Synchronous Coordination in the μLog Coordination Model

Koen De Bosschere
ELIS Department, Ghent University, Belgium
kdb@elis.rug.ac.be

Jean-Marie Jacquet
Institute of Informatics, FUNDP, Belgium
jmj@info.fundp.ac.be

ABSTRACT

In this paper we investigate the effects of synchronous communication in the μLog coordination framework. We start by defining the concept of synchronous coordination and then introduce it into the μLog framework. We formally present the transition rules for the synchronous coordination, and discuss some of the consequences of the choices we have made. From the transition rules follows that synchronous *tell* and synchronous *get* are actually very similar and can be expressed in terms of two elementary primitives: *put* and *wait*. It turns out that these two primitives are powerful enough to also support the other μLog communication primitives.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Parallel Programming; D.1.6 [**Programming Techniques**]: Logic Programming; D.4.4 [**Operating Systems**]: Communications Management

General Terms

Languages, Theory

Keywords

Coordination, μLog, semantics, synchronicity

1. INTRODUCTION

When comparing coordination languages, one quickly comes to the conclusion that most proposals only differ in the set of coordination operations they offer [8]. Some operations work on a point-to-point basis, while others are working on a global data space; some communication operations can have complex embedded conditions; the communication can be asynchronous or synchronous.

According to Bal [2], asynchronous communication is believed to be less deadlock-prone than synchronous communication as producers will never block. In practice, most coordination languages are asynchronous. There are however some notable exceptions: the programming languages Ada (Rendez-vous), Delta-Prolog [7], Quintus Prolog [16], and PMS-prolog [21] have synchronous communication primitives. In logic programming languages, the choice for synchronous communication is often motivated by the requirement to support unification during communication.

In this paper we investigate the effects of a synchronous coordination model in μLog, and prove that synchronous communication creates similarity between producers and consumers, up to the point that producers and consumers could even be interchanged under certain conditions. This raises the question whether there is redundancy in the set of coordination operations, and whether they can be reduced to a smaller set. It turns out that one primitive to create suspension terms, together with an active blackboard that can autonomously combine suspension terms and notify a generalized wait-primitive are sufficient to express the complete set of synchronous and asynchronous coordination primitives in μLog. To the best of our knowledge, this approach is novel, and has not been presented elsewhere. This paper is based on the μLog coordination model that we have developed earlier [9, 13, 14].

We first start by introducing the basic μLog framework, followed by the introduction of a synchronous *tell*-primitive. We then formally present the transition rules for all communication primitives. In section 5, we investigate the similarities between synchronous *tell* and *get*, and this leads us to a more primitive set of communication primitives (*put* and *wait*) that allows us to to express the full set of μLog communication primitives. The paper is concluded with a conclusion and an overview of related work.

2. THE μLOG COORDINATION MODEL

The μLog coordination model is based on communication with a global data space or blackboard. It has evolved from Multi-Prolog, a Prolog-based coordination language. On the blackboard, there are three primitives that deal with terms, namely, *t*ellt to put a term on a blackboard, *g*ett to remove a term from the blackboard, and *r*eadt to check the presence of a term on the blackboard. Similar primitives (*tellp*, *getp*, and *readp*) exist that work on processes (to create a process, to remove it, and to check its presence). In this paper, we will concentrate on *t*ellt-, *g*ett-, and *r*eadt-primitives, which we will call *tell*, *get* and *read* for short. Their informal semantics is as follows:

SAC 2001 Las Vegas, NV USA

1. *get* consumes a term on the blackboard, and if no suitable term is available, it suspends until a suitable term is being told on the blackboard (this is comparable to the Linda `in`-primitive).
2. *read* checks the presence of a term on the blackboard, and if no suitable term is available, it suspends until a suitable term is being told on the blackboard (this is comparable to the Linda `rd`-primitive).
3. *tell* puts a term on the blackboard, where it resides until it is consumed by a *get*. *tell* never suspends, but immediately succeeds after having told the term to the blackboard (this is comparable to the Linda `out`-primitive) [1]. Before physically telling a term on the blackboard, the μLog system will first try to directly communicate it to a suspended primitive (first to the suspended *read*-primitives as these do not consume the term, and afterwards to a *get*-primitive). If there is a suspended *get*-primitive, the term will thus never appear on the blackboard.

The semantics of this basic μLog coordination framework has been described elsewhere [13]. We will now change the semantics of the *tell*-primitive.

3. SYNCHRONOUS *tell*

The informal semantics of the synchronous *tell*-primitive is as follows:

> *tell* produces a term on the blackboard, and if no suitable get-operation is waiting, *tell* suspends until the term that is told on the blackboard is consumed by the appropriate *get*-operation.

This behavior is quite different from the *tell*-behavior in the basic μLog model. There are some immediate consequences.

1. Since *tell* cannot continue unless its term has been consumed, there is no more need for a blackboard as a term buffer. We will however keep the blackboard as synchronization buffer (see below).
2. Since there is no more need to store terms on the blackboard, the *get*-operations and *read*-operations do not have to look for a term on the blackboard anymore, but for a suspended *tell*-operation.
3. The basic operation of *tell* and *get* is now more similar. Both can suspend, and be resumed by the other one. The only major difference is that *tell* produces a value, while *get* consumes a value. If we would use full term unification to find a matching term, the communication could be bi-directional, or in other words, *tell* could be used to get information, while *get* could be used to produce information. In that case, *tell* and *get* can actually be interchanged. For the sake of simplicity we will limit terms to ground terms in this paper.

In order to be able to manipulate suspended operations, we here introduce the concepts of suspension record and suspension term [11]. A suspension record is a Multi-Prolog implementation detail that is made explicitly visible in the μLog coordination model. A suspension record contains all the information that is needed to resume a suspended process, such as the term on which it is suspended. This information is made visible in the μLog framework by means of a so-called suspension term. On suspension, a suspension term is put on the blackboard.

1. the suspension term for a blocked *get*-operation on a term t_i is represented by $\gamma(t_i)$,
2. the suspension term for a blocked *read*-operation on a term t_i is represented by $\rho(t_i)$,
3. and the suspension term for a blocked *tell*-operation on a term t_i is represented by $\tau(t_i)$.

The basic blackboard operations on terms are generalized to also deal with suspension terms. In particular, a suspension term can be removed (*get*), forcing the associated process to backtrack.

The suspension term σ can disappear on two occasions:

1. a $get(\sigma)$ is executed, in which case the suspended operation fails,
2. a matching blackboard operation is issued in which case the associated operation succeeds. Hence, $\tau(t_i)$ is resolved by $get(t_i)$, while $\gamma(t_i)$ is resolved by $tell(t_i)$.

4. FORMAL TREATMENT

The intuition just sketched may be abstracted in very general terms by assuming the existence of a set of tokens, this set being Prolog terms as in the μLog model or tuples in the case of Linda. Based on it, we define the communication primitives *tell*, *get* and *read* by the following grammar:

$$\begin{aligned} \sigma &::= t \mid \tau(t) \mid \gamma(\sigma) \mid \rho(\sigma) \\ C &::= tell(t) \mid get(\sigma) \mid read(\sigma) \end{aligned}$$

There, t denotes a token and C a communication primitive. It is worth noting that we allow *get* and *read* primitives to take tokens as well as suspension terms as arguments whereas we limit the *tell* primitive to tokens only. This restriction is motivated by the fact that suspension terms are the side-effect of a suspended communication primitive, and we do not want this side effect being created by anything but a suspended communication primitive. The semantics of the *get* and *read* primitives is extended to also work on suspension terms.

The statements of our abstract language $\mathcal{L}$ are defined as follows:

$$S ::= C \mid S\,;\,S \mid S + S$$

The symbols ; and + denote the sequential composition operator and the sequential choice operator.[1]

[1]By sequential choice we mean that first the left alternative is tried, and only if this fails, the right alternative is tried. This semantics allows us to model the backtracking semantics of e.g. Prolog.

The careful reader will have noticed that we only treat finite processes. This is done for simplicity of the presentation. Our model can be naturally generalized to infinite processes using classical techniques, as exemplified for instance in [12].

This given, a computation consists of the parallel resolution of statements. It may be described by using transition rules written in Plotkin's style [19]. The configurations

$$\langle \mathcal{R} \mid bb \rangle$$

to be considered here are composed of a multiset of statements $\mathcal{R}$, together with a multiset of suspension terms bb. The latter denotes the current contents of the blackboard during the computation. The former denotes the set of statements executed in parallel. To allow for the modelling of backtracking, statements are generalized so as to make alternative continuations explicit. This is done according to the following rules, where S denotes a statement.

$$\begin{aligned} SS &::= \Box \mid S \cdot SS \\ AC &::= \Delta \mid SS \diamond AC \end{aligned}$$

Sequences of statements are thus represented by the symbol SS with $\Box$ denoting the empty sequence and "$\cdot$" denoting the concatenation operator. Alternative statements are represented by the AC symbol with Δ denoting the empty stack of alternatives. Hence, the multiset of statements actually consists of elements of AC. The "$\cdot$" and "$\diamond$" are the semantic operators corresponding to the " ; " and " + " syntactic operators, respectively.

The transition rules are defined by case analysis of the first statement of the alternative statement for the process under consideration. They are listed in Figures 1-4.

Rules of Figure 1 define the reduction of *tell*-primitives. Rule T_1 copes with telling a token on the blackboard in the presence of a suspension term $\rho(t) \rightsquigarrow Y$. The notation $\rho(\sigma) \rightsquigarrow X$ is used to specify a suspension term, consisting of a generalized term σ, and a continuation X. This continuation is needed to be able to resume the process when the coordination primitive is unblocked (and either succeeds or fails). In this case, the process associated with the suspension term is resumed (Y). Rule T_2 deals with telling a token t on the blackboard in the presence of a blocked *get*-primitive. In this case, the process associated with the blocked *get*-primitives is resumed, and the *tell*-operation succeeds. Notice that the primitive $tell(t)$ is only consumed in Rule T_2 in the presence of a suspension term for a *get*-operation while it is not in Rule T_1 for a suspension term corresponding to a suspended *read*-operation. Rule T_3 describes the case where there is a process waiting for the suspension term $\tau(t)$ to appear on the blackboard. The process will have no further impact on the *tell*-operation, and simply resume (Y). Rule T_4 considers the case where a process is waiting to kill the suspended *tell*-operation. As a result, the *tell*-operation backtracks, continuing with AC, and the suspended *get*-operation resumes its own execution (Y). Finally, T_5 creates the τ-suspension term.

Rules of Figure 2 handle the *get*-primitives. The first two rules deal with *get*-primitives for which there is a matching term (either token or suspension term) on the blackboard. As a consequence, the corresponding primitives succeed. Rules $G_3 - G_5$ deal with the case in which there is no matching term on the blackboard, but there are matching meta-terms ($\rho(\gamma(\sigma))$ or $\gamma(\gamma(\sigma))$). First of all, G_3 tries to resume all suspended *read*-primitives. Then, if no more ρ suspension terms are found, either Rule G_4 or G_5 is applied, depending on the presence or absence of a γ suspension term. Here too, when there is a choice between resuming a *get*-primitive either by consuming its argument, or allowing it to be canceled by a meta-primitive, we prefer the former option. The rationale behind it is that the meta-primitive requires a suspension term to succeed, but a suspension term should only be created when a primitive suspends.

Rules of Figure 3 are completely analogous to the G_i-rules of Figure 2.

Rules C and S in Figure 4 finally describe the transition rules for sequential choice and sequential composition of communication primitives in the classical way.

Note that it is easy to verify that Rules T_4 and G_2 are dual rules. Rule T_4 describes the effect of a *tell*-operation when a get-operations is waiting to force the *tell*-operation to fail. Rule G_2 describes the effect of such a *get*-operation on an existing suspension record (in this case for a *tell*-operation).

The careful reader will have realized that the transition rules suggest a global state. However, they can be generalized to a distributed model following the lines of [10]. This has not been done in this paper because it is an orthogonal issue.

5. TOWARDS COORDINATION WITH TWO PRIMITIVES

Given the similarity between the synchronous *get* and *tell*-operations, we can try to further reduce the number of blackboard operations in the synchronous μLog-model.

Let's first start by comparing the synchronous *tell* and *get*-operations. In the following example

$$\begin{aligned} &sendtoken = tell(token) \\ &receivetoken = get(token) \end{aligned}$$

sendtoken and *receivetoken* can be used to realize an elementary synchronization between two processes. Due to the similarity between *get* and *tell*, the same synchronization could be realized by means of the following implementation:

$$\begin{aligned} &sendtoken = get(token) \\ &receivetoken = tell(token) \end{aligned}$$

Depending on the relative order of execution, one process will put a $\gamma(token)$, resp. $\tau(token)$ on the blackboard, which will be consumed by the corresponding $tell(token)$, resp. $get(token)$. Hence, in order to realize this type of synchronization, the only condition is that the two communication primitives are complementary.

However, the actual execution is internally non-deterministic. This means that although the result of the computation is

$$(T_1)\quad \langle\{(tell(t)\ \cdot\ SS)\ \diamond\ AC\}\cup\mathcal{R}\mid bb\cup\{\rho(t)\rightsquigarrow Y\}\rangle \rightarrow \langle\{(tell(t)\ \cdot\ SS)\ \diamond\ AC\}\cup\{Y\}\cup\mathcal{R}\mid bb\rangle$$

$$(T_2)\quad \frac{\forall X(\rho(t)\rightsquigarrow X)\notin bb}{\langle\{(tell(t)\ \cdot\ SS)\ \diamond\ AC\}\cup\mathcal{R}\mid bb\cup\{\gamma(t)\rightsquigarrow Y\}\rangle \rightarrow \langle\{SS\ \diamond\ AC\}\cup\{Y\}\cup\mathcal{R}\mid bb\rangle}$$

$$(T_3)\quad \frac{\forall X(\rho(t)\rightsquigarrow X)\notin bb \quad \forall X(\gamma(t)\rightsquigarrow X)\notin bb}{\langle\{(tell(t)\ \cdot\ SS)\ \diamond\ AC\}\cup\mathcal{R}\mid bb\cup\{\rho(\tau(t))\rightsquigarrow Y\}\rangle \rightarrow \langle\{(tell(t)\ \cdot\ SS)\ \diamond\ AC\}\cup\{Y\}\cup\mathcal{R}\mid bb\rangle}$$

$$(T_4)\quad \frac{\forall X(\rho(t)\rightsquigarrow X)\notin bb \quad \forall X(\gamma(t)\rightsquigarrow X)\notin bb \quad \forall X(\rho(\tau(t))\rightsquigarrow X)\notin bb}{\langle\{(tell(t)\ \cdot\ SS)\ \diamond\ AC\}\cup\mathcal{R}\mid bb\cup\{\gamma(\tau(t))\rightsquigarrow Y\}\rangle \rightarrow \langle\{AC\}\cup\{Y\}\cup\mathcal{R}\mid bb\rangle}$$

$$(T_5)\quad \frac{\forall X(\rho(t)\rightsquigarrow X)\notin bb \quad \forall X(\gamma(t)\rightsquigarrow X)\notin bb \quad \forall X(\rho(\tau(t))\rightsquigarrow X)\notin bb \quad \forall X(\gamma(\tau(t))\rightsquigarrow X)\notin bb}{\langle\{(tell(t)\ \cdot\ SS)\ \diamond\ AC\}\cup\mathcal{R}\mid bb\rangle \rightarrow \langle R\mid bb\cup\{\tau(t)\rightsquigarrow SS\ \diamond\ AC\}\rangle}$$

Figure 1: Tell primitives

$$(G_1)\quad \langle\{(get(t)\ \cdot\ SS)\ \diamond\ AC\}\cup\mathcal{R}\mid bb\cup\{\tau(t)\rightsquigarrow Y\}\rangle \rightarrow \langle\{SS\ \diamond\ AC\}\cup\{Y\}\cup\mathcal{R}\mid bb\rangle$$

$$(G_2)\quad \langle\{(get(\sigma)\ \cdot\ SS)\ \diamond\ AC\}\cup\mathcal{R}\mid bb\cup\{\sigma\rightsquigarrow X\ \diamond\ Y\}\rangle \rightarrow \langle\{SS\ \diamond\ AC\}\cup\{Y\}\cup\mathcal{R}\mid bb\rangle$$

$$(G_3)\quad \frac{\forall X(\tau(\sigma)\rightsquigarrow X)\notin bb \quad \forall X(\sigma\rightsquigarrow X)\notin bb}{\langle\{(get(\sigma)\ \cdot\ SS)\ \diamond\ AC\}\cup\mathcal{R}\mid bb\cup\{\rho(\gamma(\sigma))\rightsquigarrow Y\}\rangle \rightarrow \langle\{(get(\sigma)\ \cdot\ SS)\ \diamond\ AC\}\cup\{Y\}\cup\mathcal{R}\mid bb\rangle}$$

$$(G_4)\quad \frac{\forall X(\tau(\sigma)\rightsquigarrow X)\notin bb \quad \forall X(\sigma\rightsquigarrow X)\notin bb \quad \forall X(\rho(\gamma(\sigma))\rightsquigarrow X)\notin bb}{\langle\{(get(\sigma)\ \cdot\ SS)\ \diamond\ AC\}\cup\mathcal{R}\mid bb\cup\{\gamma(\gamma(\sigma))\rightsquigarrow Y\}\rangle \rightarrow \langle\{AC\}\cup\{Y\}\cup\mathcal{R}\mid bb\rangle}$$

$$(G_5)\quad \frac{\forall X(\tau(\sigma)\rightsquigarrow X)\notin bb \quad \forall X(\sigma\rightsquigarrow X)\notin bb \quad \forall X(\rho(\gamma(\sigma))\rightsquigarrow X)\notin bb \quad \forall X(\gamma(\gamma(\sigma))\rightsquigarrow X)\notin bb}{\langle\{(get(\sigma)\ \cdot\ SS)\ \diamond\ AC\}\cup\mathcal{R}\mid bb\rangle \rightarrow \langle\mathcal{R}\mid bb\cup\{\gamma(\sigma)\rightsquigarrow SS\ \diamond\ AC\}\rangle}$$

Figure 2: Get primitives

$$(R_1)\quad \frac{}{\langle \{(read(t)\ \cdot\ SS) \diamond AC\} \cup \mathcal{R} \mid bb \cup \{\tau(t) \rightsquigarrow Y\}\rangle \rightarrow \langle \{SS \diamond AC\} \cup \mathcal{R} \mid bb \cup \{\tau(t) \rightsquigarrow Y\}\rangle}$$

$$(R_2)\quad \frac{}{\langle \{(read(\sigma)\ \cdot\ SS) \diamond AC\} \cup \mathcal{R} \mid bb \cup \{\sigma \rightsquigarrow Y\}\rangle \rightarrow \langle \{SS \diamond AC\} \cup \mathcal{R} \mid bb \cup \{\sigma \rightsquigarrow Y\}\rangle}$$

$$(R_3)\quad \frac{\forall X(\tau(\sigma) \rightsquigarrow X) \notin bb \quad \forall X(\sigma \rightsquigarrow X) \notin bb}{\langle \{(read(\sigma)\ \cdot\ SS) \diamond AC\} \cup \mathcal{R} \mid bb \cup \{\rho(\rho(\sigma)) \rightsquigarrow Y\}\rangle \rightarrow \langle \{(read(\sigma)\ \cdot\ SS) \diamond AC\} \cup \{Y\} \cup \mathcal{R} \mid bb\rangle}$$

$$(R_4)\quad \frac{\forall X(\tau(\sigma) \rightsquigarrow X) \notin bb \quad \forall X(\sigma \rightsquigarrow X) \notin bb \quad \forall X(\rho(\rho(\sigma)) \rightsquigarrow X) \notin bb}{\langle \{(read(\sigma)\ \cdot\ SS) \diamond AC\} \cup \mathcal{R} \mid bb \cup \{\gamma(\rho(\sigma)) \rightsquigarrow Y\}\rangle \rightarrow \langle \{AC\} \cup \{Y\} \cup \mathcal{R} \mid bb\rangle}$$

$$(R_5)\quad \frac{\forall X(\tau(\sigma) \rightsquigarrow X) \notin bb \quad \forall X(\sigma \rightsquigarrow X) \notin bb \quad \forall X(\rho(\rho(\sigma)) \rightsquigarrow X) \notin bb \quad \forall X(\gamma(\rho(\sigma)) \rightsquigarrow X) \notin bb}{\langle \{(read(\sigma)\ \cdot\ SS) \diamond AC\} \cup \mathcal{R} \mid bb\rangle \rightarrow \langle \mathcal{R} \mid bb \cup \{\rho(\sigma) \rightsquigarrow SS \diamond AC\}\rangle}$$

Figure 3: Read primitives

$$(\mathrm{C})\quad \frac{\langle (s_1\ \cdot\ SS) \diamond ((s_2\ \cdot\ SS) \diamond AC) \mid bb\rangle \rightarrow \langle AC' \mid bb'\rangle}{\langle ((s_1\ +\ s_2)\ \cdot\ SS) \diamond AC \mid bb\rangle \rightarrow \langle AC' \mid bb'\rangle}$$

$$(\mathrm{S})\quad \frac{\langle (s_1\ \cdot\ (s_2\ \cdot\ SS)) \diamond AC \mid bb\rangle \rightarrow \langle AC' \mid bb'\rangle}{\langle ((s_1\ ;\ s_2)\ \cdot\ SS) \diamond AC \mid bb\rangle \rightarrow \langle AC' \mid bb'\rangle}$$

Figure 4: Sequential choice and composition

identical in both cases, there are two internal paths that can be followed: either generating $\gamma(token)$ that is to be consumed by $tell(token)$, or generating $\tau(token)$ that is to be consumed by $get(token)$. In order to remove this non-determinism, we can introduce an active blackboard, i.e., a blackboard that will autonomously combine the matching suspension terms and notify the corresponding operations. In this case *sendtoken* and *receivetoken* simply put their suspension terms on the blackboard. The blackboard then combines the two suspension terms, and unblocks the two blocked communication primitives. In this model, the two primitives are actually identical, except for the kind of suspension terms they generate. This allows us to rewrite the three basic μLog primitives in terms of just two lower level primitives: one to create a suspension term (called *put*), and a generalized *wait*-primitive that waits for a signal from the blackboard to continue. In practice the *wait* operation will wait for a suspension term to be combined with another suspension term on the blackboard.

Notice that *put* is quite different from *tell*. The *tell* primitive only accepts tokens as arguments because it is not allowed to create a side effect (suspension term) for which there is no corresponding process. This is different for the *put*-primitive. Suspension terms are now no longer considered side effects of a suspended coordination primitive, but data on an active blackboard. The successful combination of some suspension terms will create signals that are used to unblock *wait*-primitives. Since the suspension terms are no longer side-effects, they can now be created in their own right.

Using these two primitive operations, the synchronous μLog-operations can be re-expressed.

$$tell_s(t) = put(\tau(t)); wait(\tau(t))$$
$$get_s(t) = put(\gamma(t)); wait(\gamma(t))$$
$$read_s(t) = put(\rho(t)); wait(\rho(t))$$

Since this model makes a clear distinction between putting a suspension term on the blackboard, and waiting for the associated signal from the blackboard, is now possible to also express the asynchronous primitives from the basic μLog-model.

$$tell_a(t) = put(\tau(t))$$
$$get_a(t) = put(\gamma(t)); wait(\gamma(t))$$
$$read_a(t) = put(\rho(t)); wait(\rho(t))$$

as well as any other intermediate quasi-synchronous form where there can be some computation between the creation of the suspension term, and the synchronization point where the process waits for the result. These can be modeled as:

$$tell_q(t) = put(\tau(t)); P; wait(\tau(t))$$
$$get_q(t) = put(\gamma(t)); P; wait(\gamma(t))$$
$$read_q(t) = put(\rho(t)); P; wait(\rho(t))$$

In this example, P represents an arbitrary computation that can take place between the *put*-operation and the *wait*-operation. This allows the program to do some useful work during the reaction time of the active blackboard. This is a standard technique to allow for maximal overlap between communication and computation, as it is used in e.g. PVM or MPI. Since signals generated by the blackboard are persistent, they will never get lost, even in the case where the computation P would take longer than the active blackboard needs to react on the suspension terms. We will now look into the transition rules.

Formally, two sets of new rules need to be added. Rule (P) of Figure 5 states that the execution of a $put(\sigma)$ primitive consists of adding the suspension term σ on the blackboard bb. Rules (W_1), (W_2), and (W_3) of the same figure describe the behavior of the *wait* primitives. Essentially, they suspend until their suspension term reacts with a matching suspension term on the blackboard and creates a notification term. For *read*, *get*, and *tell* operations these notification terms correspond respectively to κ, ν and μ terms. They are produced by the rules of Figure 6. In Rule (C_1), a suspension term $\rho(\sigma)$ coming from a *read* operation is combined with a suspension term $\tau(\sigma)$ coming from a *tell* operation to create a notification term $\kappa(\sigma)$ which in turn allows the *wait* primitive associated with the *read* operation to resume. Note that the $\tau(\sigma)$ suspension term is not consumed. Rule (C_2) is similar for the *get* operation. However, here two new notification terms are created to resume the *wait* operations while the $\tau(\sigma)$ suspension term is consumed. This forces the communication between one and only one pair of *tell-get* primitives. The condition $\rho(\sigma) \notin bb$ additionally ensures that all the *read* operations concerned by the communication have been previously treated.

6. CONCLUSION AND RELATED WORK

In this paper we have investigated the effects of synchronous communication in the μLog coordination framework. We started by defining the concept of synchronous coordination and then introduced it into the μLog framework. We then formally presented the transition rules for the synchronous coordination. Two basic communication primitives (*put* and *wait*) have also been introduced. They have been proven expressive enough to translate the newly introduced synchronous version of the μLog primitives but also to code other forms including the existing asynchronous version of μLog primitives. This translation scheme relies on an active blackboard inspired by the chemical metaphor used in the CHAM machine [4] and in Gamma [3].

Related work includes the CCS language which also features the similarity between *tell* and *get* primitives. However, in contrast to it, our framework also incorporates a *read* primitive. Moreover, no equivalent is given for an explicit treatment of suspension terms. This is also a distinguishing feature of our work with those related to coordination languages. The CBS calculus [20], the Gamma calculus [3], and the Pi-calculus [17] presents similar features to our work but do not include mechanisms related to suspension terms and in general do not incorporate reification of suspensions.

Finally, a variety of *tell*, *get*, and *read*-operations are studied from the theoretical perspective of expressiveness in [5, 6, 18, 22]. However, no counterparts are given for primitives explicitly handling suspension records as we do in this paper.

$$(P) \qquad \langle \{(put(\sigma) \cdot SS) \diamond AC\} \cup \mathcal{R} \mid bb \rangle \rightarrow \langle \{SS \diamond AC\} \cup \mathcal{R} \mid bb \cup \{\sigma\} \rangle$$

$$(W_1) \qquad \langle \{(wait(\rho(\sigma)) \cdot SS) \diamond AC\} \cup \mathcal{R} \mid bb \cup \{\kappa(\sigma)\} \rangle \rightarrow \langle \{SS \diamond AC\} \cup \mathcal{R} \mid bb \rangle$$

$$(W_2) \qquad \langle \{(wait(\gamma(\sigma)) \cdot SS) \diamond AC\} \cup \mathcal{R} \mid bb \cup \{\nu(\sigma)\} \rangle \rightarrow \langle \{SS \diamond AC\} \cup \mathcal{R} \mid bb \rangle$$

$$(W_3) \qquad \langle \{(wait(\tau(\sigma)) \cdot SS) \diamond AC\} \cup \mathcal{R} \mid bb \cup \{\mu(\sigma)\} \rangle \rightarrow \langle \{SS \diamond AC\} \cup \mathcal{R} \mid bb \rangle$$

Figure 5: Put and wait primitives

$$(C_1) \qquad \langle \mathcal{R} \mid bb \cup \{\tau(\sigma)\} \cup \{\rho(\sigma)\} \rangle \rightarrow \langle \mathcal{R} \mid bb \cup \{\tau(\sigma)\} \cup \{\kappa(\sigma)\} \rangle$$

$$(C_2) \qquad \frac{\rho(\sigma) \notin bb}{\langle \mathcal{R} \mid bb \cup \{\tau(\sigma)\} \cup \{\gamma(\sigma)\} \rangle \rightarrow \langle \mathcal{R} \mid bb \cup \{\nu(\sigma), \mu(\sigma)\} \rangle}$$

Figure 6: Reaction rules

7. REFERENCES

[1] S. Ahuja, N. Carriero, and D. Gelernter. Linda and friends. *IEEE Computer*, 19(8):26–34, Aug. 1986.

[2] H. Bal, J. Steiner, and A. Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys*, 21(2):261–322, Sept. 1989.

[3] J.-P. Banatre and D. Le Metayer. Programming by Multiset Transformation. *Commun. ACM*, 36(1):98–111, Jan. 1993.

[4] G. Berry and G. Boudol. The Chemical Abstract Machine. *Theoretical Computer Science*, 96:217–248, 1992.

[5] N. Busi, R. Gorrieri, and G. Zavattaro. On the Turing Equivalence of Linda Coordination Primitives. *Electronic Notes in Theoretical Computer Science*, 7, 1997.

[6] N. Busi, R. Gorrieri, and G. Zavattaro. A Process Algebraic View of Linda Coordination Primitives. *Theoretical Computer Science*, 192(2):167–199, 1998.

[7] J. Cunha, P. Medeiros, M. Carvalhosa, and L. Pereira. Delta-Prolog: A distributed logic programming language and its implementation on distributed memory multiprocessors. In Kacsuk and Wise [15], pages 335–356.

[8] K. De Bosschere. Process-based Parallel Logic Programming Languages: A Survey of the Basic Issues. *The Journal of Systems and Software*, 39:71–82, 1997.

[9] K. De Bosschere and J.-M. Jacquet. Extending the μLog Framework with Local and Conditional Blackboard Operations. *The Journal of Symbolic Computation*, 21(4):669–697, Apr. 1996.

[10] K. De Bosschere and J.-M. Jacquet. μ^2log: Towards Remote Coordination. In P. Ciancarini and C. Hankin, editors, *Proceedings of Coordination'96*, volume 1061 of *Lecture Notes in Computer Science*, pages 142–159, Cesena/Italy, Apr. 1996. Springer-Verlag.

[11] K. De Bosschere and J.-M. Jacquet. Meta-coordination in the *μlog* coordination model. In H. Arabnia, editor, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Application*, volume 2, pages 761–767, Las Vegas, June 2000.

[12] E. Horita, J. de Bakker, and J. Rutten. Fully abstract denotational models for nonuniform concurrent languages. *Information and computation*, 115(1):125–178, 1994.

[13] J.-M. Jacquet and K. De Bosschere. On the Semantics of *μLog*. *Future Generation Computer Systems*, 10:93–135, Apr. 1994.

[14] J.-M. Jacquet and K. De Bosschere. Blackboard relations in the *μlog* coordination model. *New Generation Computing*, 2000. Accepted for publication.

[15] P. Kacsuk and M. Wise, editors. *Implementations of Distributed Prolog*. Series in Parallel Computing. Wiley, Chichester, 1992.

[16] R. Keller. A position on multiprocessing in prolog. In *Proceedings of the Joint Japanese/American Workshop ICOT/NSF*, pages 27–49, Oct. 1989.

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

[18] C. Palamidessi. Comparing the Expressive Power of the Synchronous and the Asynchronous Pi-Calculus. In *Proc. of the 24th ACM Symposium on Principles of Programming Languages (POPL)*, pages 256–265. ACM, 1997.

[19] G. Plotkin. A structured approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.

[20] K. V. S. Prasad. A Calculus of Broadcasting Systems. *Science of Computer Programming*, 25(2-3):285–327, 1995.

[21] M. Wise, D. Jones, and T. Hintz. PMS-Prolog: A Distributed Coarse-grain-parallel Prolog with Processes, Modules and Streams. In Kacsuk and Wise [15], pages 379–403.

[22] G. Zavattaro. Towards a Hierarchy of Negative Test Operators for Generative Communication. *Electronic Notes in Theoretical Computer Science*, 16(2):83–100, 1998.