Termination and Divergence Are Undecidable Under a Maximum Progress Multi-step Semantics for LinCa

Mila Majster-Cederbaum and Christoph Minnameier*

Institut für Informatik
Universität Mannheim, Germany
cmm@informatik.uni-mannheim.de

Abstract. We introduce a multi-step semantics *MTS-mp* for *LinCa* which demands *maximum progress* in each step, i.e. which will only allow transitions that are labeled with maximal (in terms of set inclusion) subsets of the set of enabled actions. We compare *MTS-mp* with the original *ITS*-semantics for LinCa specified in [CJY94] and with a slight modification of the original *MTS*-semantics specified in [CJY94]. Given a *LinCa*-process and a *Tuple Space* configuration, the possible transitions under our *MTS-mp*-semantics are always a subset of the possible transitions under the presented *MTS*-semantics for *LinCa* .

We compare the original *ITS*-semantics and the presented *MTS*-semantics with our *MTS-mp*-semantics, and as a major result, we will show that under *MTS-mp* neither termination nor divergence of *LinCa* processes is decidable. In contrast to this [BGLZ04], in the original semantics for *LinCa* [CJY94] termination is decidable.

1 Introduction

A Coordination Language is a language defined specifically to allow two or more parties (components) to communicate for the purpose of coordinating operations to accomplish some shared (computation) goal. *Linda* seems to be the mostly known Coordination Language. Ciancarini, Jensen and Yankelevich [CJY94] defined *LinCa*, the *Linda Calculus* and gave a single-step, as well as a multi-step semantics for *LinCa*.

A *Linda* process may contain several parallel subprocesses that communicate via a so called *Tuple Space*. The *Tuple Space* is some kind of global store, where tuples are stored. In *Linda*, a tuple is a vector consisting of variables and/or constants, and there is a matching relation that is similar to data type matching in common programming languages. For the purpose of investigating the properties of the coordination through the *Tuple Space* it is common practice to ignore the matching relation and internal propagation of tuples. Tuples are distinguished from each other by giving them unique names $(t_1, t_2, t_3, ...)$ and *LinCa* is based on a *Tuple Space* that is countably infinite.

* Corresponding author.

K. Barkaoui, A. Cavalcanti, and A. Cerone (Eds.): ICTAC 2006, LNCS 4281, pp. 65–79, 2006.

As far as the semantics for *LinCa* is concerned, the traditional interleaving point of view does not make any assumptions about the way concurrent actions are performed, i.e. for any number of processing units and independently of their speed all possible interleavings of actions are admitted. On the other hand, the traditional multi-step point of view allows actions to be carried out concurrently or interleaved.

Let us assume a system, where all processing units work at the same speed and where all of them are globally clocked. For such a system, we might demand *maximum progress*, i.e. as long as additional actions can be performed in the present step they must be. More formally, we consider only (set inclusion) maximal sets of actions for each step.

Consider, for example, a system where a number of workers (processes) have to perform different jobs (calculations) on some object (tuple). The objects are supplied sequentially by some environment, which is represented by the process foreman. (Readers that are familiar with *LinCa* might want to have a look at the end of Section 3, where we model the example in *LinCa*.)

In a setting with a common clock for all processes where the workers' calculations (plus taking up the object) can always be finished within one clock cycle we would (for maximum efficiency) want the systems semantics to represent the actual proceeding as follows: All workers are idle while the foreman supplies an object. The foreman waits while all the workers read the object and perform their jobs simultaneously. All workers put their results into the tuple space simultaneously while the foreman deletes the object, and so on.

In this paper we study a *MTS-mp* (*Multi-Step Transition System* with *maximum progress*) semantics that models the specified behavior. As already implicitly stated in this example, we assume a data-base-like setting, where multiple read-operations may be performed on a single instance of a tuple (whereas this is not the case for *in*-operations). As a remark, we want to add, that this detail in design does however not affect the decidability results presented in Section 5 (this is obvious due to the fact that the given encoding of a *RAM* in LinCa doesn't include any *rd*-operation). The paper is organized as follows: In Section 2, we set up notation and terminology. In Section 3, we present the original interleaving semantics for *LinCa* as well as a multi-step semantics and the *MTS-mp* semantics. In Section 4, we establish a relation between the non-maximum-progress semantics and *MTS-mp*. Finally, Section 5 includes the main purpose of this paper: i.e. termination and divergence are undecidable for *LinCa* under *MTS-mp*. This is an interesting result as we do adopt the basic version of the *LinCa* language used in [BGLZ04], where it is shown that termination is decidable for the traditional interleaving semantics. In particular, we do not apply the predicative operator $inp(t)?P\_Q$ (see, e.g. [BGM00]) that represents an "if-then-else-construct" and thereby makes it easy to give a deterministic simulation of a *RAM*.

2 Definitions

- Most sets in this paper represent multisets. Given a multiset M, we write $(a, k) \in M$ $(k \geq 0)$ iff M includes exactly k instances of the element a. We will write $a \in M$ instead of $(a, 1) \in M$ and $a \notin M$, instead of $(a, 0) \in M$. We will use the operators $\uplus$, $\setminus$ and $\subseteq$ on multisets in their intuitive meaning.

- Given a multiset M we denote by $set(M)$ the set derived from M by deleting every instance of each element except for one, i.e. $set(M) = \{a \mid \exists i > 0 \in \mathbb{N} : (a, i) \in M\}$
- Given a set S we denote the *power-multiset* (that is the set of subsets that may include multiple instances of the same element of S) of S by $\wp(S)$.
- LinCa processes:
  Note, that by *Tuple Space*, we denote the basic set from which *tuples* are chosen and by *Tuple Space* configuration we refer to the state of our store in the present computation, i.e. a *Tuple Space* configuration is a multiset over the *Tuple Space*, i.e. for each *Tuple Space* configuration M and the underlying *Tuple Space* TS, we have $M \in \wp(TS)$.

  In order to show some properties of the introduced semantics, we will sometimes modify it slightly, by adding some extra tuples to TS. We will denote these extra tuples by c, d, e and we will write TS_{cde} for $TS \cup \{c, d, e\}$, where $TS \cap \{c, d, e\} = \emptyset$.

  Given a fixed *Tuple Space* TS, we can define the set of processes $LinCa_{TS}$ as the set of processes derived from the grammar in Figure 1, where every time we apply one of the rules $\{P := in(t).P, P := out(t).P, P := rd(t).P, P :=! \; in(t).P\}$, t is substituted by an element of the *Tuple Space*. $in(t), out(t)$ and $rd(t)$ are called actions. If $t \in \{c, d, e\}$ then they are called *internal* actions, else *observable* actions. Trailing zeros (.0) will be dropped in examples.

$$\boxed{\text{P} := 0 \;\Big|\; \text{in(t).P} \;\Big|\; \text{out(t).P} \;\Big|\; \text{rd(t).P} \;\Big|\; \text{P} \mid \text{P} \;\Big|\; !\ \text{in(t).P}}$$

Fig. 1. LinCa

- $ea(P)$ with P a *LinCa*-process denotes the multiset of enabled actions of P, defined in Figure 2. We define a decomposition of (the tuples used in) $ea(P)$ into three subsets $ea_{IN}(P), ea_{OUT}(P), ea_{RD}(P)$ as given in Figure 3:

$$\boxed{\begin{array}{l}
1)\ ea(0) = \emptyset \\
2)\ ea(in(t).P) = \{in(t)\} \\
3)\ ea(out(t).P) = \{out(t)\} \\
4)\ ea(rd(t).P) = \{rd(t)\} \\
5)\ ea(!\, in(t).P) = \{(in(t), \infty)\} \\
6)\ ea(P \mid Q) = ea(P) \uplus ea(Q)
\end{array}}$$

Fig. 2. The set of enabled actions $ea(P)$ of a process $P \in LinCa$

$ea_{IN}(P) = \{(t,i) \mid (in(t),i) \in ea(P)\}$
$ea_{OUT}(P)$ analogously
$ea_{RD}(P)$ analogously

Fig. 3. The sets $ea_{IN}(P), ea_{OUT}(P), ea_{RD}(P)$ of a process $P \in LinCa$

The notions $(in(t), \infty) \in ea(P)$ and $(t, \infty) \in ea_{IN}(P)$ describe the fact, that infinitely many actions $in(t)$ are *enabled* in P. These notions will only be used for *enabled actions*, never for *Tuple Space* configurations, because (due to the *in-guardedness* of *replication*) all computed *Tuple Space* configurations remain finite.

- A Labeled Transition System is a triple $(S, Lab, \rightarrow)$, where S is the set of states, Lab is the set of labels and $\rightarrow \subseteq S \times Lab \times S$ is a ternary relation (of labeled transitions). If $p, q \in S$ and $a \in Lab$, $(p, a, q) \in \rightarrow$ is also denoted by: $p \xrightarrow{a} q$. This represents the fact that there is a transition from state p to state q with label a. We write $p \not\rightarrow$ iff $\not\exists a \in Lab, q \in S : p \xrightarrow{a} q$. In addition we often want to designate a starting state s_0, in this case we use the quadruple $(S, Lab, \rightarrow, s_0)$.

  In the Transition Systems describing the various semantics, states are pairs $< P, M >$ of *LinCa*-processes and *Tuple Space* configurations and labels are triples (I, O, R) of (possibly empty) multisets of tuples, where I represents the performed *in*-actions, O the performed *out*-actions and R the performed *rd*-actions. We write τ instead of (I, O, R) iff $I, O, R \in \wp(\{c, d, e\})$ and call τ *internal* label and a transition $s \xrightarrow{\tau} s'$ an *internal* transition. A label $a = (I, O, R) \neq \tau$ is called *observable* label and a transition $s \xrightarrow{a} s'$ is called *observable* transition.
- Let *SEM* $\in$ \{*ITS, MTS, MTS-mp*\} (see Section 3 for details). The *SEM*-semantics of $LinCa_{TS}$ is given by the Transition System $(S, Lab, \rightarrow)$, where:
  1. $S = LinCa_{TS} \times \wp(TS)$
  2. $Lab = \wp(TS) \times \wp(TS) \times \wp(TS)$
  3. $\rightarrow = \rightarrow_{SEM}$ (see Section 3)

  For a process $P \in LinCa_{TS}$ the *SEM*-semantics is considered as $(S, Lab, \rightarrow_{SEM}, < P, \emptyset >)$ and we denote it by $SEM[P]$.
- Given a LTS LTS_1 and nodes $s_1, s_1' \in S$ we define: $s_1 \overset{(I,O,R)}{\rightarrow^+} s_1'$ iff $\exists s_2, ..., s_n \in S$, such that: $s_1 \xrightarrow{\tau} s_2 \xrightarrow{\tau} ... \xrightarrow{\tau} s_n \xrightarrow{(I,O,R)} s_1'$
- Given a LTS LTS_1 with starting state s_0 we define its set of *traces* as follows: $traces(LTS_1) := \{(a_1, a_2, ...) \in Tr_{Lab} \mid \exists s_1, s_2, ... \in S : s_0 \overset{a_1}{\rightarrow^+} s_1 \overset{a_2}{\rightarrow^+} s_2 ...\}$ where $Tr_{Lab} = (Lab \setminus \{\tau\})^* \cup (Lab \setminus \{\tau\})^\infty$ and S^* (S^∞) denotes the set of finite (infinite) sequences over a set S.
- a LTS LTS_1 with starting state s_0 terminates iff:
  $\exists s_1, ..., s_n \in S, a_1, ..., a_n \in Lab : s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} ... \xrightarrow{a_n} s_n \not\rightarrow$
- a LTS LTS_1 with starting state s_0 diverges iff it has at least one infinite transition sequence, i.e: $\exists s_i \in S, a_i \in Lab : s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} ...$

– Let $LTS_1 = (S_1, Lab_1, \rightarrow_1, s_{01})$ and $LTS_2 = (S_2, Lab_2, \rightarrow_2, s_{02})$ be two Labeled Transition Systems. We write $LTS_1 \preceq LTS_2$ iff the following properties hold:
  1) $traces(LTS_1) = traces(LTS_2)$
  2) LTS_2 is able to weakly step simulate LTS_1, i.e. $\exists R \subseteq S_1 \times S_2$ such that:
    2.1) $(s_{01}, s_{02}) \in R$ and
    2.2) $(p, q) \in R \wedge p \stackrel{(I,O,R)}{\rightarrow} p' \Rightarrow \exists q' \in S_2 : q \stackrel{(I,O,R)}{\rightarrow^+} q' \wedge (p', q') \in R$.

3 Semantics

In this section, we introduce the ITS-semantics for *LinCa* based on the semantics given in [BGLZ04] and a MTS-semantics that we consider the natural extension of ITS. In the given MTS-semantics, we allow (in contrast to [CJY94]) an arbitrarily large number of *rd*-actions to be performed simultaneously on a single instance of a tuple.

To describe the various semantics, we split the semantic description in two parts: a set of rules for *potential* transitions of *LinCa*-processes (Figures 4 and 6) and an additional rule to establish the semantics in which we check if some *potential* transition is allowed under the present *Tuple Space* configuration (Figures 5, 7 and 9).

This allows us to reuse the rules in Figure 4 (henceforth called *pure syntax* rules) for the succeeding *MTS* and *MTS-mp* semantics. Choosing this representation makes it convenient to point out common features and differences of the discussed semantics.

In contrast to [BGLZ04] we label transitions. We have to do so to record which actions a step-transition performs in order to check if this is possible under the present *Tuple Space* configuration. The labels serve as a link between the rules of *pure syntax* and the semantic rule: For a *potential* transition $P \stackrel{(I,O,R)}{\rightarrow} P'$ the multisets $I/O/R$ contain the tuples on which we want to perform in/out/rd actions. In MTS (see Figure 7), such a *potential* transition is only valid for some *Tuple Space* configuration M, if $I \uplus set(R) \subseteq M$, i.e. M includes enough instances of each tuple to satisfy all performed *in*-actions and at least one additional instance for the performed *rd*-actions on that tuple (if any *rd*-actions are performed). For *out*-actions there is no such restriction.

In Figure 9 we use the notion of *maximality* of a *potential* transition for some *Tuple Space* configuration M. *Maximality* is given iff conditions 1) and 2) in Figure 8 hold, where 1) means, that all enabled out-actions have to be performed. 2) means, that as many of the *in* and *rd*-actions as possible have to be performed. More precisely 2.1) represents the case, that the number of instances of some tuple t in the present *Tuple Space* configuration M exceeds the number of enabled *in*-actions on that tuple. In this case all *in*-actions and all *rd*-actions have to be performed.

We define the relations $\rightarrow_{ITS}$, $\rightarrow_{MTS}$ and $\rightarrow_{MTS\text{-}mp}$ as the smallest relations satisfying the corresponding rule in Figure 5, 7 and 9, respectively.

$$\begin{array}{l}
1)\ in(t).P \overset{(\{t\},\emptyset,\emptyset)}{\rightarrow} P \\
2)\ out(t).P \overset{(\emptyset,\{t\},\emptyset)}{\rightarrow} P \\
3)\ rd(t).P \overset{(\emptyset,\emptyset,\{t\})}{\rightarrow} P \\
4)\ !\,in(t).P \overset{(\{t\},\emptyset,\emptyset)}{\rightarrow} P \mid !\,in(t).P \\
5)\ \dfrac{P \overset{(I,O,R)}{\rightarrow} P'}{P \mid Q \overset{(I,O,R)}{\rightarrow} P' \mid Q}
\end{array}$$

Fig. 4. ITS: pure syntax (symmetrical rule for 5 omitted)

$$\frac{P \overset{(I,O,R)}{\rightarrow} P' \in \mathit{ITS\text{-}Rules} \quad I \subseteq M \quad R \subseteq M}{<P,M> \overset{(I,O,R)}{\rightarrow}_{ITS} <P',(M \setminus I) \uplus O>}$$

Fig. 5. ITS

ITS-Rules 1) - 5) (from Figure 4)

$$6) \quad !\,in(t).P \overset{(\{(t,i)\},\emptyset,\emptyset)}{\rightarrow} \prod_i P \mid !\,in(t).P$$

$$7) \quad \frac{P \overset{(I_P,O_P,R_P)}{\rightarrow} P' \quad Q \overset{(I_Q,O_Q,R_Q)}{\rightarrow} Q'}{P \mid Q \overset{(I_P \uplus I_Q, O_P \uplus O_Q, R_P \uplus R_Q)}{\rightarrow} P' \mid Q'}$$

Fig. 6. MTS: pure syntax

We end this Section by modeling[1] the example mentioned in the Introduction in *LinCa*. A foreman supplies a group of workers with jobs.

Let $P := foreman \mid worker_1 \mid ... \mid worker_n$, where:

$foreman = out(object).wait.in(object).foreman$
$worker_i = rd(object).out(result_i).worker_i$

Ciancarini's original MTS semantics would allow P to evolve in a variety of ways. However, given a common clock and given that all workers can perform their *rd*-operations (as well as their internal calculation which we abstract from in LinCa) within one clock cycle, the expected/desired maximum-progress behavior of P (that has already been described in the introduction) corresponds to the one and only path in $MTS\text{-}mp[P]$.

[1] The *wait*-operator is used for ease of notation only, it is not part of the discussed language. For details on the usage of the *wait*-operator see Section 4.2.

$$\frac{P \overset{(I,O,R)}{\rightarrow} P' \in \textit{MTS-Rules} \quad (I \uplus Set(R)) \subseteq M}{<P,M> \overset{(I,O,R)}{\rightarrow}_{MTS} <P',(M \setminus I) \uplus O>}$$

Fig. 7. MTS

$$\begin{array}{l}
\quad 1)\ (t,i) \in ea_{OUT}(P) \Rightarrow (t,i) \in O \\
\wedge\ 2)\ (t,i) \in M \wedge (t,j) \in ea_{IN}(P) \wedge (t,k) \in ea_{RD}(P) \Rightarrow \\
\qquad (\quad 2.1)\ j < i \wedge (t,j) \in I \wedge (t,k) \in R \\
\qquad\ \ \vee\ 2.2)\ j \geq i \wedge (t,i) \in I \wedge (t,0) \in R \\
\qquad\ \ \vee\ 2.3)\ j \geq i \wedge (t,i-1) \in I \wedge (t,k) \in R \wedge k \geq 1\ )
\end{array}$$

Fig. 8. Cond. for *Maximality* of a trans. $P \overset{(I,O,R)}{\rightarrow} P'$ for some *Tuple Space* config. M

$$\frac{P \overset{(I,O,R)}{\rightarrow} P' \in \textit{MTS-Rules} \quad P \overset{(I,O,R)}{\rightarrow} P' \text{ is maximal for } M}{<P,M> \overset{(I,O,R)}{\rightarrow}_{\textit{MTS-mp}} <P',(M \setminus I) \uplus O>}$$

Fig. 9. MTS-mp

4 Relations Between ITS, MTS, MTS-mp

For all $P \in LinCa$ the following properties hold for the defined semantics *ITS*, *MTS* and *MTS-mp*:

- $ITS[P]$ is always a subgraph of $MTS[P]$, as the *pure syntax* rules for *ITS* in Figure 4 are a subset of those for *MTS* in Figure 6 and the way the semantics are based on (Figures 5 and 7) the *pure syntax* rules is the same.
- *MTS-mp*$[P]$ is always a subgraph of $MTS[P]$, as the *pure syntax* rules for *MTS* and *MTS-mp* are the same but for the *MTS-mp* semantics in Figure 9 we apply a stronger precondition than for the *MTS* semantics in Figure 7.

By $LinCa_{cde}$ we denote the *LinCa* language based on an extended *Tuple Space*. That is, we assume the existence of 3 designated tuples *c,d,e* that are not elements of the original *LinCa Tuple Space*. We extend our *MTS-mp* semanics to treat actions on these tuples just like any other actions in the purely syntactic description. However in Transition Systems whenever (I, O, R) consists of nothing but designated tuples we replace it by τ, the *internal* label. Whenever some *internal* actions are performed concurrently with some *observable* actions, the label of the resulting transition will simply consist of the *observable* ones.

By *MTS-mp*$[P]$ where $P \in LinCa_{cde}$ we denote its semantics as described above.

4.1 The Relation Between *ITS* and *MTS-mp*

In this subsection we define an encoding enc_{ITS}: $LinCa \rightarrow LinCa_{cde}$ and prove that $ITS[P] \preceq MTS\text{-}mp[enc_{ITS}(P)]$ holds.

enc_{ITS} is composed of the main encoding $\widetilde{enc}_{ITS}$ and a parallel $out(c)$:

$\widetilde{enc}_{ITS}(0) = 0$
$\widetilde{enc}_{ITS}(\mathrm{act(t).P}) = \mathrm{in(c).act(t).out(c)}.enc(\mathrm{P})$
$\widetilde{enc}_{ITS}(\mathrm{P} \mid \mathrm{Q}) = enc(\mathrm{P}) \mid enc(\mathrm{Q})$
$\widetilde{enc}_{ITS}(!\, in(t).P) = !\, in(c).in(t).out(c).enc(P)$

$enc_{ITS}(P) = \widetilde{enc}_{ITS}(P) \mid out(c)$

Theorem 1. $ITS[P] \preceq MTS\text{-}mp[enc_{ITS}(P)]$

Proof. 1) *Weak Similarity*
$enc_{ITS}(P)$ puts a prefix $in(c)$ in front of and a suffix $out(c)$ behind every action in P. The weak step simulation deterministically starts by performing the *internal* action $out(c)$ and subsequently simulates every step of the ITS Transition System by performing three steps as follows:

First, we remove the encoding-produced guarding $in(c)$-prefix from the *observable* action we want to simulate (henceforth we call this *unlocking an action*) then we perform this action and finally we perform the suffix $out(c)$ to supply the *Tuple Space* configuration with the tuple c for the simulation of the next action. As all described steps are indeed maximal, the transitions are valid for *MTS-mp*.

2) *Equality of traces*
$traces(ITS[P]) \subseteq traces(MTS\text{-}mp[enc_{ITS}(P)])$ follows immediately from weak similarity. As for the reverse inclusion: $MTS\text{-}mp[enc_{ITS}(P)]$ can either unlock an action that can be performed under the present *Tuple Space* configuration then $ITS[P]$ can perform the same action directly. $MTS\text{-}mp[enc_{ITS}(P)]$ could also unlock an action that is blocked under the present *Tuple Space* configuration, but in this case the computation (and thus the trace) halts due to the total blocking of the process $enc_{ITS}(P)$ (as the single instance of tuple c has been consumed without leaving an opportunity to provide a new one).

4.2 The Relation Between *MTS* and *MTS-mp*

First, we introduce the basic encoding enc: $LinCa \rightarrow LinCa_{cde}$, that simply prefixes every action of a process with an additional blocking $in(c)$ action.

$enc(0) = 0$
$enc(act(t).P) = in(c).act(t).enc(P)$
$enc(P \mid Q) = enc(P) \mid enc(Q)$
$enc(!\, in(t).P) = !\, in(c).in(t).enc(P)$

Second, we introduce the encoding $\widetilde{enc}_{MTS}$ which encodes a process by *enc* and provides it with an additional parallel process $\tilde{P}$. All actions performed by $\tilde{P}$ are *internal* actions, and $\tilde{P}$ will be able to produce an arbitrary number of instances of the tuple c simultaneously.

We define: $\tilde{P} := \quad !\, in(d).[rd(e).out(c) \mid out(d)]$
$\qquad\qquad\qquad\quad \mid\, !\, in(d).out(e).wait.in(e).wait.out(d)$

$\widetilde{enc}_{MTS}(P) := enc(P) \mid \tilde{P} \mid out(d)$

Strictly speaking the *wait*-operator used in $\tilde{P}$ is not included in *LinCa*. We nevertheless use it because a *wait*-action (which has no other effect on the rest of the process and is not *observable*) can be implemented by a *rd*-action in the following way. Let t^* be a designated tuple that is not used for other purposes. If P is a *LinCa*-process except for the fact, that it may contain some *wait*-actions then we consider it as the process $P[wait/rd(t^*)] \mid out(t^*)$. However, we state that the *wait*-actions are not at all needed for the correctness of the encoding and we added them only for ease of proofs and understanding.

We now define the final encoding enc_{MTS}, that adds the parallel process $out(d)$ with the single purpose to put a tuple d into the initially empty *Tuple Space* configuration to activate the process $\tilde{P}$.

Theorem 2. *MTS[P]* $\preceq$ *MTS-mp[*$enc_{MTS}(P)$*]*

Proof. 1) *Weak similarity*
The proof is similar to that of Theorem 1. Whenever we want to simulate some step $< P, M > \stackrel{(I,O,R)}{\rightarrow}_{MTS} < P', M' >$ (where $|I|+|O|+|R| = z$) $\tilde{P}$ first produces z processes $rd(e).out(c)$ by subsequently performing z times $in(d)$ and $out(d)$ in line 1 of $\tilde{P}$. Then line 2 of $\tilde{P}$ is performed, i.e. the tuple e is provided and then read simultaneously by the z $rd(e).out(c)$-processes (and deleted by $in(e)$ immediately afterwards). This causes the simultaneous production of z instances of c, which are used to unlock the desired actions in $enc(P)$ in the subsequent step. As the step we want to simulate is valid in MTS and as all other actions (besides the second *internal wait*-action of $\tilde{P}$ that is in fact performed simultaneously) are still blocked by their prefixes $in(c)$ the step is also *maximal* and thus it is valid in *MTS-mp*.

2) *Equality of traces*
Again, $traces(ITS[P]) \subseteq$ *traces(MTS–mp*$[enc_{ITS}(P)])$ follows immediately from weak similarity. We give a sketch of the proof of the reverse inclusion:

The process $\tilde{P}$ performs some kind of loop in which it continuously produces arbitrary numbers of instances of the tuple c (let the number of produced c's be z). In the subsequent step (due to our maximality-request) as many actions $in(c)$ as possible are performed. The actual number of these *unlockings* is restricted either by the number of enabled $in(c)$ processes (let this number be x, i.e. $(c, x) \in ea_{IN}(enc(P)))$ in case $x \leq z$ or by the number of instances of c that we have produced in case $x > z$.

In the next step we perform as many unlocked actions as possible. That might be all of them, if the present *Tuple Space* configuration M allows for it, or a subset of them. In any of those cases, the same set of actions can instantly be performed in $MTS[P]$ and it simply remains to show that neither the overproduction of c's, nor the unlocking of more actions than we can simultaneously perform under M will ever enable any *observable* actions, that are not already enabled in $MTS[P]$. To show this, we define a relation R' that includes all pairs $(< P, M >, < enc_{MTS}(P), M \uplus \{d\} >)$ as well as any pair $(< P, M >, s')$ where s' is a derivation from $< enc_{MTS}(P), M \uplus \{d\} >$ by τ-steps, and show, that whenever $(s_1, s_2) \in R'$ and s_2 performs an *observable* step in $MTS\text{-}mp[enc_{MTS}(P)]$, s_1 will be ready to imitate it in $MTS[P]$.

5 Termination and Divergence Are Undecidable in MTS-mp-LinCa

5.1 RAMs

A Random Access Machine (RAM) $\hat{M}$ [SS63] consists of m registers, that may store arbitrarily large natural numbers and a program (i.e. sequence of n enumerated instructions) of the form:

$$\begin{array}{c} I_1 \\ I_2 \\ \vdots \\ I_n \end{array}$$

Each I_i is of one of the following types (where $1 \leq j \leq m, s \in \mathbb{N}$):

a) $i : Succ(r_j)$
b) $i : DecJump(r_j, s)$

A configuration of $\hat{M}$ can be represented by a tuple $< v_1, v_2, ..., v_m, k > \in N^{m+1}$, where v_i represents the value stored in r_i and k is the number of the command line that is to be computed next.

Let $\hat{M}$ be a RAM and $c =< v_1, v_2, ..., v_m, k >$ the present configuration of $\hat{M}$.

Then we distinguish the following three cases to describe the possible transitions:

1) $k > n$ means that $\hat{M}$ halts, because the instruction that should be computed next doesn't exist. This happens after computing instruction I_n and passing on to I_{n+1} or by simply jumping to a nonexistent instruction.
2) if $k \in \{1, ..., n\} \wedge I_k = Succ(r_j)$ then v_j and k are incremented, i.e. we increment the value in register r_j and succeed with the next instruction.
3) if $k \in \{1, ..., n\} \wedge I_k = DecJump(r_j, s)$ then $\hat{M}$ checks whether the value v_j of r_j is > 0. In that case, we decrement it and succeed with the next instruction (i.e. we increment k). Else (i.e. if $v_j = 0$) we simply jump to instruction I_s, (i.e. we assign $k := s$).

We say a RAM $\hat{M}$ with starting configuration $< v_1, v_2, ..., v_m, k >$ terminates if its (deterministic) computation reaches a configuration that belongs to case 1). If such a configuration is never reached, the computation never stops and we say that $\hat{M}$ diverges. It is well-known [M67] that the question whether a RAM diverges or terminates under a starting configuration $< 0, ..., 0, 1 >$ is undecidable for the class of all RAMs.

It is quite obvious, that for those *LinCa*-dialects that include a predicative *in*-operator $inp(t)?P\_Q$ (with semantical meaning *if* $t \in$ *TS then P else Q*, for details see e.g. [BGM00]) the questions of termination and divergence are undecidable (moreover those dialects are even Turing complete), as for any RAM there is an obvious deterministic encoding.

However neither the original *Linda Calculus* [CJY94] nor the discussed variant (adopted from [BGLZ04]) include such an operator and the proof that neither termination nor divergence are decidable under the *MTS-mp* semantics is more difficult.

We will define encodings *term* and *div* that map *RAM*s to *LinCa*-processes such that a *RAM* $\hat{M}$ terminates (diverges) iff the corresponding Transition System $MTS\text{-}mp[term(\hat{M})]$ ($MTS\text{-}mp[div(\hat{M})]$) terminates (diverges).

While the computation of $\hat{M}$ is completely deterministic, the transitions in the corresponding *LTS* given by our encoding may be nondeterministic. Note that every time a nondeterministic choice is made, there will be one transition describing the simulation of $\hat{M}$, and one transition that will compute something useless. For ease of explanations in Sections 5.2 and 5.3 we call the first one *right* and the second *wrong*.

To guarantee that the part of the *LTS* that is reached by a *wrong* transition (that deviates from the simulation) does not affect the question of termination (divergence) we will make sure that all traces of the corresponding subtree are infinite (finite). This approach guarantees that the whole *LTS* terminates (diverges) iff we reach a finite (an infinite) trace by keeping to the *right* transitions.

Our encodings establish a natural correspondence between *RAM* configurations and *Tuple Space* configurations, i.e. the *RAM*-configuration $< v_1, v_2, ..., v_m, k >$ belongs to the *Tuple Space* configuration $\{(r_1, v_1), ..., (r_m, v_m), p_k\}$. For a *RAM* configuration c we refer to the corresponding *Tuple Space* configuration by $TS(c)$.

Theorem 3 (RAM Simulation). *For every RAM $\hat{M}$ the Transition System MTS-mp[term($\hat{M}$)] (MTS-mp[div($\hat{M}$)]) terminates (diverges) iff $\hat{M}$ terminates (diverges) under starting configuration* $< 0, ..., 0, 1 >$.

5.2 Termination Is Undecidable in MTS-mp-LinCa

Let *term*: $RAMs \rightarrow LinCa$ be the following mapping:

$$term(\hat{M}) = \prod_{i \in \{1,...,n\}} [I_i] \mid !\, in(div).out(div) \mid in(loop).out(div) \mid out(p_1)$$

where the encoding $[I_i]$ of a *RAM*-Instruction in *LinCa* is:

$$
\begin{array}{rl}
[i : Succ(r_j)] \;\; = & !\, in(p_i).out(r_j).out(p_{i+1}) \\
[i : DecJump(r_j, s)] = & !\, in(p_i).[\; out(loop) \mid in(r_j).in(loop).out(p_{i+1})\;] \\
\mid & !\, in(p_i).[\; in(r_j).out(loop) \\
& \qquad\qquad \mid wait.wait.out(r_j).in(loop).out(p_s)\;]
\end{array}
$$

Note that the first (deterministic) step of $term(\hat{M})$ will be the initial $out(p_1)$. The resulting *Tuple Space* configuration is $\{p_1\} = TS(< 0, ..., 0, 1 >)$. For ease of notation, we will henceforth also denote the above defined process where $out(p_1)$ has already been executed by $term(\hat{M})$.

We now describe (given some RAM $\hat{M}$ and configuration c) the possible transition sequences from some state $< term(\hat{M}), TS(c) >$ in $MTS\text{-}mp[term(\hat{M})]$. In cases 1 and 2 the computation in our LTS is completely deterministic and performs the calculation of $\hat{M}$. In case 3 the transition sequence that simulates *DecJump(r_j,s)* includes nondeterministic choice. As described in Subsection 5.1 performing only *right* choices (cases 3.1.1 and 4.1.1) results in an exact simulation of $\hat{M}$'s transition $c \rightarrow_{\hat{M}} c'$, i.e. the transition sequence leads to the corresponding state $< term(\hat{M}), TS(c') >$. Performing at least one *wrong* choice (cases 3.1.2, 3.2, 4.1.2 and 4.2) causes the subprocess $!\; in(div).out(div)$ to be activated, thus assuring that any computation in the corresponding subtree diverges (denoted by $\rightsquigarrow$). (In this case other subprocesses are not of concern because they can't interfere by removing the tuple div, so we substitute these subprocesses by "...".)

1. $k > n$, i.e. $\hat{M}$ has terminated. Then $<term(\hat{M}), TS(c)>$ is totally blocked.
2. $k \in \{1, ..., n\} \wedge I_k = k : Succ(r_j)$, then $\hat{M}$ increments both r_j and k. The corresponding transition sequence in $MTS\text{-}mp[term(\hat{M})]$ is:

$$
\begin{array}{l}
\quad <term(\hat{M}), TS(c)> \\
\rightarrow <term(\hat{M}) \mid out(r_j).out(p_{k+1}), TS(c) \setminus \{p_k\}> \\
\rightarrow <term(\hat{M}) \mid out(p_{k+1}), TS(c) \setminus \{p_k\} \uplus \{r_j\}> \\
\rightarrow <term(\hat{M}), TS(c) \setminus \{p_k\} \uplus \{r_j, p_{k+1}\}> \\
= \; <term(\hat{M}), TS(c')>
\end{array}
$$

3. $k \in \{1, ..., n\} \wedge I_k = k : DecJump(r_j, s) \wedge v_j \neq 0$, then $\hat{M}$ decrements r_j and increments k. The possible transition sequences in $MTS\text{-}mp[term(\hat{M})]$ are: $<term(\hat{M}), TS(c)> \stackrel{nondet.}{\rightarrow}$

3.1 **right**:

$$
\begin{array}{l}
\quad <term(\hat{M}) \mid out(loop) \mid in(r_j).in(loop).out(p_{k+1}), TS(c) \setminus \{p_k\}> \\
\rightarrow <term(\hat{M}) \mid in(loop).out(p_{k+1}), TS(c) \setminus \{p_k, r_j\} \uplus \{loop\}> \stackrel{nondet.}{\rightarrow}
\end{array}
$$

3.1.1 ***right - right:***

$$
\begin{array}{l}
\quad <term(\hat{M}) \mid out(p_{k+1}), TS(c) \setminus \{p_k, r_j\}> \\
\rightarrow <term(\hat{M}), TS(c) \setminus \{p_k, r_j\} \uplus \{p_{k+1}\}> \\
= <term(\hat{M}), TS(c')>
\end{array}
$$

3.1.2 ***right - wrong:***

$$
\begin{array}{l}
\quad <term(\hat{M}) \mid in(loop).out(p_{k+1}), TS(c) \setminus \{p_k, r_j\} \uplus \{loop\}> \\
\rightarrow <... \mid out(div), TS(c) \setminus \{p_k, r_j\}> \rightsquigarrow
\end{array}
$$

3.2 ***wrong***:

$<term(\hat{M}) \mid in(r_j).out(loop) \mid wait^2.out(r_j).in(loop).out(p_s), TS(c) \setminus \{p_k\}>$
$\rightarrow <term(\hat{M}) \mid out(loop) \mid wait.out(r_j).in(loop).out(p_s), TS(c) \setminus \{p_k, r_j\}>$
$\rightarrow <term(\hat{M}) \mid out(r_j).in(loop).out(p_s), TS(c) \setminus \{p_k, r_j\} \uplus \{loop\}>$
$\rightarrow <... \mid out(div), TS(c) \setminus \{p_k\}> \rightsquigarrow$

4. $k \in \{1, ..., n\} \wedge I_k = k : DecJump(r_j, s) \wedge v_j = 0$, then $\hat{M}$ assigns $k := s$
$<term(\hat{M}), TS(c)> \overset{nondet.}{\rightarrow}$

4.1 ***right:***

$\rightarrow <term(\hat{M}) \mid in(r_j).out(loop) \mid wait^2.out(r_j).in(loop).out(p_s), TS(c) \setminus \{p_k\}>$
$\rightarrow <term(\hat{M}) \mid in(r_j).out(loop) \mid wait.out(r_j).in(loop).out(p_s), TS(c) \setminus \{p_k\}>$
$\rightarrow <term(\hat{M}) \mid in(r_j).out(loop) \mid out(r_j).in(loop).out(p_s), TS(c) \setminus \{p_k\}>$
$\rightarrow <term(\hat{M}) \mid in(r_j).out(loop) \mid in(loop).out(p_s), TS(c) \setminus \{p_k\} \uplus \{r_j\}>$
$\rightarrow <term(\hat{M}) \mid out(loop) \mid in(loop).out(p_s), TS(c) \setminus \{p_k\}>$
$\rightarrow <term(\hat{M}) \mid in(loop).out(p_s), TS(c) \setminus \{p_k\} \uplus \{loop\}> \overset{nondet.}{\rightarrow}$

4.1.1 ***right - right:***

$<term(\hat{M}) \mid out(p_s), TS(c) \setminus \{p_k\}>$
$\rightarrow <term(\hat{M}), TS(c) \setminus \{p_k\} \uplus \{p_s\}>$
$= <term(\hat{M}), TS(c')>$

4.1.2 ***right - wrong:***

$<... \mid out(div), TS(c) \setminus \{p_k\}> \rightsquigarrow$

4.2 ***wrong:***

$<term(\hat{M}) \mid out(loop) \mid in(r_j).in(loop).out(p_{k+1}), TS(c) \setminus \{p_k\}>$
$\rightarrow <term(\hat{M}) \mid in(r_j).in(loop).out(p_{k+1}), TS(c) \setminus \{p_k\} \uplus \{loop\}>$
$\rightarrow <... \mid out(div), TS(c) \setminus \{p_k\}> \rightsquigarrow$

5.3 Divergence Is Undecidable in MTS-mp-LinCa

Let *div*: $RAMs \rightarrow LinCa$ be the following mapping:

$$div(\hat{M}) = \prod_{i \in \{1,...,n\}} [I_i] \mid in(flow) \mid out(p_1)$$

where the encoding $[I_i]$ of a *RAM*-Instruction in *LinCa* is:

$$\begin{array}{rcl} [i : Succ(r_j)] & = & !\, in(p_i).out(r_j).out(p_{i+1}) \\ [i : DecJump(r_j, s)] & = & !\, in(p_i).in(r_j).out(p_{i+1}) \\ & \mid & !\, in(p_i).[\, in(r_j).out(flow) \\ & & \qquad \mid wait^2.out(r_j).in(flow).out(p_s)\,] \end{array}$$

Note that the first (deterministic) step of $div(\hat{M})$ will be the initial $out(p_1)$. The resulting *Tuple Space* configuration is $\{p_1\} = TS(< 0, ..., 0, 1 >)$. For ease of notation, we will henceforth also denote the above defined process where $out(p_1)$ has already been executed by $div(\hat{M})$.

We now describe (given some *RAM* $\hat{M}$ and configuration c) the possible transition sequences from some state $< div(\hat{M}), TS(c) >$ in *MTS-mp*$[div(\hat{M})]$. In cases 1 and 2 the computation in our *LTS* is completely deterministic and performs the calculation of $\hat{M}$. In case 3 the transition sequence that simulates *DecJump(*r_j*,s)* includes nondeterministic choice. As described in Subsection 5.1 performing only *right* choices (cases 3.1 and 4.1.1) results in an exact simulation of $\hat{M}$s transition $c \rightarrow_{\hat{M}} c'$, i.e. the transition sequence leads to the corresponding state $< div(\hat{M}), TS(c') >$. Performing at least one *wrong* choice (cases 3.2, 4.1.2 and 4.2) causes the tuple *flow* to be removed from the *Tuple Space* configuration, thus leading to some state $< P, M >$ where P is totally blocked under M, denoted by $< P, M > \not\rightarrow$. For cases 1 and 2 see preceding subsection.

3. $k \in \{1,..,n\} \wedge I_k = k : DecJump(r_j, s) \wedge v_j \neq 0$, then $\hat{M}$ decrements r_j and increments k. The possible transition sequences in *MTS-mp*$[div(\hat{M})]$ are:
$<div(\hat{M}), TS(c)> \overset{nondet.}{\rightarrow}$

3.1 ***right:***

$$
\begin{aligned}
& <div(\hat{M}) \mid in(r_j).out(p_{k+1}), TS(c) \setminus \{p_k\}> \\
\rightarrow & <div(\hat{M}) \mid out(p_{k+1}), TS(c) \setminus \{p_k, r_j\}> \\
\rightarrow & <div(\hat{M}), TS(c) \setminus \{p_k, r_j\} \uplus \{p_{k+1}\}> \\
= & <div(\hat{M}), TS(c')>
\end{aligned}
$$

3.2 ***wrong:***

$$
\begin{aligned}
& <div(\hat{M}) \mid in(r_j).out(flow) \mid wait^2.out(r_j).in(flow).out(p_s), TS(c) \setminus \{p_k\}> \\
\rightarrow & <div(\hat{M}) \mid out(flow) \mid wait.out(r_j).in(flow).out(p_s), TS(c) \setminus \{p_k, r_j\}> \\
\rightarrow & <div(\hat{M}) \mid out(r_j).in(flow).out(p_s), TS(c) \setminus \{p_k, r_j\} \uplus \{flow\}> \\
\rightarrow & <\Pi\ [I_i] \mid in(flow).out(p_s), TS(c) \setminus \{p_k\}> \not\rightarrow
\end{aligned}
$$

4. $k \in \{1, ..., n\} \wedge I_k = k : DecJump(r_j, s) \wedge v_j = 0$, then $\hat{M}$ assigns $k := s$
$<div(\hat{M}), TS(c)> \overset{nondet.}{\rightarrow}$

4.1 ***right:***

$$
\begin{aligned}
& <div(\hat{M}) \mid in(r_j).out(flow) \mid wait^2.out(r_j).in(flow).out(p_s), TS(c) \setminus \{p_k\}> \\
\rightarrow & <div(\hat{M}) \mid in(r_j).out(flow) \mid wait.out(r_j).in(flow).out(p_s), TS(c) \setminus \{p_k\}> \\
\rightarrow & <div(\hat{M}) \mid in(r_j).out(flow) \mid out(r_j).in(flow).out(p_s), TS(c) \setminus \{p_k\}> \\
\rightarrow & <div(\hat{M}) \mid in(r_j).out(flow) \mid in(flow).out(p_s), TS(c) \setminus \{p_k\} \uplus \{r_j\}> \\
\rightarrow & <div(\hat{M}) \mid out(flow) \mid in(flow).out(p_s), TS(c) \setminus \{p_k\}> \\
\rightarrow & <div(\hat{M}) \mid in(flow).out(p_s), TS(c) \setminus \{p_k\} \uplus \{flow\}> \overset{nondet.}{\rightarrow}
\end{aligned}
$$

4.1.1 ***right - right:***

$$
\begin{aligned}
& <div(\hat{M}) \mid out(p_s), TS(c) \setminus \{p_k\}> \\
\rightarrow & <div(\hat{M}), TS(c) \setminus \{p_k\} \uplus \{p_s\}> \\
= & <div(\hat{M}), TS(c')>
\end{aligned}
$$

4.1.2 ***right - wrong:***

$$<\Pi\ [I_i] \mid in(flow).out(p_s), TS(c) \setminus \{p_k\}> \not\rightarrow$$

4.2 ***wrong:***
$<div(\hat{M}) \mid in(r_j).out(p_{k+1}), TS(c) \setminus \{p_k\}> \not\rightarrow$

6 Conclusion

In order to guarantee maximum utilization of processing units in a MIMD setting, we modified Ciancarini's MTS-semantics for LinCa. We restricted the valid paths of the Multi Step Transition System to those in which in each step there are performed as many actions as possible. Pursuing the aim of maximizing the resource utilization we found it astounding that the restriction to paths satisfying the maximum progress condition causes a change in expressiveness. The fact that a *RAM* can be simulated (nondeterministically) is non-trivial for two reasons: First, we are not able to implement an if-then-else construct (or at least there is no obvious way to do that) without the usage of *predicative* tuple space operators. Second, we do not even allow for a *choice*-operator and as a consequence we have to "neutralize" remaining process-artifacts in order to prevent them from interfering with the calculation at some time in the future.

We also discussed the relation between our semantics and ITS and MTS, respectively. The outcome of our analysis is that in all future approaches of maximizing the resource utilization for LinCa in a multiple-step scenario, one has to take into account that - unpleasantly - there are programs for which termination is undecidable. Nevertheless the existence of such programs does not mean that demanding maximum progress is not meaningful or useless.

References

[BGLZ04] Mario Bravetti, Roberto Gorrieri, Roberto Lucchi, Gianluigi Zavattaro. *Adding Quantitative Information to Tuple Space Coordination Languages*, Bologna, Italy, July 04.

[BGM00] Frank S. de Boer, Maurizio Gabbrielli, Maria Chiara Meo, *A Timed Linda Language*, Lecture Notes in Computer Science, Volume 1906, Pages 299-304, Jan 2000.

[BGZ00] Nadia Busi, Roberto Gorrieri, Gianluigi Zavattaro. *On the Expressiveness of Linda Coordination Primitives* Information and Computation Vol. 156(1-2), p.90-121, January 2000.

[BZ05] Nadia Busi, Gianluigi Zavattaro. *Prioritized and Parallel Reactions in Shared Data Space Coordination Languages*, COORD05. Namur, Belgium. LNCS 3454, p.204-219, 2005.

[CJY94] Paolo Ciancarini , Keld K. Jensen , Daniel Yankelevich. *On the Operational Semantics of a Coordination Language* Selected papers from the ECOOP'94 Workshop on Models and Languages for Coordination of Parallelism and Distribution, Object-Based Models and Languages for Concurrent Systems, p.77-106, 1994.

[M67] M.L.Minksy - *Computation: finite and infinite machines*, Prentice Hall, Englewoof Cliffs, 1967.

[SS63] J.C. Sheperdson, J.E. Sturgis. *Computability of recursive functions.* Journal of the ACM, Vol. 10, p. 217-255, 1963.