Prioritized and Parallel Reactions in Shared Data Space Coordination Languages

Nadia Busi and Gianluigi Zavattaro

Dipartimento di Scienze dell'Informazione, Università di Bologna,
Piazza di Porta S. Donato 5, I-40127 Bologna, Italy.
E-mail: {busi,zavattar}@cs.unibo.it

Abstract. Reactive programming has been added to the traditional Linda programming style in order to deal with the dynamic aspects of those applications in which it is important to observe modifications of the environment which occur quickly. Reactive programming is embedded in shared data spaces by defining the *observable* entities and the corresponding *reactions* that usually are processes possibly executing coordination primitives. Typical observable entities are the presence in the space of a datum (state-based) or the execution of a particular primitive (event-based). In this paper we consider different models of reaction execution adopted in the coordination literature so far, namely the *prioritized* model (used e.g. in MARS [3], TuCSoN [8] and LIME [9]) and the *parallel* execution model (considered e.g. in JavaSpaces [13], TSpaces [14] and WCL [11]). Prioritized reactions are usually implemented at server-side as during their execution no other coordination primitives can take place; *parallel* reactions, more suited when reactions are executed at client-side, are executed in parallel with the other processes in the system (thus other coordination primitives can interleave with the reactions). Using a process algebraic setting we perform a rigorous investigation of the relative advantages and disadvantages of these two models for reaction execution.

1 Introduction

The development of Linda-like coordination languages [4] for use over wide-area or mobile networks has driven the consideration of new primitives being added to allow new styles of coordination. One set of such primitives are those that allow reactive-programming, essentially allowing a program to be notified on the availability of a tuple or the execution of a particular primitive. Examples of Linda-like coordination languages including such reactive mechanisms are MARS [3], TuCSoN [8], LIME [9], JavaSpaces [13], TSpaces [14] and WCL [11].

Similarly, in all these languages reactions are processes that possibly execute coordination primitives; the difference is in *what* can activate the reactions and *how* the reactions are executed. We call *observable entities* those entities that can activate a reaction; we call reaction *execution model* the rules governing the execution of the activated reactions.

In this paper we focus on the two different reaction execution models considered in the coordination literature so far. In the *prioritized* model the reactions are executed (activated and completed) immediately after the observation occurs. While reactions are under execution, the other processes in the system cannot access the data space. This model is usually adopted when reactions are executed on server-side, that is within the implementation of the shared data space. On the contrary, in the *parallel* execution model, the reactions are only activated immediately after the observation, and they are executed in parallel with the other processes in the system; in this way, reactions do not block the execution of the other processes in the system. This model is particularly suited when the reactions are executed on client-side, i.e. when the shared data space implementation simply invokes the reactions that are actually executed by the devices hosting the processes accessing the data space. The prioritized model is adopted in MARS [3] and TuCSoN [8]. LIME [9] supports both the reaction execution models. JavaSpaces [13], TSpaces [14] and WCL [11] adhere to the parallel model.

In order to compare these two approaches, we consider two Linda-like process calculi extended with reactive mechanisms that differ for the reaction execution model. As far as the parallel execution model is concerned, we consider the calculus and the results proved in a previous paper [1] where a calculus modeling the $notify$ primitive of JavaSpaces has been introduced. More precisely, the primitive $notify(a, R)$ is analysed that installs a listener responsible for activating the execution of the reaction R everytime a new instance of a datum a is introduced in the shared data space. As far as the prioritized execution model is concerned, we introduce a new calculus named `ReactLinda` which is essentially the adaptation of the calculus in [1] to the prioritized model. The new calculus has only two differences with respect to the previous one; the first difference is the name of the reactive coordination primitive ($reactsTo(a, R)$ instead of $notify(a, R)$); the second difference is in the operational semantics: in `ReactLinda` the presence of uncompleted reactions disables the other processes in the system from executing coordination primitives.

We use the two process calculi in order to investigate the possibility to encode one mechanism into the other one. Intuitively, we have that the parallel execution model can be encoded in the prioritized one. The idea is to delegate the execution of the reactions to parallel processes that are blocked willing to consume some activation data that are produced by the prioritized reactions. More formally, we can consider an encoding function $[\![\ ]\!]$ such that

$$[\![notify(a, R)]\!] \quad = \quad reactsTo(a, out(a_R)) \mid\, !in(a_R).R$$

where a_R is the name used for the activation datum, and $!in(a_R).R$ is the parallel composition of an unbonded amount of processes willing to consume the activation datum and subsequently execute the reaction R.

On the contrary, the modeling of $reactsTo$ in terms of $notify$ is clearly more complex because it is necessary to block the execution of the processes while reactions are in execution. This can be realized exploiting locks that forbid the processes from accessing the data space. Following this approach, the problem is how to detect the termination of the reactions in order to remove the lock; indeed, the number of listeners that observe a specific event is unpredictable (it depends on the run time behaviour of the system) thus also the number of reactions that are triggered by that event.

This informal discussion shows (some of) the difficulties that are encoutered while trying to encode the prioritized into the parallel reaction execution model. Using the process calculi we are able to formally prove an interesting impossibility result. This result is based on the decidability of properties such as process *termination* (i.e. existence of a finite completed computation) and process *divergence* (i.e. existence of an infinite computation). In [1] we proved that in the calculus with $notify$ termination is undecidable while divergence is decidable. In `ReactLinda`, on the contrary, both termination and divergence are undecidable.

As a consequence we have the following impossibility result: there exists no computable encoding of the prioritized reaction execution model into the parallel model that preserves divergence. From a theoretical point of view, this proves that the prioritized model is strictly more expressive than the parallel model. On the other hand, from a practical point of view, the undecidability of divergence (and termination) is a rather negative result. In fact, in the prioritized reaction execution model, reactions are expected to complete with neither deadlock nor divergence otherwise the rest of the system is blocked indefinitely.

In light of this observation about the practical usage of prioritized reactions, we continue our investigation trying to point out a significant decidable fragment for `ReactLinda`. In this perspective, we eliminate the unique explicit infinite operator, i.e. the replication operator, from `ReactLinda` and we investigate the decidability of termination and divergence in the obtained fragment, that we call `ReactLinda-!`.

The elimination of the replication operator does not necessarily brings to a finite calculus as reactive programming permits to simulate replication. For instance, the process $reactsTo(a, out(a).P).out(a)$ activates an unbounded amount of copies of process P, simply by permitting to a reaction to perform an *out* operation that triggers another instance of the reaction itself.

Even if the replication operator can be simulated following this approach, `ReactLinda-!` is more decidable than `ReactLinda`. Indeed, in `ReactLinda-!` both termination and divergence turn out to be decidable (while they are both undecidable in `ReactLinda`). This result is proved resorting to a Petri net semantics.

This theoretical result is of interest also from a practical viewpoint. Indeed, `ReactLinda-!` can be used as a reference model for prioritized reactions where both termination and divergence can be decided. In languages such as MARS and LIME restrictions are imposed to the reactive model in order to guarantee the completion of reactions. For instance, in MARS reactions cannot trigger other reactions, while in LIME blocking operations cannot be used inside reactions. Our decidability results open an alternative approach which can be based on the verification of the reactive programs before their actual execution. In the case the reactions can deadlock or diverge the run time system can decide to avoid their execution.

The paper is structured as follows. In Section 2 we present the syntax and the operational semantics of the considered calculi. In Section 3 we prove the undecidability results while in Section 4 the decidability results are discussed. Finally, Section 5 reports some conclusive remarks. Due to space limitation the formal proofs of theorems are not reported, but the proof techniques are carefully detailed.

2 The Process Calculi

In this Section, we introduce the process calculus `ReactLinda` based on the prioritized reaction execution model discussed in the Introduction. It is obtained as the adaptation to the prioritized model of the calculus introduced in [1] based on the parallel model. Syntactically, the main difference is that in the new calculus we separate processes, data, listeners and reactions in four different syntactic cathegories while in [1] they are all treated as parallel processes.

By borrowing typical techniques from the tradition of process calculi for concurrency (e.g., Milner's CCS [6]), a process (as well as a reaction) is described as a term of an algebra where the basic actions are the Linda coordination primitives in, rd and out, or the reactive operation $reactsTo$. The data space is represented by the multiset of the data actually available. The listeners are modeled with pairs (a, R) where a is the datum whose production is observed, and R is the corresponding reaction.

Formally, we consider a denumerable set of names for data, called $Data$, ranged over by a, b, ... A data space is a multiset of $Data$; formally the possible data spaces, ranged over by DS, DS', $\oplus$ are taken from $DataSpace = \mathcal{M}(Data)$. In the following we use $\oplus$ to denote multiset union, $\bigoplus_i M_i$ to denote the multiset union of the indexed multisets M_i, and we use a to denote also the singleton $\{a\}$.

Reactions are finite programs that can only execute input, read and output operations. We follow the LIME assumption according to which reactions cannot contain reactive statements. The set $Reac$ of reactions, ranged over by R, R', ..., is the set of terms generated by the following grammar:

$$R \ ::= \ \mathbf{0} \ \mid \ \mu.R \ \mid \ R|R \qquad\qquad \mu \ ::= \ out(a) \ \mid \ in(a) \ \mid \ rd(a)$$

where μ denotes an instance of one of the (non-reactive) coordination primitives. In the following we use $\prod_i R_i$ to denote the parallel composition of the indexed reactions R_i or $\mathbf{0}$ if the set of index is empty.

The reaction $\mathbf{0}$ represents the empty program; $\mu.R$ is a reaction that starts executing the primitive μ then becomes R; $R|R'$ is the parallel composition of the two reactions R and R'.

Listeners are formally modeled as follows. Let Lst, ranged over by L, L', ..., be the set of multisets on the cartesian product between $Data$ and $Reac$, namely $Lst = \mathcal{M}(Data \times Reac)$. The listener (a, R) is responsible for activating the reaction R on production of datum a.

The set $Proc$ of processes, ranged over by P, P', ... is defined by the following grammar:

$$P \ ::= \ \mathbf{0} \ | \ \mu.P \ | \ reactsTo(a, R).P \ | \ P|P \ | \ !P$$

where μ is defined as above for reactions. Processes extend reactions with the $reactsTo(a, R)$ prefix and the replication operator $!P$. In the following we use $\prod_i P_i$ to denote the parallel composition of the indexed processes P_i or the process $\mathbf{0}$ if the set of index is empty.

We will reason upto a structural congruence relation used for rearranging parallel composed processes (as well as reactions), for abstracting away from the empty program and for (un)folding the replicated processes. Formally, let $\equiv$ be the least congruence on processes and reactions such that

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

The set of configurations $Conf = Proc \times DataSpace \times Lst \times Reac$ is ranged over by $[P, DS, L, R]$ where P denotes the active processes, DS the current data space, L the set of actually installed listeners, and R represents the active reactions. We assume that initially the configurations start with an empty data space, without listeners installed, and without active reactions.

The semantics is defined in terms of a reduction relation $[P, DS, L, R] \rightarrow [P', DS', L', R']$ used to denote that the configuration on the left hand side can evolve, performing one step, to the configuration on the right hand side. The reduction semantics is defined as the least relation on configurations satisfying the axioms and rules in the Table 1.

The axioms `(inP)` and `(rdP)` model the execution of the in and rd operations performed by processes: the former removes a datum from the space, while the latter simply checks the presence of a datum leaving the data space unchanged. It is important to note that in both the axioms the reactions are assumed to be $\mathbf{0}$; in this way processes can perform in and rd operations only if there are no active reactions. The same assumption is made in the rules `rct` and `outP` modeling the other two operations that processes can perform. The axioms `(inR)` and `(rdR)` are the corresponding ones for reactions. The axiom `(rct)` models the execution of a $reactsTo(a, R)$ primitive; its effect is the addition of a listener.

The rule `(outP)` define the semantics of the $out(a)$ operation. The premise of the rule guarantees that all the interested listeners are taken into account. The rule `(outR)` is the corresponding one for output operations performed by reactions. Finally, rule `(cong)` is used to ensure that structurally congruent terms have the same operational semantics.

We denote with `ReactLinda-!` the fragment of `ReactLinda` obtained by removing the replication operator.

As described in the Introduction, we will focus on the decidability of two typical properties, *termination* (i.e. the existence of a finite completed computation) and *divergence* (i.e. the existence of an infinite computation). Formally, let $[P, DS, L, R] \not\rightarrow$ denote that there exists no $[P', DS', L', R']$ such

$$(\texttt{inP}) \qquad [in(a).P|Q, a \oplus DS, L, \mathbf{0}] \rightarrow [P|Q, DS, L, \mathbf{0}]$$

$$(\texttt{rdP}) \qquad [rd(a).P|Q, a \oplus DS, L, \mathbf{0}] \rightarrow [P|Q, a \oplus DS, L, \mathbf{0}]$$

$$(\texttt{inR}) \qquad [P, a \oplus DS, L, in(a).R|R'] \rightarrow [P, DS, L, R|R']$$

$$(\texttt{rdR}) \qquad [P, a \oplus DS, L, rd(a).R|R'] \rightarrow [P, a \oplus DS, L, R|R']$$

$$(\texttt{rct}) \qquad [reactsTo(a, R').P|Q, DS, L, \mathbf{0}] \rightarrow [P|Q, DS, L \oplus (a, R'), \mathbf{0}]$$

$$(\texttt{outP}) \qquad \frac{(b, R) \in L \text{ implies } b \neq a}{\begin{array}{c}[out(a).P|Q, DS, \bigoplus_i (a, R_i) \oplus L, \mathbf{0}] \rightarrow \\ [P|Q, a \oplus DS, \bigoplus_i (a, R_i) \oplus L, \prod_i R_i]\end{array}}$$

$$(\texttt{outR}) \qquad \frac{(b, R) \in L \text{ implies } b \neq a}{\begin{array}{c}[P, DS, \bigoplus_i (a, R_i) \oplus L, out(a).R|R'] \rightarrow \\ [P, a \oplus DS, \bigoplus_i (a, R_i) \oplus L, \prod_i R_i|R|R']\end{array}}$$

$$(\texttt{cong}) \ \frac{P \equiv P'' \quad P' \equiv P''' \quad R \equiv R'' \quad R' \equiv R''' \quad [P, DS, L, R] \rightarrow [P', DS', L', R']}{[P'', DS, L, R''] \rightarrow [P''', DS', L', R''']}$$

Table 1. The operational semantics of `ReactLinda`.

that $[P, DS, L, R] \rightarrow [P', DS', L', R']$. We say that P terminates, denoted with $P \downarrow$, if there exist $[P_1, DS_1, L_1, R_1], \ldots, [P_n, DS_n, L_n, R_n]$ such that $[P, \emptyset, \emptyset, \mathbf{0}] \rightarrow [P_1, DS_1, L_1, R_1] \rightarrow \ldots \rightarrow [P_n, DS_n, L_n, R_n] \not\rightarrow$. On the other hand, we say that P diverges, denoted with $P \uparrow$, if for any natural number i there exists $[P_i, DS_i, L_i, R_i]$ such that $[P, \emptyset, \emptyset, \mathbf{0}] = [P_0, DS_0, L_0, R_0]$ and $[P_i, DS_i, L_i, R_i] \rightarrow [P_{i+1}, DS_{i+1}, L_{i+1}, R_{i+1}]$.

3 The Undecidability Results

In this Section we discuss the undecidability of termination and divergence for `ReactLinda`.

We adopt the following technique: we reduce the decidability of termination (resp. divergence) to the decidability of termination (resp. divergence) on Random Access Machines, a well known Turing powerful formalism.

A Random Access Machines (RAM) [12] is composed of a finite set of registers, that can hold arbitrary large natural numbers, and by a program, that is a sequence of simple numbered instructions, like arithmetical operations (on the contents of registers) or conditional jumps.

It is not restrictive to assume that the registers $r_1, \ldots, r_n$ are initially empty. The execution of the program begins with the first instruction and 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. If the program terminates, the result of the computation is the contents of the registers.

In [7] it is shown that the following two instructions are sufficient to model every recursive function:

- $Succ(r_j)$: adds 1 to the content of register r_j;
- $DecJump(r_j, s)$: if the content of register r_j is not zero, then decreases it by 1 and go to the next instruction, otherwise jumps to instruction s.

We start presenting an encoding of RAMs in `ReactLinda`. This encoding is inspired by a previous encoding described in [1] based on the $notify(a, R)$ primitive; the main difference is that the reactions do not perform blocking operations. Indeed, under the prioritized model a blocked reaction blocks also the rest of the system.

The encoding is *nondeterministic* as it introduces some extra infinite computations; nevertheless, it is ensured that a RAM terminates if and only if the corresponding encoding has a terminating computation. As termination cannot be decided in Turing equivalent formalisms, the same holds also for `ReactLinda`.

The encoding implements nondeterministically $DecJump$ operations: two possible behaviours can be chosen, the first is valid if the tested register is not zero, the second otherwise. If the wrong choice is made, the computation is ensured to be infinite; in this case, we cannot say anything about the corresponding RAM. Nevertheless, if the computation terminates, it is ensured that it corresponds to the computation of the corresponding RAM. Conversely, any computation of the RAM is simulated by the computation of the corresponding encoding in which no wrong choice is performed.

Given the RAM with program PRG (composed by the instructions $I_1 \ldots I_k$) and registers $r_1 \ldots r_n$, the corresponding encoding $[\![PRG]\!]$ is defined in Table 2. The basic idea underlying this encoding is to represent instructions with replicated processes, the program counter with a datum that activate the corresponding instruction, and the content of each register r_j with a corresponding number of r_j data. Namely, the instruction I_i is modeled with the process $!in(p_i).[\![I_i]\!]$ where p_i is the program counter tuple and $[\![I_i]\!]$ is a process responsible for updating the register and the program counter. Before starting the computation, a listener is installed executing $reactsTo(div, out(div))$; this listener has the ability to activate an infinite computation. More precisely, the infinite computation is started in case a *loop* datum is consumed by the initially spawn process $in(loop).out(div)$. Finally, the actual computation is started emitting the program counter tuple p_1.

$$
\begin{array}{ll}
[\![PRG]\!] & = \prod_{i \in 1\ldots k} !in(p_i).[\![I_i]\!] \mid reactsTo(div, out(div)).out(p_1) \\
 & \mid\ !in(inc).INC \mid !in(dec).DEC \mid in(loop).out(div) \\
[\![i : Succ(r_j)]\!] & = out(r_j).reactsTo(zero_j, out(inc)).out(p_{i+1}) \\
[\![i : DecJump(r_j, s)]\!] & = out(c_i) \\
 & \mid\ in(c_i).out(loop).in(r_j).in(loop). \\
 & \quad reactsTo(zero_j, out(dec)).out(p_{i+1}) \\
 & \mid\ in(c_i).out(zero_j).in(zero_j).out(p_s) \\
\text{where:} & \\
INC & = out(loop).in(match).in(loop) \\
DEC & = out(match)
\end{array}
$$

Table 2. Termination preserving encoding of RAMs in `ReactLinda`.

Every time an increment (resp. a decrement) on the register r_j is performed, a new listener $(zero_j, out(inc))$ (resp. $(zero_j, out(dec))$) is spawn. The presence of these listeners permits to check if the actual content of a register r_j is zero by verifying if the occurrences of $(zero_j, out(inc))$ corresponds to those of $(zero_j, out(dec))$.

An increment instruction on r_j simply increments the number of the occurrences of the datum r_j and install a new listener $(zero_j, out(inc))$. When a $DecJump$ instruction is executed, a nondeterministic choice between the two possible branches of the instruction occurs. The choice is modeled putting in parallel the two branches that compete for consuming the datum c_i. We now analyse the computation in case the wrong choice is done. There are two cases to analyse: (i) a decrement on a register containing zero, (ii) a jump for zero on a non-empty register.

In the case (i), $out(loop).in(r_j).in(loop).reactsTo(zero_j, out(dec)).out(p_{i+1})$ is activated with no r_j data available. Thus, the program produces the datum *loop* and blocks trying to execute $in(r_j)$. The produced datum *loop* will be consumed by the process $in(loop).out(div)$ thus an infinite computation is started.

In the case (ii), the process $out(zero_j).in(zero_j).out(p_s)$ is activated when there are more occurrences of the listener $(zero_j, out(inc))$ than those of the listener $(zero_j, out(dec))$. When the datum $zero_j$ is emitted, its production is notified to the listeners; then the corresponding data *inc* and *dec* are produced. These data activates corresponding processes INC and DEC. Each DEC emits a datum *match* while each INC produces a datum *loop*, and requires a *match* datum to be consumed before removing the emitted *loop*. As there are more INC processes than DEC, one of the processes INC will block waiting for an unavailable *match* datum; thus it will not consume its corresponding *loop*. As before, an infinite computation will be activated.

The correspondence between the RAMs and their modeling in `ReactLinda` is stated by the following Theorem. The undecidability of termination in the calculus `ReactLinda` is a trivial corollary of this Theorem.

Theorem 1. *Let us consider a RAM with program PRG (composed by the instructions $I_1 \dots I_k$) and registers $r_1 \dots r_n$, and the corresponding encoding $[\![PRG]\!]$ in* `ReactLinda` *defined in Table 2. We have that the RAM terminates its computation if and only if $[\![PRG]\!] \downarrow$.*

Now, we prove that also divergence is undecidable in `ReactLinda`. In order to prove this result we present how to encode RAMs preserving divergence, i.e. a RAM does not terminate if and only if the corresponding encoding has an infinite computation.

The new encoding is similar to the encoding above because it is nondeterministic: the branch of $DecJump$ is selected nondeterministically independently of the actual contents of the registers. The difference is that in case the wrong branch is selected the computation is guaranteed to block. Thus, the encoding has an infinite computation if and only if the RAM does not terminate.

$[\![PRG]\!]$	$= \prod_{i \in 1 \dots k} !in(p_i).[\![I_i]\!] \mid out(p_1)$
$[\![i : Succ(r_j)]\!]$	$= out(r_j).reactsTo(zero_j, INC).out(p_{i+1})$
$[\![i : DecJump(r_j, s)]\!]$	$= out(c_i)$
	$\mid \; in(c_i).in(r_j).reactsTo(zero_j, DEC).out(p_{i+1})$
	$\mid \; in(c_i).out(zero_j).in(zero_j).out(p_s)$
where:	
INC	$= in(match)$
DEC	$= out(match)$

Table 3. Divergence preserving encoding of RAMs in `ReactLinda`.

We overload the notation $[\![PRG]\!]$ used also for the new encoding reported in Table 3. If the decrement branch is wrongly selected all the computation blocks because the input operation $in(r_j)$ cannot be executed; in case a wrong jump is selected, the computation blocks because one of the INC reactions will not terminate due to the impossibility to execute the $in(match)$ operation.

The following theorem states that the new RAM encoding preserves divergence, thus divergence is undecidable in the calculus `ReactLinda`.

Theorem 2. *Let us consider a RAM with program PRG (composed by the instructions $I_1 \dots I_k$) and registers $r_1 \dots r_n$, and the corresponding encoding $[\![PRG]\!]$ in* `ReactLinda` *defined in Table 3. We have that the RAM does not terminate its computation if and only if $[\![PRG]\!] \uparrow$.*

4 The decidability results

In this Section we show that both termination and divergence are decidable in `ReactLinda-!`. We reduce termination on `ReactLinda-!` to termination on Place/Transition Petri nets. As termination is decidable on such class of Petri nets [2], we get the decidability result for termination on `ReactLinda-!`. Divergence on `ReactLinda-!` is reduced to a property similar to divergence on the class of nets corresponding to `ReactLinda-!` processes.

4.1 P/T Nets

We recall Place/Transition nets with unweigthed flow arcs (see, e.g., [10]). Here we provide a characterization of this model which is convenient for our aims.

Definition 1. *Given a set S, a* finite multiset *over S is a function $m : S \to I\!N$ such that the set $dom(m) = \{s \in S \mid m(s) \neq 0\}$ is finite. The* multiplicity *of an element s in m is given by the natural number $m(s)$. The set of all finite multisets over S, denoted by $\mathcal{M}_{fin}(S)$, is ranged over by m. A multiset m such that $dom(m) = \emptyset$ is called* empty*. The set of all finite sets over S is denoted by $\wp_{fin}(S)$.*

Given the multiset m and m', we write $m \subseteq m'$ if $m(s) \leq m'(s)$ for all $s \in S$ while $\oplus$ denotes their multiset union*: $m \oplus m'(s) = m(s) + m'(s)$. The operator $\setminus$ denotes* multiset difference*: $(m \setminus m')(s) =$ if $m(s) \geq m'(s)$ then $m(s) - m'(s)$ else 0. The* scalar product*, $j \cdot m$, of a number j with m is $(j \cdot m)(s) = j \cdot (m(s))$.*

To lighten the notation, we sometimes use the following abbreviation. If m is a multiset containing only one occurrence of an element s (i.e., $dom(m) = \{s\}$ and $m(s) = 1$) we denote m by only s. Multiset union is represented also by comma, i.e., $m, m' = m \oplus m'$. Let m be a multiset over S and m' a multiset over $S' \supseteq S$, such that $(m'(s') = 0)$ for each $s' \in S' \setminus S$; with abuse of notation, we sometimes use m in place of m', and vice versa.

Definition 2. *A P/T net is a pair (S, T) where S is the set of* places *and $T \subseteq \mathcal{M}_{fin}(S) \times \mathcal{M}_{fin}(S)$ is the set of* transitions*.*

Finite multisets over the set S of places are called markings*. Given a marking m and a place s, we say that the place s contains $m(s)$* tokens*.*

A P/T net is finite if both S and T are finite.

A P/T system is a triple $N = (S, T, m_0)$ where (S, T) is a P/T net and m_0 is the initial marking*.*

A transition $t = (c, p)$ is usually written in the form $c \longrightarrow p$. The marking c, usually denoted by ${}^\bullet t$, is called the preset *of t and represents the tokens to be* consumed*; the marking p, usually denoted by $t^\bullet$, is called the* postset *of t and represents the tokens to be* produced*.*

A transition t is enabled *at m if ${}^\bullet t \subseteq m$. The execution of a transition t enabled at m produces the marking $m' = (m \setminus {}^\bullet t) \oplus t^\bullet$. This is written as $m \stackrel{t}{\longrightarrow} m'$ or simply $m \longrightarrow m'$ when the transition t is not relevant. We use*

σ, τ to range over sequences of transitions; the empty sequence is denoted by ε; let $\sigma = t_1, \ldots, t_n$, we write $m \xrightarrow{\sigma} m'$ to mean the firing sequence $m \xrightarrow{t_1} \cdots \xrightarrow{t_n} m'$.

A marking m is dead *if $m \not\longrightarrow$. We say that the P/T system N* terminates *if there exists a dead marking reachable from the intial marking m_0, i.e., $m_0 \longrightarrow^* m \not\longrightarrow$. We say that the P/T system N is* divergent *if there exists an infinite sequence of markings $m_1, \ldots, m_i, \ldots$ such that $m_i \longrightarrow m_{i+1}$ for $i \leq 0$.*

4.2 Reducing termination on `ReactLinda-!` to termination on nets

The basic idea underlying the definition of a net semantics is to decompose a process P in the (finite) multiset of its sequential subprocesses that appear at top-level (i.e., occur unguarded in P); this multiset is then considered as the marking of a P/T Petri net.

Besides processes, a `ReactLinda-!` configuration also contains data and listeners, that will be represented as places in the net as well. The execution of a computational step of the process will correspond to the firing of a transition in the corresponding net.

The main problems to face with in the costruction of a P/T net with the same behaviour of a process w.r.t. termination are the following:

- when a new datum is produced, *all* the reactions of the currently installed listeners are spawn,
- the execution of the process is blocked until the execution of all the reactions is terminated.

A faithful representation of the above described features in a Petri net requires to consider nets extended with transfer and inhibitor arcs; unfortunately, termination becomes undecidable on such an extended class of nets.

However, a closer look to the `ReactLinda-!` calculus permits to note that the sets of *reactsTo* and *out* operations that an initial process can perform are finite.[1] Hence, for a given process P,

- the number of occurrences of the *reactsTo* primitive is an upper bound to the number of occurrences of a given listener,
- the number of occurrences of the *out* primitive occurring in the initial process is an upper bound to the number of times that the reactions start executing.

The upper limit to the number of occurrences of a given listener permits to faithfully model the activation of a reaction for each active listener when an output of a datum is performed. Instead of representing n active occurrences of listener (a, R) as n tokens in place (a, R), we adopt the following representation: each pair (a, R) of potential listeners is represented by the following set of places

[1] Note that this finiteness property holds because the calculus does contain neither replication nor the possibility to install new listeners during the execution of reactions.

in the net: $\{0\star(a,R), 1\star(a,R), \ldots, k\star(a,R)\}$, where k is the number of *reactsTo* primitives occurring in the initial process; a token in place $h\star(a,R)$ models the situation where exactly h active occurrences of listener (a,R) appear in the net.

The prioritized execution of reactions cannot be faithfully modeled by a Petri net; however, the upper limit to the number of times that the reactions set can be activated permits to construct a net with the same behaviour of a `ReactLinda-!` process w.r.t. termination. The basic idea is to consider $k+1$ copies of the subnets that model the behaviour of processes, where k is the number of *out* primitives occurring in the initial process. One of the subnets represents the "main" process, whereas the remaining subnets are used to model the execution of reactions: each time an output of datum a is performed by the main process, a yet unused subnet is filled with the marking corresponding to the reactions of the listeners waiting for the emission of a.

After activation, the computation is carried out by the subnet. According to the operational semantics of Section 2, if an $out(b)$ operation is performed the reactions of the installed listeners (b, R_i) are activated by producing tokens in those places of the subnet that correspond to the R_i.

In any moment, exactly one subnet is active. A distinct label is associated to the places of each subnet: label m is associated to the places of the subnet representing the main process, while labels $r_1, \ldots, r_k$ decorate places of the subnets used for reactions. Each subnet is equipped with three control places, decorated with the same label of the subnet. A token in place $l : running$ represents the fact that the subnet corresponding to label l is active. Places $l : unused$ and $l : finished$ are used only for subnets that represent reactions. A token in place $l : unused$ represents the fact that the subnet labelled by l has not been used yet, and can be used to execute a reaction; a token in place $l : finished$ denotes the fact that the subnet labelled by l is supposed to have successfully terminated its execution.

The behaviour of reactions is modeled in the following way: a subnet representing a reaction, and labelled with r, may choose to terminate its execution and to pass the control to the main process. If the execution was not terminated, the behaviour of the net does not correspond to the behaviour of a configuration. However, some tokens remain in some place of the subnet, corresponding to a sequential subprocess; the presence of a token in such places, together with the presence of a token in the $finished$ place, enables a transition that produces a token in a place *loop*; place *loop* has a self-loop on itself, hence a token in *loop* forbids termination of the net.

To define the net corresponding to a process, we introduce some auxiliary definitions:

Definition 3. *The set of sequential subprocesses of a process is defined inductively as follows:*

$$\begin{array}{ll} sub(\mathbf{0}) = \emptyset & sub(\mu.P) = \{\mu.P\} \cup sub(P) \\ sub(reactsTo(a,R).P) = \{sub(R)\} \cup sub(P) & sub(P|Q) = sub(P) \cup sub(Q) \end{array}$$

(in)	$l : running, l : in(a).P, a \;\longrightarrow\; l : running, l : dec(P)$
(rd)	$l : running, l : rd(a).P, a \;\longrightarrow\; l : running, l : dec(P), a$
(react)	$m : running, m : reactsTo(a, R).P, k \star (a, R) \;\longrightarrow$ $m : running, m : dec(P), (k+1) \star (a, R)$
(out − M)	$m : running, m : out(a), P, r : unused, \bigoplus_{i:a_i=a} k_i \star (a_i, R_i) \;\longrightarrow$ $m : dec(P), r : running, r : \bigoplus_{i:a_i=a} k_i \cdot R_i, \bigoplus_{i:a_i=a} k_i \star (a_i, R_i)$
(out − R)	$r : running, r : out(a).P, \bigoplus_{i:a_i=a} k_i \star (a_i, R_i) \;\longrightarrow$ $m : dec(P), r : running, r : \bigoplus_{i:a_i=a} k_i \cdot R_i, \bigoplus_{i:a_i=a} k_i \star (a_i, R_i)$
(end − R)	$r : running \;\longrightarrow\; r : finished, m : running$
(startloop)	$r : finished, r : P \;\longrightarrow\; loop$
(loop)	$loop \;\longrightarrow\; loop$

Table 4. The transitions schemata.

The decomposition *of a process in the set of top-level subprocesses is defined as follows:*

$$dec(\mathbf{0}) = \emptyset \qquad dec(\mu.P) = \{\mu.P\}$$
$$dec(reactsTo(a, R).P) = \{reactsTo(a, R).P\} \qquad dec(P|Q) = dec(P) \cup dec(Q)$$

The set of names occurring in a process is defined as follows:

$$names(\mathbf{0}) = \emptyset \qquad names(P|Q) = names(P) \cup names(Q)$$
$$names(in(a).P) = names(rd(a).P) = names(out(a).P) = \{a\} \cup names(P)$$
$$names(reactsTo(a, R).P) = \{a\} \cup names(R) \cup names(P)$$

The set of potential listeners of a process is defined as follows:

$$listeners(\mathbf{0}) = \emptyset$$
$$listeners(\mu.P) = listeners(P)$$
$$listeners(reactsTo(a, R).P) = \{(a, R)\} \cup listeners(P)$$
$$listeners(P|Q) = listeners(P) \cup listeners(Q)$$

With $\#rct(P)$ we denote the number of occurrences of $reactsTo$ primitives in P, with $\#out(P)$ the number of occurrences of out operations in P.

Given a process P, the set of labels for subnets is $L = \{m, r_1, \ldots, r_{\#out(P)}\}$, ranged over by l, l'. The set $Trans$ contains all the instances of the transitions schemata reported in Table 4. With $l : m$ we denote the multiset obtained by decorating each element in m with label l, i.e., for all $s \in S$, $l : m(s) = m(s)$. Axioms (in) and (rd) deal with the execution of an *in* and a *read* operation, respectively, in one of the subnets. Axiom (react), corresponds to the production of a new listener of kind (a, R): if there are k active occurrences of listener (a, R), after the execution of the notify operation there are $k + 1$ active occurrences of listener (a, R); hence, a token is moved from place $k \star (a, R)$ to place $(k + 1) \star (a, R)$. Axiom (out-M) models the effect of an output operation of datum a in the main process: a not yet used subnet is chosen and marked with the tokens corresponding to the decomposition of all the reactions corresponding to listeners on datum a, and the control is passed to the chosen subnet. On the other hand, if the output operation is performed in a reaction, according to axiom (out-R) the tokens corresponding to the decomposition of reactions of active listeners on the datum are produced in the active subnet. Axiom (end-R) permits to end the execution of a (possibly not terminated) subnet corresponding to a reaction and to return the control to the main program. If the execution of a subnet corresponding to a reaction has been stopped before all reaction processes are terminated, by axiom (startloop) a token in place *loop* is produced; by axiom (loop), this token prevents the whole net to terminate.

Definition 4. *Let P be a* ReactLinda-! *process. We define the system $Net(P) = (S, T, m_0)$, where*

$$
\begin{aligned}
S &= L \times sub(P) \cup \; names(P) \; \cup \\
&\quad \{k \star (a, R) \mid (a, R) \in listeners(P) \wedge 0 \leq k \leq \#rct(P)\} \cup \\
&\quad L \times \{running, unused, finished\} \cup \{loop\} \\
T &= \{(c, p) \in Trans \mid c, p \subseteq S\} \\
m_0 &= dec(P)
\end{aligned}
$$

Note that $Net(P)$ is a finite P/T net.

The following lemma permits to reduce the decidabiliy of termination for process P to the decidabiliy of termination for the P/T system $Net(P)$.

Lemma 1. *Let P be a* ReactLinda-! *process. P terminates if and only if $Net(P)$ terminates.*

Theorem 3. *Let P be a* ReactLinda-! *process. The termination problem $P \downarrow$ is decidable.*

4.3 Deciding divergence on ReactLinda-!

The decidability of divergence is obtained by removing from the net constructed in the previous subsection the (startloop) and (loop) transition, and by using an adaptation of the procedure to check divergence of a P/T system. The basic technique consists in the construction of the so-called coverability tree [5]; the system diverges if a marking m is found that covers (i.e., is bigger than) another marking m' appearing in the path from the root of the tree (i.e., the initial marking m_0) to m. The net system used to check divergence may exhibit some wrong behaviour in the case the control is returned to the main program when some process in the reactions set is not terminated. For example, consider $reactsTo(a, in(b)).out(a).reactsTo(loop, out(loop)).out(loop)$. This process reaches a deadlock during the execution of the reaction $in(b)$, as no datum b will be produced. However, in the net system corresponding to the process it may happen that the control is returned to the continuation of the main program, i.e., $reactsTo(loop, out(loop)).out(loop)$ even if the reaction $in(b)$ has not been executed. Hence, the net may exhibit a divergent computation. However, all the markings in such a divergent computation are "erroneous", and can be recognized by the following feature: there exist two places, $r : finished$ and $r : P'$ (for some process P'), such that both places contain at least one token. This means that the subnet r has stopped its execution when there is at least one reaction that has not finished its execution. Hence, the existence of a divergent computation of such a net system can be decided by a slight variation of the basic procedure: the system diverges if a marking m is found that covers another marking m' appearing in the path from the root of the tree to m, and moreover m is not an erroneous marking.

The set $Trans'$ contains all the instances of the transitions schemata reported in Table 4, but axioms `(startloop)` and `(loop)`. The P/T net used to decide divergence of P is constructed as follows:

Definition 5. *Let P be a* `ReactLinda-!` *process. We define the system $Net'(P) = (S', T', m'_0)$, where*

$$\begin{aligned} S' &= L \times sub(P) \cup \ names(P) \cup \\ &\quad \{k \star (a, R) \mid (a, R) \in listeners(P) \wedge 0 \leq k \leq \#rct(P)\} \cup \\ &\quad L \times \{running, unused, finished\} \\ T' &= \{(c, p) \in Trans' \mid c, p \subseteq S\} \\ m'_0 &= dec(P) \end{aligned}$$

Note that $Net'(P)$ is a finite P/T net.

Definition 6. *A marking m of $Net'(P)$ is* erroneous *is there exist i and P' s.t. the following holds: $m(r_i : finished) = 1$ and $m(r_i : P') > 0$.*

Note that if a marking m is erroneous then any marking reachable from m is erroneous. The following lemmata permit to reduce decidability of divergence for the calculus `ReactLinda-!` processes to decidability of a similar property on the class of net systems constructed in Definition 4.3.

Lemma 2. *Let P be a* `ReactLinda-!` *process. The existence of a divergent computation not containing erroneous markings is a decidable property of the net system $Net'(P)$.*

Lemma 3. *A* `ReactLinda-!` *process P diverges if and only if there exists a divergent computation in $Net'(P)$ that does not contain erroneous markings.*

Theorem 4. *Let P be a* `ReactLinda-!` *process. The divergence problem $P \uparrow$ is decidable.*

5 Conclusion

The contribution and the theoretical results proved in this paper have been already discussed in the Introduction. A relevant result, that has some relevance also from a practival viewpoint, is the decidability of both termination and divergence in `ReactLinda-!`, the replication free calculus based on the prioritized reaction execution model. It is worth to notice that a similar result does not hold for the calculus in [1] based on the parallel execution model. Indeed, if we remove replication from that calculus termination is still undecidable. This result follows from the possibility to install new listeners during the execution of reactions. This is natural under the parallel model due to the interleaving between processes and reactions. For instance, the $notify$ operation of JavaSpaces [13] can be freely executed within reactions. On the contrary, under the prioritized model, the set of the installed listeners is usually leaved unchanged during the reactions execution. This holds, e.g., in LIME [9] where the $reactsTo$ operation cannot be performed within reactions.

References

1. N. Busi and G. Zavattaro. On the Expressivenes of Event Notification in Data-Driven Coordination Languages. In *Proc. of ESOP 2000*, volume 1782 of *Lecture Notes in Computer Science*, pages 41–55. Springer-Verlag, Berlin, 2000.
2. A. Cheng, J. Esparza, and J. Palsberg, Complexity results for 1-safe nets. *Theoretical Computer Science*, 147:117–136, 1995.
3. G. Cabri, L. Leonardi, and F. Zambonelli. Reactive Tuple Spaces for Mobile Agent Coordination. In *Proc. 2nd Int. Workshop on Mobile Agents*, volume 1477 of *Lecture Notes in Computer Science*, pages 237–248. Springer-Verlag, Berlin, 1998.
4. D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.
5. R. M. Karp and R. E. Miller, Parallel Program Schemata. *Journal of Computer and System Sciences* 3: 147-195, 1969.
6. R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.
7. M.L. Minsky. *Computation: finite and infinite machines.* Prentice-Hall, 1967.
8. A. Omicini and F. Zambonelli. Coordination of mobile information agents in TuCSoN. *Journal of Internet Research*, 8(5), 1998.
9. G.P. Picco, A.L. Murphy, and G.-C. Roman. Lime: Linda meets mobility. In *Proc. of the 21st ICSE*, 1999.
10. W. Reisig. *Petri nets: An Introduction.* EATCS Monographs in Computer Science, Springer, 1985.
11. A. Rowstron. WCL: A web co-ordination language. *World Wide Web Journal*, 1(3):167–179, 1998.
12. J.C. Shepherdson and J.E. Sturgis. Computability of recursive functions. *Journal of the ACM*, 10:217–255, 1963.
13. J. Waldo et al. Javaspace specification. Technical report, Sun Microsystems, 1998.
14. P. Wyckoff, S. McLaughry, T. Lehman, and D. Ford. T spaces. *IBM Systems Journal*, 37(3):454–474, 1998.