On the Expressive Power of
Process Interruption and Compensation⋆
(extended abstract)

Mario Bravetti and Gianluigi Zavattaro

Dipartimento di Scienze dell'Informazione, Università di Bologna,
Mura A.Zamboni 7, I-40127 Bologna, Italy.
`bravetti,zavattar@cs.unibo.it`

Abstract. The investigation of the foundational aspects of linguistic mechanisms for programming long running transactions (such as the `scope` operator of WS-BPEL) has recently renewed the interest in process algebraic operators that *interrupt* the execution of one process, replacing it with another one called the *compensation*. We investigate the expressive power of two of such operators, the *interrupt* operator of CSP and the *try-catch* operator for exception handling. We consider two non Turing powerful fragments of CCS (without restriction and relabeling, but with either replication or recursion). We show that the addition of such operators strictly increases the expressive power of the calculi. The calculi with replication and either interrupt or try-catch turn out to be *weakly* Turing powerful (Turing Machines can be encoded but only nondeterministically). The calculus with recursion is weakly Turing powerful when extended with interrupt, but it is Turing complete (Turing Machine can be modeled deterministically) when extended with try-catch.

1 Introduction

The investigation of the foundational aspects of the so-called service composition languages (see, e.g., WS-BPEL [OAS03] and WS-CDL [W3C04]) has recently attracted the attention of the concurrency theory community. In particular, one of the main novelties of such languages is concerned with primitives for programming *long running transactions*. These primitives permit, on the one hand, to interrupt processes when some unexpected failure occur and, on the other hand, to activate alternative processes responsible to *compensate* those activities that, even if completed, must be undone due to the failure of other related activities.

Several recent papers propose process calculi that include operators for process interruption and compensation. Just to mention a few, we recall StAC [BF04], cJoin [BMM04], cCSP [BHF03], π_t [BLZ03], SAGAS [BMM05], web-pi [LZ05], ORC [MC07], SCC [BB$^+$06], COWS [LPT07], and the Conversation Calculus [VCS08]. This huge amount of calculi, including process interruption and compensation as first-class operators, is the pragmatic proof that traditional

---

basic process calculi (that do not include neither process interruption nor compensation) are not completely adequate when one wants to perform a formal investigation of long running transactions, or of fault and compensation handling in languages for service composition.

The aim of this paper is to formally investigate the expressiveness boundary between traditional process calculi and the mechanisms for process interruption and compensation. Instead of performing our investigation on yet another new calculus, we consider standard CCS [Mil89] extended with process interruption and compensation operators taken from the tradition of either process algebras or programming languages. Namely, we consider the *interrupt* operator of CSP [Hoa85] and the *try-catch* operator for exception handling from languages such as C++ or Java. The interrupt operator $P \triangle Q$ executes P until Q executes its first action; when Q starts executing, the process P is definitely interrupted. The try-catch operator $\mathtt{try}\,P\,\mathtt{catch}\,Q$ executes P, but if P performs a *throw* action it is definitely interrupted and Q is executed instead.

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

If we assume that the main activity $\mathtt{P}$ communicates internal failure with the action $\overline{throw}$[1] and completion with $\overline{end}$, and the request for compensation corresponds with the action $\overline{x}$, we can model the behaviour of $\mathtt{scope_x(P,F,C)}$ with both the interrupt:

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

and the try-catch operator:

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

where the vertical bar means parallel composition.

Even if the two considered operators are apparently very similar, we prove an important expressiveness gap between them. More precisely, we consider two non Turing complete fragments of CCS, that we call $CCS_!$ and CCS_{rec}, corresponding to CCS without restriction and relabeling, but with replication or recursion, respectively. We have chosen these two language because, even if not Turing complete, they are expressive enough to model communicating processes (performing input and output operations as in standard service communication) with an infinite behaviour described by means of the two traditional operators of process

---

[1] We use the typical notation of process calculi: an overlined action (e.g. $\overline{a}$) is complementary with the corresponding non-overlined one (e.g. action a), and complementary actions allows parallel processes to synchronize.

algebras: recursion as in CCS [Mil89] or replication as in π-calculus [MPW92]. We extend these calculi with either the interrupt operator (obtaining the calculi that we call $CCS_!^{\triangle}$ and $CCS_{rec}^{\triangle}$, respectively) or the try-catch operator (obtaining $CCS_!^{\text{tc}}$ and CCS_{rec}^{tc}, respectively). We prove that the four obtained extensions are strictly more expressive than the two original basic calculi. The two extensions $CCS_!^{\triangle}$ and $CCS_!^{\text{tc}}$ of the calculus with replication, as well as the calculus $CCS_{rec}^{\triangle}$ with recursion and interrupt, are *weakly* Turing powerful. By weakly Turing powerful, we mean that Turing Machines can be modeled but only in a nondeterministic manner, i.e., a Turing Machine terminates if and only if the corresponding modeling in the calculus has a terminating computation. On the other hand, the calculus CCS_{rec}^{tc} with recursion and try-catch is Turing complete as it permits also the deterministic modeling of Turing Machines.

In order to prove these results we investigate the decidability of *convergence* and *termination* in the considered calculi. By convergence we mean the existence of at least one terminating computation, by termination we mean that all computations terminate. For the weakly Turing powerful calculi, we first prove that convergence is undecidable showing the existence of a nondeterministic modeling of Random Access Machines (RAMs) [Min67], a well known register based Turing complete formalism. Then, we prove that termination is decidable resorting to the theory of well structured transition systems [FS01]. The decidability of termination proves the impossibility to model deterministically any Turing powerful formalism. On the other hand, for the Turing complete calculi we present a deterministic modeling of RAMs.

The most significant technical contribution of this paper concerns the proof of decidability of termination in $CCS_{rec}^{\triangle}$. This because, while proving decidability of termination in $CCS_!^{\text{tc}}$ is done by resorting to the approach in [BGZ03], proving termination in $CCS_{rec}^{\triangle}$ requires introducing an order over terms with an unbounded nesting depth of the interrupt operators. For this reason we need to resort to a completely different technique which is based on devising a particular transformation of terms into *trees* (of unbounded depth) and considering an ordering on such trees. The particular transformation devised must be "tuned" in such a way that the ordering obtained is: from the one hand a well quasi ordering (and to prove this we exploit the Kruskal Tree theorem [Kru60]), from the other hand strongly compatible with the operational semantics. Obtaining and proving the latter result is particularly intricate and it also requires us to slightly modify the operational semantics of the interruption operator in a termination preserving way and to technically introduce different kinds of trees on subterms and contexts in order to interpret transitions on trees.

The paper is structured as follows. In Section 2 we define the considered calculi. In Section 3 we show the undecidability of convergence in $CCS_!^{\triangle}$ and $CCS_!^{\text{tc}}$ (hence the same trivially holds also in $CCS_{rec}^{\triangle}$ and CCS_{rec}^{tc}). In Section 4 we show the undecidability of termination in CCS_{rec}^{tc}. Section 5 is dedicated to showing decidability of termination for $CCS_!^{\text{tc}}$ and $CCS_{rec}^{\triangle}$ (hence the same trivially holds also for $CCS_!^{\triangle}$). In Section 6 we draw some conclusive remarks. Due to space limitation the proofs are omitted, the details are available in [BZ08].

$$\alpha.P \xrightarrow{\alpha} P \qquad\qquad \frac{P \xrightarrow{\alpha} P'}{P|Q \xrightarrow{\alpha} P'|Q}$$

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

Table 1. The transition system for finite core CCS (symmetric rules of `PAR` and `SUM` omitted).

2   The Calculi

We start considering the fragment of CCS [Mil89] without recursion, restriction, and relabeling (that we call finite core CCS or simply finite CCS). After we present the two infinite extensions with either replication or recursion, the new *interrupt operator*, and finally the *try-catch* operator.

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

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

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

For input and output actions α, i.e. $\alpha \neq \tau$, we write $\overline{\alpha}$ for the complementary of α; that is, if $\alpha = x$ then $\overline{\alpha} = \overline{x}$, if $\alpha = \overline{x}$ then $\overline{\alpha} = x$. The channel names that occur in P are denoted with $n(P)$. The names in a label α, written $n(\alpha)$ is the set of names in α, i.e. the empty set if $\alpha = \tau$ or the singleton $\{x\}$ if α is either x or $\overline{x}$.

Table 1 contains the set of the transition rules for finite core CCS.

Definition 2. (CCS$_!$) *The class of CCS$_!$ processes is defined by adding the production $P \ ::= \ !\alpha.P$ to the grammar of Definition 1.*

The transition rule for replication is

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

We consider a guarded version of replication in which the replicated process is in prefix form. We make this simplification in order to have a finitely branching transition system, that allows us to apply directly the theory of well structured

transition system in order to prove the decidability of termination. Neverthe-less, the proof discussed in Section 5 can be extended also to general replication exploiting an auxiliary transition system which is finitely branching and termi-nation equivalent to the initial transition system. This transition system can be obtained using standard techniques (see, e.g., [BGZ03,BGZ08]).

Definition 3. ($\mathbf{CCS}_{rec}$) *We assume a denumerable set of process variables, ranged over by X. The class of CCS_{rec} processes is defined by adding the pro-ductions $P ::= X \mid recX.P$ to the grammar of Definition 1. In the process $recX.P$, $recX$ is a binder for the process variable X and P is the scope of the binder. We consider (weakly) guarded recursion, i.e., in the process $recX.P$ each occurrence of X (which is free in P) occurs inside a subprocess of the form $\alpha.Q$.*

The transition rule for recursion is

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

where $P\{recX.P/X\}$ denotes the process obtained by substituting $recX.P$ for each free occurrence of X in P, i.e. each occurrence of X which is not inside the scope of a binder $recX$. Note that $CCS_!$ is equivalent to a fragment of CCS_{rec}. In fact, the replication operator $!\alpha.P$ of $CCS_!$ is equivalent to the recursive process $recX.\big(\alpha.(P|X)\big)$.

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

Definition 4. ($\mathbf{CCS}_!^{\triangle}$ and $\mathbf{CCS}_{rec}^{\triangle}$) *The class of $CCS_!^{\triangle}$ and $CCS_{rec}^{\triangle}$ processes is defined by adding the production $P ::= P\triangle P$ to the grammars of Definition 2 and Definition 3, respectively.*

The transition rules for the interrupt operator are

$$\frac{P \stackrel{\alpha}{\longrightarrow} P'}{P\triangle Q \stackrel{\alpha}{\longrightarrow} P'\triangle Q} \qquad \frac{Q \stackrel{\alpha}{\longrightarrow} Q'}{P\triangle Q \stackrel{\alpha}{\longrightarrow} Q'}$$

We complete the list of definitions of the considered calculi presenting the extensions with the new *try-catch* operator.

Definition 5. ($\mathbf{CCS}_!^{\mathtt{tc}}$ and $\mathbf{CCS}_{rec}^{\mathtt{tc}}$) *The class of $CCS_!^{\mathtt{tc}}$ and $CCS_{rec}^{\mathtt{tc}}$ processes is defined by adding the productions $P ::= \mathtt{try}\,P\,\mathtt{catch}\,P$ and $\alpha ::= \mathtt{throw}$ to the grammars of Definition 2 and Definition 3, respectively. The new action $\mathtt{throw}$ is used to model the raising of an exception.*

The transition rules for the try-catch operator are

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

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

In the following we will consider only closed processes, i.e. processes without free occurrences of process variables. Given a closed process Q, its internal runs $Q \longrightarrow Q_1 \longrightarrow Q_2 \longrightarrow \dots$ are given by its reduction steps, (denoted with $\longrightarrow$), i.e. by those transitions $\longrightarrow$ that the process can perform in isolation, independently of the context. The internal transitions $\longrightarrow$ correspond to the transitions labeled with τ, i.e. $P \longrightarrow P'$ iff $P \overset{\tau}{\longrightarrow} P'$. We denote with $\longrightarrow^+$ the transitive closure of $\longrightarrow$, while $\longrightarrow^*$ is the reflexive and transitive closure of $\longrightarrow$.

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

3 Undecidability of Convergence in $\mathrm{CCS}_!^{\triangle}$ and $\mathrm{CCS}_!^{\mathsf{tc}}$

We prove that $\mathrm{CCS}_!^{\triangle}$ and $\mathrm{CCS}_!^{\mathsf{tc}}$ are powerful enough to model, at least in a nondeterministic way, any Random Access Machine [SS63] (RAM), a well known register based Turing powerful formalism.

A RAM (denoted in the following with R) is a computational model composed of a finite set of registers $r_1, \dots, r_n$, that can hold arbitrary large natural numbers, and by a program composed by indexed instructions $(1 : I_1), \dots, (m : I_m)$, 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 $(i, c_1, \dots, c_n)$ where i is the program counter indicating the next instruction to be executed, and $c_1, \dots, c_n$ are the current contents of the registers $r_1, \dots, r_n$, respectively. Given a configuration $(i, c_1, \dots, c_n)$, its computation proceeds by executing the instructions in sequence, unless a jump instruction is encountered. The execution stops when an instruction number higher than the length of the program is reached. Note that the computation of the RAM proceeds deterministically (it does not exhibit non-deterministic behaviors).

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

In [Min67] it is shown that the following two instructions are sufficient to model every recursive function:

- $(i : Succ(r_j))$: adds 1 to the contents of register r_j;

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

Our encoding is nondeterministic because it introduces computations which do not follow the expected behavior of the modeled RAM. However, all these computations are infinite. This ensures that, given a RAM, its modeling has a terminating computation if and only if the RAM terminates. This proves that *convergence* is undecidable.

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

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

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

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

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

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

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

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

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

We now show how to model the registers using either the interruption or the try-catch operators. In both cases we exploit the following idea. Every time the register r_j is incremented, a dec_j process is spawned which permits the

subsequent execution of a corresponding decrement operation. In case of test for zero on the register r_j, we will exploit either the interruption or the try-catch operators in order to remove all the active processes dec_j, thus resetting the register. If the register is not empty when it is reset, the computation of the encoding does not reproduce the RAM computation any longer. Nevertheless, such "wrong" computation surely does not terminate, thus we can conclude that we faithfully model at least the terminating computations. Divergence in case of "wrong" reset is guaranteed by the fact that if the register is not empty, k instances of dec_j processes are removed with $k > 0$, and k instances of the process $\overline{loop}$ (previously produced by the corresponding k increment operations) will never be removed.

As discussed above, the presence of $\overline{loop}$ processes prevents the encoding from converging. This is guaranteed by considering, e.g., the following divergent process

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

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

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

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

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

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

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

Definition 6. *Let R be a RAM with program instructions $(1 : I_1), \ldots, (m : I_m)$ and registers $r_1, \ldots, r_n$. Let also Γ be either $\triangle$ or $\texttt{tc}$. Given the configuration $(i, c_1, \ldots, c_n)$ of R we define*

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

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

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

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

1. *if $(i, c_1, \ldots, c_n) \to_R (i', c_1', \ldots, c_n')$ then we have $[\![(i, c_1, \ldots, c_n)]\!]_R^\Gamma \to^+ [\![(i', c_1', \ldots, c_n')]\!]_R^\Gamma$*
2. *if $[\![(i, c_1, \ldots, c_n)]\!]_R^\Gamma \longrightarrow Q_1 \longrightarrow Q_2 \longrightarrow \cdots \longrightarrow Q_l$ is a, possibly zero-length, internal run of $[\![(i, c_1, \ldots, c_n)]\!]_R^\Gamma$ then one of the following holds:*
   - *there exists k, with $1 \leq k \leq l$, such that $Q_k \equiv [\![(i', c_1', \ldots, c_n')]\!]_R^\Gamma$, with $(i, c_1, \ldots, c_n) \to_R (i', c_1', \ldots, c_n')$;*
   - *$Q_l \longrightarrow^+ [\![(i', c_1', \ldots, c_n')]\!]_R^\Gamma$, with $(i, c_1, \ldots, c_n) \to_R (i', c_1', \ldots, c_n')$;*
   - *Q_l does not converge.*

Corollary 1. *Let R be a RAM. We have that the RAM R terminates if and only if $[\![(1, 0, \ldots, 0)]\!]_R^\Gamma$ converges (for both $\Gamma = \triangle$ and $\Gamma = \mathtt{tc}$).*

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

4 Undecidability of Termination in $\mathbf{CCS}_{rec}^{\mathtt{tc}}$

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

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

Let R be a RAM with registers $r_1, \ldots, r_n$, and instructions $(1 : I_1), \ldots, (m : I_m)$. We start presenting the modeling of the instructions which is similar to the encoding presented in the previous section. Note that here the assumption on

registers to all have value 0 in a terminating configuration is not needed. We encode each instruction $(i : I_i)$ with the process $[\![(i : I_i)]\!]$, which is a shorthand for the following process

$$
\begin{aligned}
[\![(i : I_i)]\!] &: recX.p_i.\overline{inc_j}.\overline{p_{i+1}}.X & \text{if } I_i = Succ(r_j) \\
[\![(i : I_i)]\!] &: recX.p_i.(\ \overline{zero_j}.\overline{p_s}.X\ +\ \overline{dec_j}.ack.\overline{p_{i+1}}.X\ ) & \text{if } I_i = DecJump(r_j, s)
\end{aligned}
$$

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

The register r_j, that we assume initially empty, is modeled by the process $[\![r_j = 0]\!]$ which is a shorthand for the following process (to simplify the notation we use also the shorthand R_j defined below)

$$
\begin{aligned}
[\![r_j = 0]\!] &: recX.\big(zero_j.X\ +\ inc_j.\mathtt{try}\,R_j\,\mathtt{catch}\,(\overline{ack}.X)\big) \\
R_j &: recY.\big(dec_j.\mathtt{throw}\ +\ inc_j.\mathtt{try}\,Y\,\mathtt{catch}\,(\overline{ack}.Y)\big)
\end{aligned}
$$

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

Formally, we have that the register r_j with contents $c_j > 0$ is modeled by the following process composed of the nesting of c_j try-catch operators

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

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

Definition 7. *Let R be a RAM with program instructions $(1 : I_1), \dots, (m : I_m)$ and registers $r_1, \dots, r_n$. Given the configuration $(i, c_1, \dots, c_n)$ we define with*

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

the encoding of the RAM R in $CCS_{rec}^{\mathtt{tc}}$.

The new encoding faithfully reproduces the behavior of a RAM as stated by the following proposition. In the following Proposition we use the notion of *deterministic* internal run defined as follows: an internal run $P_0 \longrightarrow P_1 \longrightarrow \ldots \longrightarrow P_l$ is deterministic if for every process P_i, with $i < l$, P_{i+1} is the unique process Q such that $P_i \longrightarrow Q$.

Proposition 2. *Let R be a RAM with program instructions $(1 : I_1), \ldots, (m : I_m)$ and registers $r_1, \ldots, r_n$. Given a configuration $(i, c_1, \ldots, c_n)$ of R, we have that, if $i > m$ then $[\![(i, c_1, \ldots, c_n)]\!]_R$ is a dead process, otherwise:*

1. *if $(i, c_1, \ldots, c_n) \to_R (i', c_1', \ldots, c_n')$ then we have $[\![(i, c_1, \ldots, c_n)]\!]_R \to^+ [\![(i', c_1', \ldots, c_n')]\!]_R$*
2. *there exists a non-zero length deterministic internal run $[\![(i, c_1, \ldots, c_n)]\!]_R^\Gamma \longrightarrow Q_1 \longrightarrow Q_2 \longrightarrow \cdots \longrightarrow [\![(i', c_1', \ldots, c_n')]\!]_R^\Gamma$ such that $(i, c_1, \ldots, c_n) \to_R (i', c_1', \ldots, c_n')$.*

Corollary 2. *Let R be a RAM. We have that the RAM R terminates if and only if $[\![(1, 0, \ldots, 0)]\!]_R$ terminates.*

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

5 Decidability of Termination in $CCS_!^{\mathtt{tc}}$ and $CCS_{rec}^{\triangle}$

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

These results are based on the theory of well-structured transition systems [FS01]. We start recalling some basic definitions and results concerning well-structured transition systems, that will be used in the following.

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

Definition 8. *A* well-quasi-ordering *(wqo) is a quasi-ordering $\leq$ over a set $\mathcal{S}$ such that, for any infinite sequence $s_0, s_1, s_2, \ldots$ in $\mathcal{S}$, there exist indexes $i < j$ such that $s_i \leq s_j$.*

Transition systems can be formally defined as follows.

Definition 9. *A* transition system *is a structure $TS = (\mathcal{S}, \to)$, where $\mathcal{S}$ is a set of* states *and $\to \subseteq \mathcal{S} \times \mathcal{S}$ is a set of* transitions. *We write $Succ(s)$ to denote the set $\{s' \in \mathcal{S} \mid s \to s'\}$ of immediate successors of S. TS is* finitely branching *if all $Succ(s)$ are finite.*

Well-structured transition system, defined as follows, provide the key tool to decide properties of computations.

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

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

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

The following theorem (a special case of a result in [FS01]) will be used to obtain our decidability results.

Theorem 1. *Let $(\mathcal{S}, \rightarrow, \leq)$ be a finitely branching, well-structured transition system with strong compatibility, decidable $\leq$ and computable Succ. The existence of an infinite computation starting from a state $s \in \mathcal{S}$ is decidable.*

The proof of decidability of termination in $CCS_{rec}^{\triangle}$ is not done on the original transition system, but on a termination equivalent one. The new transition system does not eliminate interrupt operators during the computation; in this way, the nesting of interrupt operators can only grow and do not shrink. As we will see, this transformation will be needed for proving that the ordering that we consider on processes is strongly compatible with the operational semantics. Formally, we define the new transition system $\overset{\alpha}{\longmapsto}$ for $CCS_{rec}^{\triangle}$ considering the transition rules of Definition 3 (where $\overset{\alpha}{\longmapsto}$ is substituted for $\overset{\alpha}{\longrightarrow}$) plus the following rules

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

Notice that the first of the above rules is as in Definition 4, while the second one is different because it does not remove the $\triangle$ operator.

As done for the standard transition system, we assume that the reductions $\longmapsto$ of the new semantics corresponds to the τ–labeled transitions $\overset{\tau}{\longmapsto}$. Also for the new semantics, we say that a process P terminates if and only if all its computations are finite, i.e. it cannot give rise to an infinite sequence of reductions $\longmapsto$.

Proposition 3. *Let $P \in CCS_{rec}^{\triangle}$. Then P terminates according to the semantics $\longrightarrow$ iff P terminates according to the new semantics $\longmapsto$.*

We now separate in two subsections the proofs of decidability of termination in $CCS_!^{\mathsf{tc}}$ and in $CCS_{rec}^{\triangle}$.

5.1 Termination is decidable in $(CCS_!^{\mathsf{tc}}, \longrightarrow)$

The proof for $CCS_!^{\mathsf{tc}}$ is just a rephrasal of the proof of decidability of termination in CCS without relabeling and with replication instead of recursion reported in [BGZ08].

We define for $(CCS_!^{\mathsf{tc}}, \longrightarrow)$ a quasi-ordering on processes which turns out to be a well-quasi-ordering compatible with $\longrightarrow$. Thus, exploiting Theorem 1 we show that termination is decidable.

Definition 11. *Let* $P \in CCS_!^{\mathsf{tc}}$. *With* $Deriv(P)$ *we denote the set of processes reachable from* P *with a sequence of reduction steps:*

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

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

Definition 12. *We define* $\equiv$ *as the least congruence relation satisfying the following axioms:* $\qquad P|Q \equiv Q|P \qquad P|(Q|R) \equiv (P|Q)|R \qquad P|\mathbf{0} \equiv P$

Now we are ready to define the quasi-ordering on processes:

Definition 13. *Let* $P, Q \in CCS_!^{\mathsf{tc}}$. *We write* $P \preceq Q$ *iff there exist* n, P', R, $P_1, \ldots, P_n$, $Q_1, \ldots, Q_n$, $S_1, \ldots, S_n$ *such that* $P \equiv P'|\prod_{i=1}^{n} \mathtt{try}\, P_i\, \mathtt{catch}\, S_i$, $Q \equiv P'|R|\prod_{i=1}^{n} \mathtt{try}\, Q_i\, \mathtt{catch}\, S_i$, *and* $P_i \preceq Q_i$ *for* $i = 1, \ldots, n$.

Theorem 2. *Let* $P \in CCS_!^{\mathsf{tc}}$. *Then the transition system* $(Deriv(P), \longrightarrow, \preceq)$ *is a finitely branching well-structured transition system with strong compatibility, decidable* $\preceq$ *and computable Succ.*

Corollary 3. *Let* $P \in CCS_!^{\mathsf{tc}}$. *The termination of process* P *is decidable.*

5.2 Termination is decidable in $(CCS_{rec}^{\triangle}, \longmapsto)$

According to the ordering defined in Definition 13, we have that $P \preceq Q$ if Q has the same structure of nesting of try-catch operators and it is such that in each point of this nesting Q contains at least the same processes (plus some other processes in parallel). This is a well-quasi-ordering in the calculus with replication because, given P, it is possible to compute an upper bound to the number of nesting in any process in $Deriv(P)$. In the calculus with recursion this upper bound does not exist as recursion permits to generate nesting of unbounded depth (this e.g. is used in the deterministic RAM modeling of Section 4). For this reason, we need to move to a different ordering inspired by the ordering on trees used by Kruskal in [Kru60]. This allows us to use the Kruskal Tree theorem that states that the trees defined on a well quasi ordering is a well quasi ordering.

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

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

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

Definition 14. *Let $P \in \mathcal{P}$. We define the flattened parallel components of P, $FPAR(P)$, as the sequence over $\mathcal{P}_{seq} \cup \mathcal{P}_{int}$ given by*

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

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

In the following we map processes into ordered trees (with both a left to right ordering of children at every node and the usual son to father ordering).

Definition 15. *A tree t over a set E is a partial function from $\mathbf{N}^*$ to E such that $dom(t)$ is finite, is closed with respect to sequence prefixing and is such that $\vec{n}; m \in dom(t)$ and $m' \le m$, with $m' \in \mathbf{N}$, implies $\vec{n}; m' \in dom(t)$.*

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

Let $\mathcal{P}_{rint} = \{\triangle Q \mid Q \in \mathcal{P}\}$ be a set representing compensations.

Definition 16. *Let $P \in \mathcal{P}$. We define the tree of P, $TREE(P)$, as the minimal tree $TREE(P)$ over $\mathcal{P}_{seq}^* \cup \mathcal{P}_{rint}$ (and minimal auxiliary tree $TREE^{odd}(P')$ over $\mathcal{P}_{seq}^* \cup \mathcal{P}_{rint}$, with $P' \in \mathcal{P}_{int}$) satisfying*

$$(\epsilon, FSEQ(P)) \in TREE(P)$$
$$(\vec{n}, l) \in TREE^{odd}(FINT(P)_i) \text{ implies } (i; \vec{n}, l) \in TREE(P)$$

$$(\epsilon, \triangle Q) \in TREE^{odd}(P' \triangle Q)$$
$$(\vec{n}, l) \in TREE(P') \text{ implies } (1; \vec{n}, l) \in TREE^{odd}(P' \triangle Q)$$

Example 2. The tree of the process $a + b|\big((recX.(a.X|c))\triangle Q\big)|c|((a|c)\triangle S)$ for some processes Q and S is $\{(\epsilon, a+b; c), (1, \triangle Q), (1; 1, recX.(a.X|c)), (2, \triangle S), (2; 1, a; c)\}$.

In the following, we define the ordering between processes by resorting to the ordering on trees used in [Kru60] applied to the particular trees obtained from processes by our transformation procedure. In particular, in order to do this we introduce the notion of injective function that strictly preserves order inside trees: a possible formal way to express homeomorphic embedding between trees, used in the Kruskal's theorem [Kru60], that we take from [Sim85].

Let t be a tree. We take $\leq_t$ to be the ancestor pre-order relation inside t, defined by: $\vec{n} \leq_t \vec{m}$ iff $\vec{m}$ is a prefix $\vec{n}$ (or $\vec{m} = \vec{n}$). Moreover, we take $\wedge_t$ to be the minimal common ancestor of a pair of nodes, i.e. $\vec{n_1} \wedge_t \vec{n_2} = min\{\vec{m} | \vec{n_1} \leq_t \vec{m} \wedge \vec{n_2} \leq_t \vec{m}\}$.

Definition 17. *We say that an injective function φ from $dom(t)$ to $dom(t')$ strictly preserves order inside trees iff for every $\vec{n}, \vec{m} \in dom(t)$ we have:*

- *$\vec{n} \leq_t \vec{m}$ implies $\varphi(\vec{n}) \leq_{t'} \varphi(\vec{m})$*
- *$\varphi(\vec{n} \wedge_t \vec{m}) = \varphi(\vec{n}) \wedge_{t'} \varphi(\vec{m})$*

Definition 18. *Let $P, Q \in \mathcal{P}$. $P \preceq Q$ iff there exists an injective function φ from $dom(TREE(P))$ to $dom(TREE(Q))$ such that φ strictly preserves order inside trees and for every $\vec{n} \in dom(\varphi)$:*

- *either there exists $R \in \mathcal{P}$ such that $TREE(P)(\vec{n}) = TREE(Q)(\varphi(\vec{n})) = \triangle R$*
- *or $TREE(P)(\vec{n}), TREE(Q)(\varphi(\vec{n})) \in \mathcal{P}^*_{seq}$ and, if $len(TREE(P)(\vec{n})) > 0$, there exists an injective function f from $\{1, \ldots, len(TREE(P)(\vec{n}))\}$ to $\{1, \ldots, len(TREE(Q)(\varphi(\vec{n})))\}$ such that for every $i \in dom(f): TREE(P)(\vec{n})_i = TREE(Q)(\varphi(\vec{n}))_{f(i)}$.*

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

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

Definition 19. *Let $P \in CCS^\triangle_{rec}$. With $Deriv(P)$ we denote the set of processes reachable from P with a sequence of reduction steps:*

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

We are now ready to state our main result, that can be proved by contemporaneously exploiting Higman's Theorem on sequences [Hig52] and Kruskal's Theorem on trees [Kru60].

Theorem 3. *Let $P \in CCS^\triangle_{rec}$. Then the transition system $(Deriv(P), \longmapsto, \preceq)$ is a finitely branching well-structured transition system with strong compatibility, decidable $\preceq$ and computable Succ.*

Corollary 4. *Let $P \in CCS^\triangle_{rec}$. The termination of process P is decidable.*

As replication is a particular case of recursion, we have that the same decidability result holds also for $CCS^\triangle_!$.

6 Conclusion and Related Work

Following a recent trend of research devoted to the investigation of the foundational properties of languages for service oriented computing by means of process calculi including mechanisms for process interruption and compensation (see, e.g., [BF04,BMM04,BHF03,BLZ03,BMM05,LZ05,MC07,BB$^+$06,LPT07,VCS08]), we have investigated the expressive power of two basic operators for process interruption and compensation taken from the tradition of either process algebras or programming languages. Namely, we have considered the *interrupt* operator of CSP [Hoa85] and the *try-catch* construct of languages such as C++ or Java.

We have formalized an expressiveness gap between the traditional input-output communication primitives of process algebras and the considered operators. Formally, we have proved that CCS [Mil89] without restriction and relabeling, and with replication instead of recursion (which is not Turing complete) turns out to be weakly Turing powerful when extended with the considered operators. On the other hand, the same fragment of CCS with recursion instead of replication (which is still non Turing complete) turns out to be weakly Turing powerful when extended with the interrupt operator, while it is Turing complete when extended with try-catch.

It is worth to compare the results proved in this paper with similar results presented in [BGZ03]. In that paper, the interplay between replication/recursion and restriction is studied: a fragment of CCS with restriction and replication is proved to be weakly Turing powerful, while the corresponding fragment with recursion is proved to be Turing complete. This result is similar to what we have proved about the interplay between replication/recursion and the try-catch operator. This proves a strong connection between restriction and try-catch, at least as far as the computational power is concerned. Intuitively, this follows from the fact that, similarly to restriction, the try-catch operator defines a new scope for the special `throw` action which is bound to a specific exception handler. On the contrary, the interrupt operator does not have the same computational power. In fact, the calculus with recursion and interrupt is only weakly Turing powerful. This follows from the fact that this operator does not provide a similar binding mechanism between the interrupt signals and the interruptible processes.

It is worth to compare our criterion for the evaluation of the expressive power with the criterion used by Palamidessi in [Pal03] to discriminate the expressive power of the synchronous and the asynchronous π-calculus. Namely, in that paper, it is proved that there exists no modular embedding of the synchronous into the asynchronous π-calculus that preserves any reasonable semantics. When we prove that termination (resp. convergence) is undecidable in one calculus while it is not in another one, we also prove that there exists no encoding (thus also no modular embedding) of the former calculus into the latter that preserves any semantics sensible to termination (resp. convergence). By semantics sensible to some property, we mean any semantics that distinguishes one process that satisfies the property from one process that does not. If we assume that the termination of one computation is observable (as done for instance in process calculi with explicit termination [BBR08]), we have that any reasonable seman-

tics (according to the notion of reasonable semantics presented in [Pal03]) is sensible to both termination and convergence.

We conclude by mentioning the investigation of the expressive power of the disrupt operator (similar to our interruption operator) done by Baeten and Bergstra in a technical report [BB00]. In that paper, a different notion of expressive power is considered: a calculus is more expressive than another one if it generates a larger set of transition systems. We consider a stronger notion of expressive power: a calculus is more expressive than another one if it supports a more faithful modeling of Turing complete formalisms.

References

[BBR08] J.C.M. Baeten, T. Basten, and M.A. Reniers. *Process algebra (equational theories of communicating processes)*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2008.

[BB00] J.C.M. Baeten, J. Bergstra. *Mode transfer in process algebra*. Report CSR 00-01, Technische Universiteit Eindhoven. This paper is an expanded and revised version of J. Bergstra, A mode transfer operator in process algebra, Report P8808, Programming Research Group, University of Amsterdam, 2000. Available at `http://alexandria.tue.nl/extra1/wskrap/publichtml/200010731.pdf`

[BLZ03] L. Bocchi, C. Laneve, and G. Zavattaro. A calculus for long running transactions. In *FMOODS'03: Proceedings of the 6th IFIP International Conference on Formal Methods for Open Object-based Distributed Systems*, volume 2884 of *LNCS*, pages 124–138. Springer, 2003.

[BB+06] M. Boreale and R. Bruni, L. Caires, R. De Nicola, I. Lanese, M. Loreti, F. Martins, U. Montanari, A. Ravara, D. Sangiorgi, V.T. Vasconcelos, and G. Zavattaro. SCC: A Service Centered Calculus. In *WS-FM 2006: Proceedings of the Third International Workshop on Web Services and Formal Methods*, volume 4184 of *LNCS*, pages 38–57. Springer, 2006.

[BZ08] M. Bravetti and G. Zavattaro. On the Expressive Power of Process Interruption and Compensation. Technical report available at: `http://cs.unibo.it/∼zavattar/papers.html`

[BMM04] R. Bruni, H.C. Melgratti, and U. Montanari. Nested Commits for Mobile Calculi: Extending Join. In *TCS 2004: IFIP 18th World Computer Congress, TC1 3rd International Conference on Theoretical Computer Science*, pages 563-576. Kluwer, 2004.

[BMM05] R. Bruni and H.C. Melgratti, and U. Montanari. Theoretical foundations for compensations in flow composition languages. In *POPL 2005: Proceedings of the 32nd Symposium on Principles of Programming Languages*, pages 209–220, ACM Press, 2005.

[BGZ03] N. Busi, M. Gabbrielli, and G. Zavattaro. Replication vs. Recursive Definitions in Channel Based Calculi. In *ICALP'03: Proceedings of 30th International Colloquium on Automata, Languages and Programming*, volume 2719 of *LNCS*, pages 133–144, Springer, 2003.

[BGZ08] N. Busi, M. Gabbrielli, and G. Zavattaro. On the Expressive Power of Recursion, Replication, and Iteration in Process Calculi. Technical report available at: `http://cs.unibo.it/∼zavattar/papers.html`. Extended version of [BGZ03].

[BF04]   M. Butler and C. Ferreira. An operational semantics for StAC, a language for modelling long-running business transactions. In *COORDINATION'04: Proceedings of the 6th International Conference on Coordination Models and Languages*, volume 2949 of *LNCS*, pages 87–104. Springer, 2004.

[BHF03]   M. Butler, T. Hoare, and C. Ferreira. A trace semantics for long-running transactions. In *Proceedings of 25 Years of Communicating Sequential Processes*, volume 3525 of *LNCS*, pages133–150. Springer, 2005.

[FS01]   A. Finkel and P. Schnoebelen. Well-Structured Transition Systems Everywhere! *Theoretical Computer Science*, 256:63–92, 2001.

[Hig52]   G. Higman. Ordering by divisibility in abstract algebras. In *Proc. London Math. Soc.*, vol. 2, pages 236–366, 1952.

[Hoa85]   T. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[Kru60]   J.B. Kruskal. Well-Quasi-Ordering, The Tree Theorem, and Vazsonyi's Conjecture. Transactions of the American Mathematical Society, vol. 95(2): 210–225, 1960.

[LZ05]   C. Laneve and G. Zavattaro. Foundations of web transactions. In *FOSSACS 2005: Proceedings of the 8th International Conference on Foundations of Software Science and Computational Structures*, volume 3441 of *LNCS*, pages 282–298. Springer, 2005.

[LPT07]   A. Lapadula, R. Pugliese, and F. Tiezzi. A Calculus for Orchestration of Web Services. In *ESOP 2007: Proceedings of 16th European Symposium on Programming*, volume 4421 of *LNCS*, pages 33–47. Springer, 2007.

[Mil89]   R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[MPW92] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes, Part I + II. *Information and Computation*, 100(1):1–77, 1992.

[Min67]   M. L. Minsky. *Computation: finite and infinite machines*. Prentice-Hall, Englewood Cliffs, 1967.

[MC07]   J. Misra and W. R. Cook. Computation Orchestration. *Journal of Software and System Modeling*, volume 6(1): 83-110, 2007.

[OAS03]   OASIS. *WS-BPEL: Web Services Business Process Execution Language Version 2.0*. Technical report, OASIS, 2003.

[Pal03]   C. Palamidessi. Comparing the Expressive Power of the Synchronous and the Asynchronous pi-calculus. In *Mathematical Structures in Computer Science*, 13(5): 685-719, Cambridge University Press, 2003. A short version of this paper appeared in POPL'97.

[SS63]   J. C. Shepherdson and J. E. Sturgis. Computability of recursive functions. *Journal of the ACM*, 10:217–255, 1963.

[Sim85]   S. G. Simpson. Nonprovability of certain combinatorial properties of finite trees. In *Harvey Friedman's Research on the Foundations of Mathematics*, pages 87-117. North-Holland, 1985.

[VCS08]   H.T. Vieira, L. Caires, and J.C. Seco. The Conversation Calculus: A Model of Service-Oriented Computation. In *ESOP 2008: Proceedings of 17th European Symposium on Programming*, volume 4960 of *LNCS*, pages 269–283. Springer, 2008.

[W3C04]   W3C. *WS-CDL: Web Services Choreography Description Language*. Technical report, W3C, 2004.