State- and Event-based Reactive Programming in Shared Dataspaces

Nadia Busi[1], Antony Rowstron[2] and Gianluigi Zavattaro[1]

[1] 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

[2] Microsoft Research, 7 J J Thomson Avenue, Cambridge, CB3 0FB
E-mail: antr@microsoft.com

Abstract. The traditional Linda programming style, based on the introduction and consumption of data from a common repository, seems not adequate for highly dynamic applications in which it is important to observe modification of the environment which occur quickly. On the other hand, reactive programming seems more appropriate for this kind of applications. In this paper we consider some approaches recently presented in the literature for embedding reactive programming in shared dataspaces, we present a possible classification for these approaches, and we perform a rigorous investigation of their relative advantages and disadvantages.

1 Introduction

The development of Linda-like coordination languages [Gel85] for use over wide-area 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 insertion of tuples into particular tuples spaces, or dataspaces. Examples of Linda-like coordination languages including such primitives are JavaSpaces [W$^+$98], TSpaces [WMLF98] and WCL [Row98].

One interesting observation is that the primitives incorporated within the coordination languages, whilst all aiming to provide support for reactive programming, have different (informal) semantics. Indeed, it is possible to classify the primitives incorporated as being: (1) state-based, (2) event based, and (3) hybrid.

In this paper we provide a formal semantics to be used to compare the expressive power and the interchangeability of these reactive mechanisms, where by interchageability we mean the possibility to substitute one mechanism in place of the others without affecting the internal behaviour of the coordinated components.

For each of the classes we consider a typical coordination primitive, for clarity we assume there is a single dataspace:

(1) $forEach(a, P)$: spawns an instance of process P for each occurrence of a currently in the dataspace;

(2) $notify(a, P)$: produces a listener, that we denote with $on(a, P)$, which will spawn an instance of P each time a new a is produced;

(3) $monitor(a, P)$: spawns an instance of process P for each occurrence of a currently in the dataspace, and produces a listener $on(a, P)$ which will spawn an instance of P each time a new a is produced.

Furthermore, for event-based mechanisms we consider a $dereg(a, P)$ primitive which removes one occurrence of listener $on(a, P)$.

The primitive $notify$ has been proposed as a coordination operations by JavaSpaces [W$^+$98], $monitor$ has been introduced for the first time by one of the authors in WCL [Row98], while $forEach$ is an adaptation to our context of the so-called `copy-collect` operation [RW98]: this primitive, proposed as a solution for the typical multiple-read problem of Linda, has the ability to atomically copy all the data satisfying a certain pattern from one dataspace to another.

In this paper we demonstrate that the three approaches are equivalent by showing how each of the approaches can be reproduced in terms of the others. However, only some of these translations are adequate for open environments (new components may be introduced in the system at run time and therefore dynamically); in particular, the mapping from the state- to the event-based approach and vice versa are valid only for closed applications (where the components involved in the system are a priori known and therefore statically).

In Section 2 we further motivate the introduction of reactive primitives. In Section 3 we present a process calculus used to perform a rigorous investigation of the three reactive primitives; the results of this comparison are reported in Section 4. Finally, Section 5 contains some conclusive remarks.

2 Reactive Programming in Shared Dataspaces

In order to demonstrate why reactive programming primitives are attractive in Linda-like coordination languages, let us consider some of the functionality of a simple instant messenger type tool: the functionalities that allows a user to display a list of "buddies" and their current status (e.g., in meeting, at lunch, busy and so forth).

Each user runs a *buddy list agent* which enables them to signal to other users what their current status is, and to observe the status of the other users. A user can change their state by pressing buttons on the buddy list agent. Although this example is simple, it demonstrates the power of reactive programming.

A simple way to create the buddy list agent is to create a shared dataspace into which the buddy list agents place tuples representing their status. Whenever a user starts a buddy list agent, it inserts a status tuple into the shared dataspace, containing the user's name and their initial status. When a user changes their status in the buddy list agent, the agent removes their status tuple and inserts a new tuple with the user's name and their new status. When the agent starts, a list of all users with a status tuple is displayed, also showing their current status. As other users change their states, these changes are reflected on all the users buddy lists.

The state tuples in the shared dataspace at any one time represent the *global status* of (most of) the users and a buddy list agent can examine these tuples to determine who is currently available and their status. This means the application does not involve a centralised coordinator. It should be noted that the tuple space does not necessarily contain the entire global state because tuples are removed to be updated. Therefore, if a buddy list agent is updating their user's status tuple, the user will not be represented by a tuple in the tuple space.

The implementation of such a scheme using the standard Linda primitives is not easy, requiring shared counters and so forth. However, reactive programming primitives should be able to enable this programming style.

So, let us consider how the monitor primitive can be used in this example. When a buddy list agent is started it performs a *monitor* on the shared tuple space. This has the effect of returning all the current state tuples and any ones that are inserted in future. In the description of the buddy list agent it was described how it is possible for the shared dataspace not to contain all state tuples as one or more of the agents may be updating their status tuple. However, this does not matter because any agent updating their status tuple will reinsert the status tuple. When this occurs the *monitor* primitive will consider the inserted tuple and return it. When a buddy list agent receives the tuple it is able to check the users name locally and discover whether the tuple represents the status of as yet unseen user or if it is an update of an existing user's status.

The monitor primitive was introduced in WCL [Row98]. It explicitly returns the tuples that match the template in the dataspace, as well as tuples subsequently inserted (until a *dereg* is performed). JavaSpaces, in contrast, provides a *notify* primitive which *does not* return tuples that are already in the tuple space which match the template, simply tuples inserted subsequently[1]. This has several side effects, one is that the *notify* primitive can not be used in the same way as the *monitor* primitive in the buddy list example. From a programmer's perspective, the issue is how to ensure that each status tuple is read once. If all matching tuples in the dataspace are retrieved prior issuing the *notify* primitive (for example using a *forEach* primitive), a tuple can be inserted once the *forEach* has completed but before the *notify* has been started, and therefore, be missed. Alternatively, if the *notify* is issued first, and after a *forEach* is performed, a single tuple can be returned twice (and there is no explicit ordering). Therefore, from a programmers perspective the use of a monitor appears more powerful and flexible.

---

[1] It actually returns a notification that a tuple has been inserted not the tuple, and the process must explicitly retrieve the new tuple.

3 The Calculus

In this section, we introduce a process calculus based on the Linda coordination primitives plus the reactive mechanisms discussed in the Introduction,

By borrowing typical techniques from the tradition of process calculi for concurrency (e.g., Milner's CCS [Mil89]), an agent is described as a term of an algebra where the basic actions are typical Linda coordination primitives or one of the considered reactive based coordination operations.

To be general, we consider a denumerable set of names for data, called $Data$, ranged over by a, b, ... The set $Prog$ of programs, ranged over by P, P', ..., is the set of terms generated by the following grammar:

$$
\begin{array}{lll}
P & ::= & \mathbf{0} \mid \mu.P \mid \langle a \rangle \mid on(a,P) \mid K \mid P|P \\
\mu & ::= & out(a) \mid in(a) \mid rd(a) \mid \\
 & & forEach(a,P) \mid notify(a,P) \mid monitor(a,P) \mid dereg(a,P)
\end{array}
$$

where μ denotes an instance of one of the possible coordination primitives, and K stands for a generic element of a set $Name$ of program names; we assume that all program name occurrences are equipped with a corresponding (guarded) defining equation of the form $K = P$. Program names are used to support recursive definitions as, for example, in the term $Ren_{ab} = in(a).out(b).Ren_{ab}$, which represents a program able to repeatedly rename messages of the kind a in messages of the kind b.

A term P is the parallel composition (we use the standard parallel composition operator $|$) of the active programs, plus the data which are currently available in the data repository, and terms which denote listeners used for the modeling of event-based reactive programming. Term $\mathbf{0}$ represents a program that can do nothing. Term $\mu.P$ is a program that can do the action μ and after behaves like P. The term $\langle a \rangle$ denotes an instance of datum a which is currently available for $rd(a)$ and $in(a)$ operations; on the other hand, $on(a,P)$ represents a listener responsible to activate a new instance of program P each time a new occurrence of datum a is produced.

In the following we will exploit a structural congruence relation in order to equate terms which represents the same system even if they are syntactically different. Let $equiv$ be the least congruence relation satisfying:

$$
\begin{array}{ll}
P|Q \equiv Q|P & P|(Q|R) \equiv (P|Q)|R \\
P|\mathbf{0} \equiv P & P \equiv K \quad \text{if } P = K
\end{array}
$$

In the following we will reason upto structural congruence, i.e., we will not make any distinction between P and Q whenever $P \equiv Q$.

We use the following notation: $P \notin R$ (to indicate that P is not a subterm of program R), $\prod_n P$ (to denote the parallel composition of n instances of program P), and $\prod_i P_i$ (to denote the parallel composition of the indexed programs P_i).

The operational semantics is defined by the transition relation $(Prog, \longrightarrow)$ defined as the least relation satisfying the axioms and rules reported in Table 1.

$in(a).P \vert \langle a\rangle \vert R \longrightarrow P \vert R$	
$rd(a).P \vert \langle a\rangle \vert R \longrightarrow P \vert \langle a\rangle \vert R$	
$forEach(a,P).Q \vert \prod_n \langle a\rangle \vert R \longrightarrow Q \vert \prod_n P \vert \prod_n \langle a\rangle \vert R$	$\langle a\rangle \notin R$
$notify(a,P).Q \vert R \longrightarrow Q \vert on(a,P) \vert R$	
$monitor(a,P).Q \vert \prod_n \langle a\rangle \vert R \longrightarrow Q \vert \prod_n P \vert on(a,P) \vert \prod_n \langle a\rangle \vert R$	$\langle a\rangle \notin R$
$dereg(a,P).Q \vert on(a,P) \vert R \longrightarrow Q \vert R$	
$out(a).P \vert \prod_i on(a,P_i) \vert R \longrightarrow P \vert \langle a\rangle \vert \prod_i P_i \vert \prod_i on(a,P_i) \vert R$	for any S, $on(a,S) \notin R$

Table 1. The operational semantics.

In the following we denote by $P \longrightarrow^* P'$ the fact that either $P \equiv P'$ or there exist $P_0 \dots P_n$ such that $P_0 \equiv P$, $P_n \equiv P'$, and $P_i \longrightarrow P_{i+1}$ (for $0 \leq i < n$).

The first two axioms deal with the $in(a)$ and $rd(a)$ coordination operations: both operations require the presence of a term $\langle a\rangle$; in the second case the result of the execution of the operation is that this term is consumed.

The third axioms describe the $forEach(a,P)$ operation: the result of its execution is the spawning of a new process P for each instance of $\langle a\rangle$ (observe that this is ensured by the side condition $\langle a\rangle \notin R$). The result of the execution of the $notify(a,P)$ operation is the spawning of the listener $on(a,P)$. The $monitor(a,P)$ primitive combines the two above operations: a new process P is spawned for each instance of $\langle a\rangle$ and a new listener $on(a,P)$ is produced.

The $dereg(a,P)$ requires the presence of a listener $on(a,P)$, and this term is removed as effect of the execution of this operation.

The $out(a)$ operation produces a new term $\langle a\rangle$; moreover, for each listener $on(a,P)$ in the environment, a new program P is spawned (observe that this is ensured by the side condition: for any S, $on(a,S) \notin R$).

In the following we will focus on three variants of the calculus, in which only one among the three reactive primitives $forEach$, $notify$, and $monitor$ is considered. The three calculi are denoted with $L[forEach]$, $L[notify]$, and $L[monitor]$, respectively. We will also consider a fourth subcalculus in which both the $notify$ and the $forEach$ operations are considered: this calculus is denoted by $L[forEach, notify]$.

4 Comparing the Reactive Mechanisms

In this section we compare the expressive power of the different reactive mechanisms by investigating the encodability of one mechanism in terms of the others. We will show that in general, for each pair of calculi there exists an encoding function from the first calculus to the second.

Two kinds of encodings are used: one adequate for closed systems only, and one suitable for open systems too. In the first case, indeed, it is necessary to assume that all the programs involved in the system are a priori known; on the other hand, the second class of encodings does not make this kind of assumption.

To be more precise, we state that an encoding function $[\![ ]\!]$ from one calculus to another is *open* if the following costraints are satisfied:

$$\begin{aligned} [\![P]\!] &= [P] | \prod_{n(P)} R_a \\ [\![P|Q]\!] &= [\![P]\!] | [\![Q]\!] \end{aligned}$$

where $n(P)$ denotes the set of names of data which occur in the program P, and R_a denotes a program (depending on the considered encoding) used to manage the name a occurring inside P.

We refer to this class of encodings as "open" because the addition of a new program Q in parallel with P does not require to recompute the overall encoding of P; indeed, given P, its encoding $[\![P]\!]$ and a program Q to be added in parallel with P, we have that the new encoding

$$[\![P|Q]\!] = [P] | [Q] | \prod_{n(P) \cup n(Q)} R_a = [\![P]\!] | [\![Q]\!] | \prod_{n(Q) \setminus n(P)} R_a$$

can be obtained simply by adding new programs in parallel with the initial encoding $[\![P]\!]$

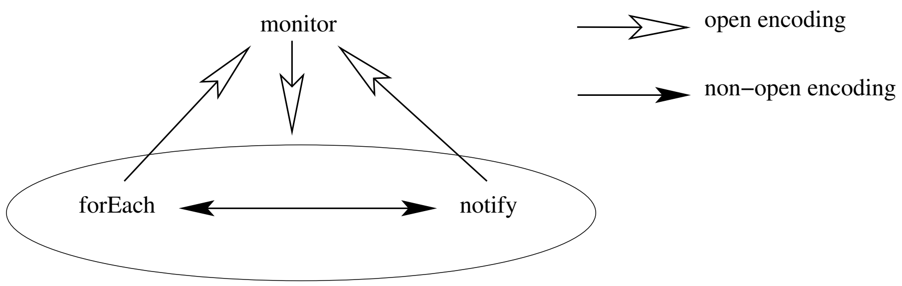

Fig. 1. Summary of the encodings.

The results presented in the rest of this section are summarized in Figure 1. In Subsection 4.1 (resp. 4.2) we show the existence of an open encoding of L[notify] (resp. L[forEach]) in L[monitor]. This means that the *monitor* primitive is expressive enough to model both the *notify* and the *forEach* operations. We show the existence of an open encoding of L[monitor] in L[forEach,notify] in Subsection 4.3.

As far as static systems are concerned, also the $notify$ and the $forEach$ primitives are interchangeable: we show the existence of non–open encodings of L[notify] in L[forEach] (and vice versa) in Subsection 4.4 (resp. 4.5). By composing each of these encodings with the open encoding of the *monitor* primitive in the language containing both the $notify$ and the $forEach$ operations, we obtain a non–open encoding of L[monitor] in L[notify] (and in L[forEach]).

4.1 Encoding L[notify] in L[monitor]

In this section we show that it is possible to model the event-based reactive mechanism of the *notify* primitive using the *monitor* operation.

The hybrid approach of the *monitor* primitive observes all the data already available at the instant in which the operation is performed, as also the future incoming entries. On the other hand, the *notify* operation observes only the incoming entries. In order to overcome this difference, for each datum $\langle a \rangle$ we exploit an auxiliary datum $\langle a' \rangle$; this kind of data are produced and subsequently removed every time an $out(a)$ operation is performed. In this way the auxiliary data $\langle a' \rangle$ are not persistent in the dataspace, but they are stored only temporarily.

When we need to model a $notify(a, P)$ operation, we use $monitor(a', P)$ which observes the auxiliary data only; as these data are not persistent, only subsequent productions will be observed.

Formally, the encoding function is defined as $\llbracket P \rrbracket = [P]$ where $[P]$ is inductively defined as follows:

$$
\begin{array}{lll}
[\mathbf{0}] = \mathbf{0} & [\langle a \rangle] = \langle a \rangle & \\
[on(a, P)] = monitor(a', [P]) & [K] = K' & \\
[P|Q] = [P]|[Q] & [\mu.P] = \mu.[P] & \mu \neq notify(a, Q), out(a) \\
[notify(a, P).Q] = monitor(a', [P]).[Q] & & \\
[out(a).P] = out(a').in(a').out(a).[P] & &
\end{array}
$$

where, for each program name K in $L[notify]$ with definition $K = P$, we assume the existence of a corresponding K' in $L[monitor]$ with definition $K' = [P]$. Moreover, we assume that for each encoding $\llbracket P \rrbracket$ the auxiliary names a' are different from each of the names of data occurring in P.

This encoding satisfies the above contraints; thus it is open. Moreover, we have that it is also homomorphic with respect to the parallel operation, i.e., $\llbracket P|Q \rrbracket = \llbracket P \rrbracket | \llbracket Q \rrbracket$. In the terminology of [dBP91] this property is called modularity with respect to the parallel composition operator.

The correctness of this encoding is formally stated by the following theorem which states that, given a program P of $L[notify]$, each computation step of P can be simulated by $\llbracket P \rrbracket$, and that each computation of $\llbracket P \rrbracket$ can be extended in such a way that it corresponds to an equivalent computation of P. Due to space limit we do not report the proof of this theorem (as also the proofs of the theorems in the following sections).

Theorem 1. *Given a program P of $L[notify]$ we have that:*

- *if $P \longrightarrow P'$ then $\llbracket P \rrbracket \longrightarrow^+ \llbracket P' \rrbracket$;*
- *if $\llbracket P \rrbracket \longrightarrow^+ Q$ then there exists P' such that $Q \longrightarrow^* \llbracket P' \rrbracket$ and $P \longrightarrow^+ P'$.*

An interesting property of this encoding concerns the use of the auxiliary names a'. As stated above, data $\langle a' \rangle$ are produced (and subsequently removed) simply to notify the execution of $out(a)$ operations. Observe that this production and subsequent consumption operations could be executed in interleaving with other operations performed by concurrent processes. As an example consider the encoding $[\![notify(a,P)|out(a)]\!] = monitor(a', [\![P]\!])|out(a').in(a').out(a)$. Consider now the computation of the encoding in which first $\langle a' \rangle$ is produced and consumed, after the *monitor* operation is performed, and finally, the $out(a)$ primitive is executed.

This computation is particularly of interest because no reaction is activated even if the output of $\langle a \rangle$ is executed after the execution of the program representing the $notify(a, P)$ process. However, this is not a problem for the encoding because this particular computation corresponds to the computation of the initial program in which the $notify$ operation is executed only after the output of $\langle a \rangle$.

4.2 Encoding L[forEach] in L[monitor]

Now we concentrate on the modeling of the state-based primitive $forEach$ using the hybrid approach adopted by *monitor*. The difference between the two operations is that *monitor* observes not only the data already available, but is also activates a listener which observes the future incoming entries. This difference can be covered simply by removing this listener immediately after its activation: following this approach, a $forEach$ operation is modeled by a *monitor* primitive followed by a *dereg*. Formally, the new encoding can be defined as follows

$$[\![\![P]\!]\!] = [\![P]\!] | \prod_{a \in n(P)} \langle lock_a \rangle$$

where $[\![P]\!]$ is inductively defined as above, with only two non-trivial cases

$$\begin{aligned} [\![forEach(a,P).Q]\!] &= in(lock_a).monitor(a, [\![P]\!]). \\ &\quad\; dereg(a, [\![P]\!]).out(lock_a).[\![Q]\!] \\ [\![out(a).P]\!] &= in(lock_a).out(a).out(lock_a).[\![P]\!] \end{aligned}$$

where we assume that for each encoding $[\![\![P]\!]\!]$ the auxiliary names $lock_a$ are all distinct from the names a occurring inside P.

Also in this case the correctness of the encoding is stated by a theorem similar to Theorem 1; the difference here is in the fact that we have to consider also the data $\langle lock_a \rangle$.

Theorem 2. *Given a program P of $L[forEach]$ we have that:*

- *if $P \longrightarrow P'$ then $[\![\![P]\!]\!] \longrightarrow^+ [\![\![P']\!]\!] | \prod_{a \in n(P) \setminus n(P')} \langle lock_a \rangle$;*
- *if $[\![\![P]\!]\!] \longrightarrow^+ Q$ then there exists P' such that $P \longrightarrow^+ P'$ and $Q \longrightarrow^* [\![\![P']\!]\!] | \prod_{a \in n(P) \setminus n(P')} \langle lock_a \rangle$.*

The encoding exploits, for each name of datum a occurring in the source program P, the auxiliar datum $\langle lock_a \rangle$, to implement mutual exclusion between the execution of the programs corresponding to the output operation $out(a)$ and the reactive operations $forEach(a)$. Mutual exclusion is achieved simply by forcing the withdrawal (and subsequent release) of the datum $\langle lock_a \rangle$ before (and after) each sequence of critical operations to be executed in mutual exclusion.

This locking policy is necessary in order to ensure that the listener produced by the execution of a $monitor(a)$ operation is deregistered before a subsequent output operation $out(a)$ is performed (e.g., by some other concurrent process).

Consider, as an example, the encoding of $\langle a \rangle | forEach(a, out(a))$ if we do not use mutual exclusion. In this case the target program becomes

$$\langle a \rangle | monitor(a, out(a)).dereg(a, out(a))$$

This program could activate an infinite computation in the case the *dereg* operation is delayed indefinitely: this could happen if a loop is activated in which first the reaction $out(a)$ is executed, and after the listener $on(a, out(a))$ reacts by spawning a new instance of $out(a)$. On the other hand, the source program $\langle a \rangle | forEach(a, out(a))$ has no infinite computation.

Observe that the locking policy involves only operations on the same name; the concurrent execution of operations modeling an $out(a)$ and a $forEach(b, P)$ primitive, for example, is allowed because the two operations consider the two distinct data $\langle lock_a \rangle$ and $\langle lock_b \rangle$, respectively. Finally, observe that the encoding is open even if it is not modular.

4.3 Encoding of L[monitor] in L[forEach,notify]

In this section we investigate the possibility to model the hybrid approach exploiting both the state- and the event-based approaches. Intuitively, this can be done simply by modeling the *monitor* operation with a $forEach$ immediately followed by a $notify$ operation.

Following this approach, we react to the data currently present in the repository as also to those data which will be introduced subsequently. The unique problem that may happen, occurs if new interesting data are produced between the execution of the $forEach$ and the $notify$ operations; in this case, the produced instance of the datum does not activate the expected reaction. To avoid this problem we could exploit a locking policy similar to the one adopted in the previous subsection.

Formally, we define the new encoding as

$$[\![P]\!] \;=\; [\![P]\!] | \prod_{a \in n(P)} \langle lock_a \rangle$$

where $[\![P]\!]$ is inductively defined as above, with only three significant cases

$$\begin{array}{ll} [\![monitor(a, P).Q]\!] & = in(lock_a).forEach(a, [\![P]\!]). \\ & \quad notify(a, [\![P]\!]).out(lock_a).[\![Q]\!] \\ [\![on(a, P)]\!] & = on(a, [\![P]\!]) \\ [\![out(a).P]\!] & = in(lock_a).out(a).out(lock_a).[\![P]\!] \end{array}$$

where we assume that for each encoding $[\![P]\!]$ the auxiliary names $lock_a$ are all distinct from the names a occurring inside P.

The correctness of this encoding is a consequence of a theorem corresponding to Theorem 2 where the language $L[monitor]$ is considered instead of $L[forEach]$. This encoding exploits a locking policy which avoid the concurrent execution of operations representing *monitor* and *out* operations executed on the same name a: these operations must be executed in mutual exclusion in order to avoid that some events are not observed (then some reactions could be lost).

As an example of undesired computation consider $out(a)|monitor(a, LOOP)$, where $LOOP$ is any program which performs an infinite computation. This program has only infinite computations as it is ensured that the reaction $LOOP$ is activated, both in the case that $out(a)$ is executed before *monitor* and in the case it is executed after. Consider now the encoding of this program in the case the locking policy is not adopted:

$$out(a)|forEach(a, [\![LOOP]\!]).notify(a, [\![LOOP]\!])$$

This second program has at least one finite computation; indeed consider the case in which $out(a)$ is scheduled exactly between the execution of the $forEach$ and the $notify$ operations.

One could think to solve this problem simply by changing the order of the two reactive operations obtaining the new encoding:

$$out(a)|notify(a, [\![LOOP]\!]).forEach(a, [\![LOOP]\!])$$

This new program has only infinite computations; however, it could activate the undesired computation in which two reactions are activated in the case the $out(a)$ operation is executed in interleaving with the two reactive primitives.

Also in this case, the locking policy involves only concurrent operations performed on the same name. Similarly to the previous subsection, the encoding is open even if not modular.

4.4 Encoding L[notify] in L[forEach]

In the previous subsections we have formally proved the intuitive result that the hybrid paradigm is powerful enough to model both the event- and the state-based reactive approaches; moreover, we showed that the $notify$ and the $forEach$ primitives permit to emulate the hybrid *monitor* operation (at the price of introducing some locking mechanism). It is also interesting to observe that all the encodings that we have presented are suitable for open applications.

In this section we start the investigation of the modeling of the event-based approach using the state-based one. The interesting result is that even if an encoding exists, it is not suitable for open applications; namely, it does not satisfy the constraints we have fixed for open encodings. The problem is that the encoding that we present requires the a priori knowledge of all the possible programs that will be executed in the system. This is against the basic requirements of open applications in which we usually assume that there exist components of the system which are added at run-time.

The encoding is based on the idea that listeners can be represented by auxiliary data; namely, for each possible listener $on(a, P_{a_i})$ we use an auxiliary datum $\langle a_i \rangle$ which is introduced in the dataspace. Whenever an output operation $out(a)$ is performed, the presence of these auxiliary data $\langle a_i \rangle$ is checked, and for each of them the corresponding reaction is activated; this operation can be obtained simply by executing a sequence of operations $forEach(a_i, P_{a_i})$ for all possible reactions P_{a_i}. The drawback of this approach is that it is necessary to know a priori all the possible reactions P_{a_i} which could be involved.

Formally, let P be a program of $L[notify]$ to be encoded in $L[forEach]$; for each name a occurring in P, i.e., $a \in n(P)$, we denote with $ON_P(a)$ the programs $P_{a_1}, \ldots, P_{a_l}$ which could be the possible reactions associated to a in P, i.e., all those programs P_a appearing in operations $notify(a, P_a)$ or terms $on(a, P_a)$. For each of the programs $P_{a_i} \in ON_P(a)$, we consider an auxiliary name a_i and a program name K_{a_i}. With ON_P we denote the function which associates to each $a \in n(P)$ the programs in $ON_P(a)$.

The encoding is defined as follows

$$[\![P]\!] \;=\; [\![P]\!]_{ON_P} | \prod_{a \in n(P)} \langle lock_a \rangle$$

where $[\![P]\!]_{ON_P}$ is inductively defined with only three non-trivial cases

$$\begin{array}{ll}
[\![notify(a, P_{a_i}).Q]\!]_{ON_P} & = in(lock_a).out(a_i).out(lock_a).[\![Q]\!]_{ON_P} \\
[\![on(a, P_{a_i})]\!]_{ON_P} & = \langle a_i \rangle \\
[\![out(a).P]\!]_{ON_P} & = in(lock_a).forEach(a_1, K_{a_1}).forEach(a_2, K_{a_2}) \ldots \\
 & \quad forEach(a, K_{a_l}).out(a).out(lock_a).[\![P]\!]_{ON_P} \\
 & \qquad \qquad \text{if } ON_P(a) = P_{a_1} \ldots P_{a_l}
\end{array}$$

where we assume that for each encoding $[\![P]\!]$ the auxiliary names $lock_a$ are all distinct from the names a occurring inside P, and that the program names K_{a_i} are all distinct from the other program names K occurring in P. For each of this program name K_{a_i}, with $P_{a_i} \in ON_P(a)$, we consider the following definition $K_{a_i} = [\![P_{a_i}]\!]_{ON_P}$.

The program names K_{a_i} are used to model the corresponding reactions P_{a_i}. This appraoch is necessary, e.g., to model programs of $L[notify]$, see for example $notify(a, out(a)).out(a)$, which have an infinite behaviour even if they are not recursively defined. This cannot happen in $L[forEach]$ where only recursively defined programs could give rise to infinite computations. As an example, consider the following program corresponding to $[\![notify(a, out(a)).out(a)]\!]$ which exploit a recursive definition for the program name K_{a_1}:

$$\begin{array}{l}
in(lock_a).out(a_1).out(lock_a).in(lock_a).forEach(a_1, K_{a_1}).out(lock_a) \\
K_{a_1} \;=\; in(lock_a).forEach(a_1, K_{a_1}).out(lock_a)
\end{array}$$

It is worth noting that this encoding does not satisfy the constraints we have fixed for open encodings; this because the inner encoding function $[\![\,]\!]$ depends on the initial term considered by the outer encoding $[\![\,]\!]$. For example, encoding P

in parallel with Q is usually different from encoding P in parallel with a different program R.

In this case, the theorem proving the correctness of the encoding should be rephrased in order to manage the new kind of non-open encoding.

Theorem 3. *Given a program P of $L[notify]$ we have that:*

- *if $P \longrightarrow P'$ then $[\![P]\!] \longrightarrow^+ [\![P']\!]_{ON_P} | \prod_{a \in n(P)} \langle lock_a \rangle$;*
- *if $[\![P]\!] \longrightarrow^+ Q$ then there exists P' such that $P \longrightarrow^+ P'$ and $Q \longrightarrow^* [\![P']\!]_{ON_P} | \prod_{a \in n(P)} \langle lock_a \rangle$.*

Also this encoding adopts mutual exclusion among the execution of operations performed on the same name. In order to undertand the importance of this locking policy consider the program $notify(a, notify(a, LOOP)).out(a)$ in which it is ensured that only one reaction can be activated (i.e., $LOOP$ cannot be activated). On the other hand, if we consider its encoding without the locking policy we obtain

$$
\begin{array}{l}
out(a_1).forEach(a_1, K_{a_1}).forEach(a_2, K_{a_2}).out(a) \\
K_{a_1} \;=\; out(a_2) \\
K_{a_2} \;=\; LOOP'
\end{array}
$$

where $LOOP'$ is the encoding of $LOOP$. This program could give rise to an undesired infinite computation in the case the first reaction K_{a_1} is executed in interleaving with the two $forEach$ operations.

4.5 Encoding L[forEach] in L[notify]

In this section we consider the problem of encoding the state-based reactive programming approach in the event-based one. Also in this case we show that the encoding exists, but it is not suitable for open applications.

The idea on which the encoding is based is to associate to each datum $\langle a \rangle$ a group of listeners $on(a_i, P_{a_i})$, one for each possible reaction P_{a_i}. In this context, if we want to model the execution of a $forEach(a, P_i)$ operation it is sufficient to produce a datum $\langle a_i \rangle$: as reaction to the production of this datum a number of reactions P_i, corresponding to the number of occurrences of the listener $on(a_i, P_{a_i})$, corresponding to the number of occurrences of $\langle a \rangle$, are activated.

Formally, let P be a program of $L[forEach]$ that we want to encode in $L[notify]$; for each name a occurring in P, i.e., $a \in n(P)$, we denote with $RE_P(a)$ the programs $P_{a_1}, \ldots, P_{a_l}$ which could be the possible reactions associated to a in P, i.e., all those programs P_a appearing in operations $forEach(a, P_a)$. For each of the programs $P_{a_i} \in RE_P(a)$, we consider an auxiliary name a_i and a program name K_{a_i}. With RE_P we denote the function which associates to each $a \in n(P)$ the programs in $RE_P(a)$.

The encoding is defined as follows

$$
[\![P]\!] \;=\; [\![P]\!]_{RE_P} | \prod_{a \in n(P)} \langle lock_a \rangle
$$

where $[\![P]\!]_{RE_P}$ is inductively defined with only the following non-trivial cases

$$\begin{array}{ll}
[\![\langle a\rangle]\!]_{RE_P} & = \langle a\rangle|on(a_1,K_{a_1})|on(a_2,K_{a_2})|\ldots|on(a_l,K_{a_l}) \\
& \qquad\qquad\qquad\qquad \text{if } RE_P(a) = P_{a_1}\ldots P_{a_l} \\
[\![out(a).P]\!]_{RE_P} & = in(lock_a).notify(a_1,K_{a_1}).notify(a_2,K_{a_2})\ldots \\
& \quad notify(a_l,K_{a_l}).out(lock_a).out(a).[\![P]\!]_{RE_P} \\
& \qquad\qquad\qquad\qquad \text{if } RE_P(a) = P_{a_1}\ldots P_{a_l} \\
[\![forEach(a,P_{a_i}).Q]\!]_{RE_P} & = in(lock_a).out(a_i).out(lock_a).[\![Q]\!]_{RE_P} \\
[\![in(a).P]\!]_{RE_P} & = in(a).in(lock_a).dereg(a_1,K_{a_1}).dereg(a_2,K_{a_2})\ldots \\
& \quad dereg(a_l,K_{a_l}).out(lock_a).[\![P]\!]_{RE_P} \\
& \qquad\qquad\qquad\qquad \text{if } RE_P(a) = P_{a_1}\ldots P_{a_l}
\end{array}$$

where we assume that for each encoding $[\![P]\!]$ the auxiliary names $lock_a$ are all distinct from the names a occurring inside P, and that the program names K_{a_i} are all distinct from the other program names K occurring in P. For each of these program names K_{a_i}, with $P_{a_i} \in RE_P(a)$, we consider the following definition $K_{a_i} = [\![P_{a_i}]\!]_{RE_P}$. For the same reasons discussed in the previous subsection, also this encoding is not open.

The theorem proving the correctness of the encoding should be rephrased as follows.

Theorem 4. *Given a program P of $L[forEach]$ we have that:*

- *if $P \longrightarrow P'$ then $[\![P]\!] \longrightarrow^+ [\![P']\!]_{RE_P}|\prod_{a\in n(P)}\langle lock_a\rangle$;*
- *if $[\![P]\!] \longrightarrow^+ Q$ then there exists P' such that $P \longrightarrow^+ P'$ and $Q \longrightarrow^* [\![P']\!]_{RE_P}|\prod_{a\in n(P)}\langle lock_a\rangle$.*

Also this encoding adopts mutual exclusion among the execution of operations performed on the same name. In order to understand the importance of this locking policy consider the program $out(a)|forEach(a, forEach(a, LOOP))$; observe that if this program activates the first reaction, then also the second one will be executed (in this case the program has an infinite computation).

Consider now the corresponding encoding in the case we do not exploit the locking policy. There are two possible reactions associated to the datum $\langle a\rangle$ that we denote with $P_{a_1} = forEach(a, LOOP)$ and $P_{a_2} = LOOP$. The encoding is

$$\begin{array}{l}
notify(a_1,K_{a_1}).notify(a_2,K_{a_2}).out(a)|out(a_1) \\
K_{a_1} \;=\; out(a_2) \\
K_{a_2} \;=\; LOOP'
\end{array}$$

where $LOOP'$ is the encoding of $LOOP$. This program could give rise to an undesired computation in which only the first reaction is activated; consider the computation in which the first $notify$ is executed, after the datum $\langle a_1\rangle$ is produced, the reaction K_{a_1} is activated, and finally $\langle a_2\rangle$ is produced without producing any reaction (because the second $notify$ operation has not been executed yet). In this case even if the first reaction is activated the overall computation in finite.

5 Conclusion

In this paper we have investigated three possible primitives for reactive programming to be embedded to Linda-like languages: $forEach$ (reactions depend on the current state of the repository), $notify$ (reactions depends on the future output operations), and *monitor* (which combines both the kind of reactions).

We have showed that the three approaches are interchangeable: namely, we have presented a possible way to translate any application developed following an approach, in an equivalent one based on a different kind of reactive mechanism. The interesting fact is that some of the translations are not adequate for open applications, this because they require to know a priori all the possible programs involved in the system. The lesson we have learned is that the *monitor* operation appears as the more powerful because it permits to model the other two primitives in a more flexible way.

Putting together the results proved in this paper and in a previous paper [BZ00] investigating the $notify$ primitive only, we obtain the interesting result that there exists a significant gap of expressiveness between a reactive Linda (Linda extended with at least one of the three reactive primitives) and the basic Linda (with only input, output, and read operations). Indeed, in [BZ00] two of the authors proved that a process calculus with only *in* and *out* operations is not Turing-powerful, while it becomes (weakly) Turing-powerful in the case the $notify$ operation is added to the calculus. In this paper we showed that $notify$ can be modeled also with *monitor* and $forEach$, thus the same expressiveness result holds also for these reactive primitives.

References

[BZ00] 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.

[dBP91] F.S. de Boer and C. Palamidessi. Embedding as a Tool for Language Comparison: On the CSP Hierarchy. In *Proc. of CONCUR'91*, volume 527, pages 127–141. Springer-Verlag, Berlin, 1991.

[Gel85] D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

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

[Row98] A. Rowstron. WCL: A web co-ordination language. *World Wide Web Journal*, 1(3):167–179, 1998.

[RW98] A. Rowstron and A. Wood. Solving the Linda multiple `rd` problem using the `copy-collect` primitive. *Science of Computer Programming*, 31(2-3):335–358, 1998.

[W+98] J. Waldo et al. Javaspace specification - 1.0. Technical report, Sun Microsystems, March 1998.

[WMLF98] P. Wyckoff, S. McLaughry, T. Lehman, and D. Ford. T spaces. *IBM Systems Journal*, 37(3):454–474, 1998.