Lecture Notes in Computer Science* **2719** 133–144.

Busi, N., Gabbrielli, M. and Zavattaro, G. (2004) Comparing Recursion, Replication, and Iteration in Process Calculi. In: Proc. ICALP'04. *Springer-Verlag Lecture Notes in Computer Science* **3142** 307–319.

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

Giambiagi, P., Schneider, G. and Valencia, F. D. (2004) On the Expressiveness of CCS-like Calculi. In: Proceedings of FOSSACS 04. *Springer-Verlag Lecture Notes in Computer Science* **2987** 226–240.

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

Kanellakis, P. C. and Smolka, S. A. (1990) CCS expressions, finite state processes, and three problems of equivalence. *Information and Computation* **86** (1) 43–68.

Milner, R. (1989) *Communication and Concurrency*, Prentice-Hall.

Milner, R., Parrow, J. and Walker, D. (1992) A calculus of mobile processes. *Journal of Information and Computation* **100** 1–77.

Minsky, M. L. (1967) *Computation: finite and infinite machines*, Prentice-Hall.

Nielsen, M., Palamidessi, C. and Valencia, F. D. (2002) On the Expressive Power of Temporal Concurrent Constraint Programming Languages. *Proc. of PPDP'02,* ACM Press.

Shepherdson, J. C. and Sturgis, J. E. (1963) Computability of recursive functions. *Journal of the ACM* **10** 217–255.

Srba, J. (2003) Undecidability of Weak Bisimilarity for PA-Processes. In: Proc. of DLT'02. *Springer-Verlag Lecture Notes in Computer Science* **2450** 197–208.

https://doi.org/10.1017/S096012950999017X Published online by Cambridge University Press