Publish/Subscribe vs. Shared Dataspace Coordination Infrastructures

Is it just a matter of taste?

Nadia Busi
Department of Computer Science,
University of Bologna, Italy.
E-mail: `busi@cs.unibo.it`

Gianluigi Zavattaro
Department of Computer Science,
University of Bologna, Italy.
E-mail: `zavattar@cs.unibo.it`

Abstract

Two main architectural styles emerged in the panorama of inter-agent coordination infrastructures: the publish/subscribe model, according to which agents communicate via the raising and catch of events, and the shared-dataspace – Linda-like – approach, in which communication is achieved by introducing and consuming objects from a common repository. In this paper we perform a rigorous comparison of the two approaches with a particular emphasis on their interchangeability; we obtain the following results: (i) the shared dataspace model can be reduced to the publish/subscribe architecture, while (ii) the vice versa holds only if a global coordination operation, such as copy-collect *or* inp*, is provided among the shared dataspace operations.*

1 Introduction

The rapid evolution of web-based infrastuctures calls for the creation of suitable architectural styles and methodologies for the development of distributed applications. In this context, agent technologies have emerged as one of the most promising approaches. However, there is not yet a common agreement on which is the most suitable inter-agent communication and coordination platform. Two main architectures have been proposed in the literature:

1. *Publish/Subscribe*: inter-agent communication takes place through the raising of events performed by agents via a *publish* operation. Events are multicast to those agents which have previously registered their interest via a *subscribe* operation.
2. *Shared dataspace*: agents communicate by producing, consuming, and testing for the presence of objects in a shared, common repository.

The two approaches have been adopted in many proposals for inter-agent communication infrastructures: see, e.g., JEDI [4] for the publish/subscribe approach and Linda [6], WCL [10], MARS [3], and Lime [9] for the shared dataspace one. Despite of their diffusion, a rigorous comparison of the two models is still lacking. For example, questions such as "which approach is more convenient for this application?" or "are there problems which can be solved with one approach but not with the other one?" are still waiting for a definitive answer.

This paper is a first step towards a deeper understanding of the similarities and differences of the two architectures; in particular, we study the interchangeability, namely the possibility to substitute one mechanism in place of the other without affecting the internal behaviour of the agents.

We show that the shared dataspace can be reduced to the publish/subscribe model simply by adding agents which have the responsibility to represent the state of the simulated common repository. On the other hand, a reduction of the publish/subscribe into the shared dataspace model cannot be provided in the case only the basic output, input, and read dataspace operations are considered. Interestingly, we show that this negative result does not hold any more as soon as a global dataspace operation, such as the *copy-collect* primitive proposed in [11], is added. This coordination operator, introduced in order to solve the well-known *multiple read* problem of Linda, has the ability to collect all the data satisfying a certain pattern available in the dataspace.

The paper is organized as follows: Section 2 reports some informal observations on the comparison between events and data, the two abstractions used in the considered architectures to carry information among agents; Sections 3 and 4 respectively investigate the reduction of the shared dataspace to the publish/subscribe architecture, and the vice versa; finally, Section 5 reports some conclusive remarks.

2 Events vs. Data

Our analysis starts from an investigation of the similarities and differences between *events* and *data*, the two abstractions exploited by the considered infrastructures to obtain agent communication. We perform our analysis by considering three different aspects: *creation*, *life-time*, and *visibility*.

The creation is non-blocking both for events and data: an agent can raise an event in each possible context and an agent can introduce a new datum in a shared repository whatever is its actual state.

A first basic difference can be observed on the lifetime: after its raising, an event plays a role in the overall system only during the multicast protocol; on the other hand, a datum remains available in the dataspace until it is explicitly withdrawn. This property is usually referred to as *generative communication* [6]: a datum, after its production, has an independent life inside the dataspace.

Concerning the visibility, it is worth to point out at least two differences between events and data. A datum can be read from any agent, even from agents not present in the system at the time the datum was produced; this property is usually referred to as *time-uncoupling*. On the other hand, an event can be observed only by those agents which registered their interest before the raising of the event. A second observation concerns the ability to perform a destructive consumption of information: data can be removed from the dataspace, thus disallowing other agents to read it; on the other hand, an agent cannot hide an event to the other agents in the system.

3 Reducing shared dataspace to publish/subscribe

A trivial encoding of shared dataspace coordinated systems using the publish/subscribe infrasctructure could exploit a specialized agent, that we call the *dataspace agent*, responsible to simulate the shared repository; agents produce data by raising events, the dataspace agent stores all these events, and the agents interested in reading or consuming data simply interact with the dataspace agent.

Even if correct, this encoding is not completely satisfactory in distributed applications, because the dataspace agent is a bottleneck of the system. This problem can be solved simply by partitioning the dataspace and give the responsibility for the management of each partition to different agents. Bringing this approach to the limit, we could think to model each single datum with a separate agent.

We now introduce two simple calculi, $\mathcal{DS}$ and $\mathcal{PS}$, based on the shared dataspace and the publish/subscribe approaches, respectively; after, we present an enconding of $\mathcal{DS}$ in $\mathcal{PS}$ inspired by the above approach (each emitted datum is represented by a separate process).

The calculi we introduce borrow typical techniques from the tradition of process calculi for concurrency (e.g., Milner's CCS [7]); an agent is described as a term of an algebra where the basic actions are typical coordination primitives.

We start by presenting $\mathcal{DS}$, the calculus based on the shared dataspace coordination model. To be general, we consider a denumerable set of data, denoted by $Data$, ranged over by a, b, ... The set $Agent$ of agents, ranged over by A, A', ..., is the set of terms generated by the following grammar:

$$\begin{array}{lcl} A & ::= & \mathbf{0} \ \mid \ \mu.A \ \mid \ K \\ \mu & ::= & out(a) \ \mid \ in(a) \ \mid \ rd(a) \ \mid \ eval(A) \end{array}$$

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

Agent $\mathbf{0}$ is an agent that can do nothing. Agent $\mu.A$ is an agent that can do the action μ and then behaves like A.

A shared dataspace system configuration is composed by a multiset of active agents and by a multiset of available data. Formally, a system configuration is a pair $(Ag, DS) \in \mathcal{M}(Agent) \times \mathcal{M}(Data)$ where $\mathcal{M}(Set)$ is used to denote the set of the multisets of elements taken from Set. A multiset is usually represented by the classic set notation (using the brackets $\{\,\}$ which we omit in the case of singletons), but where multiplicity of occurrences is relevant. Given a multiset M we use $M(s)$ to denote the number of occurrences of s in M. Multiset union is denoted by $\oplus$ and we will use also $\bigoplus_n M$ to denote the multiset union of n instances of the multiset M.

A system configuration evolves according to the execution of coordination operations. This is formally described by a transition system, defined by the set of rewriting rules on system configurations listed in Table 1 (where we use characters in bold to emphasize the elements directly involved in the considered transitions).

The rules can be described as follows. The execution of the output operation has the effect of adding the emitted datum to the dataspace. On the other hand, the input operation removes the datum, if it is present, otherwise the rule does not apply (blocking operation). Similarly for the read operation, where however the datum is not removed. The execution of $eval(A')$ enables a new instance of agent A' to be added to the multiset of agents. Finally, an agent name K can do what its defining agent A can do.

$$
\begin{array}{l}
(\mathbf{out}(\mathbf{a}).\mathbf{A} \oplus Ag, DS) \longrightarrow (\mathbf{A} \oplus Ag, DS \oplus \mathbf{a}) \\
(\mathbf{in}(\mathbf{a}).\mathbf{A} \oplus Ag, DS \oplus \mathbf{a}) \longrightarrow (\mathbf{A} \oplus Ag, DS) \\
(\mathbf{rd}(\mathbf{a}).\mathbf{A} \oplus Ag, DS \oplus \mathbf{a}) \longrightarrow (\mathbf{A} \oplus Ag, DS \oplus \mathbf{a}) \\
(\mathbf{eval}(\mathbf{A}').\mathbf{A} \oplus Ag, DS) \longrightarrow (\mathbf{A}' \oplus \mathbf{A} \oplus Ag, DS) \\
\dfrac{(\mathbf{A}, \mathbf{DS}) \longrightarrow (\mathbf{A}', \mathbf{DS}')}{(\mathbf{K} \oplus Ag, \mathbf{DS}) \longrightarrow (\mathbf{A}' \oplus Ag, \mathbf{DS}')} \quad \text{if } K = A
\end{array}
$$

Table 1. The semantics of the shared dataspace calculus $\mathcal{DS}$.

We are now ready to introduce $\mathcal{PS}$, the calculus based on the publish/subscribe approach. Agents are modeled as in the previous calculus as terms comprising the basic coordination primitives; in this case we consider primitives for event publication $pub(e)$, event subscription $sub(e, A)$, event unsubscription $unsub(e, A)$, plus an operator for agent creation $new(A)$. The result of a subscription $sub(e, A)$ consists of the production of a listener responsible to catch the event e, and then activate the corresponding reaction A. More precisely, whenever an instance of the event e is raised as result of the execution of $pub(e)$, the listener should create a new agent A. The listener can be removed exploiting the event unsubscribe operation $unsub(e, A)$.

Let $Event$, ranged over by e, f, $\ldots$, be a denumerable set of possible events which can be exchanged among agents. Also in this calculus $Agent$ is the set of possible agents, and $Name$ is the set of agent names. In this case the definition of $Agent$ adopts a different set of coordination primitives:

$$\mu ::= pub(e) \;|\; sub(e, P) \;|\; unsub(e, P) \;|\; new(A)$$

A publish/subscribe system configuration is composed by a multiset of active agents and by a multiset of active listeners. Formally, a system configuration is a pair $(Ag, LS) \in \mathcal{M}(Agent) \times \mathcal{M}(Event \times Agent)$. A pair $\langle e, A \rangle$ represents a listener which is responsible for the activation of agent A each time an event e is raised.

The rules for $\mathcal{PS}$ are listed in Table 2. The execution of the subscription operation adds a new listener, while the execution of an unsubscription removes one instance (if available) of the specified listener. The publish operation activates a new multiset of agents representing the reactions to the raised event: this set of agent, denoted by $\{\mathbf{A}' \text{ for each occurrence of } \langle \mathbf{e}, \mathbf{A}' \rangle \text{ in } LS\}$ corresponds to the multiset obtained by taking an agent A' for each occurrence of active listener $\langle e, A' \rangle$. The new operation simply produces a new agent, while the last rule indicates that an agent name K can do what its defining agent A can do.

We are now ready to present the encoding $[\![\;]\!]$ which maps each agent of the calculus $\mathcal{DS}$ in a corresponding agent of $\mathcal{PS}$. As explained at the beginning of this section we represent data with processes; more precisely, each emitted datum a is modeled with a listener $\langle a, \mathbf{0} \rangle$. Following this approach, an output operation is modeled with a subscription operation which produces the corresponding listener, an input operation becomes an unsubscribe, and a read operation simply consumes and reproduces the listener.

Without loss of generality we assume that for each datum $a \in Data$ and each agent name K of $\mathcal{DS}$ there exist a corresponding distinct event name $a \in Event$ and an agent name K of $\mathcal{PS}$. The encoding is inductively defined as follows:

$$
\begin{array}{rcl}
[\![\mathbf{0}]\!] & = & \mathbf{0} \\
[\![K]\!] & = & K \\
[\![out(a).A]\!] & = & sub(a, \mathbf{0}).[\![A]\!] \\
[\![in(a).A]\!] & = & unsub(a, \mathbf{0}).[\![A]\!] \\
[\![rd(a).A]\!] & = & unsub(a, \mathbf{0}).sub(a, \mathbf{0}).[\![A]\!] \\
[\![eval(A').A]\!] & = & new([\![A']\!]).[\![A]\!]
\end{array}
$$

where we assume a definition $K = [\![A]\!]$ in $\mathcal{PS}$ for each definition $K = A$ in $\mathcal{DS}$. It is interesting to observe that this encoding does not exploit the publish operation.

The correctness of the encoding is proved by the fact that given an initial configuration $(Ag, \emptyset)$ of $\mathcal{DS}$, the corresponding initial configuration $([\![A]\!], \emptyset)$ of $\mathcal{PS}$ (obtained by substituting each agent A in Ag with its mapping $[\![A]\!]$) has exactly the same possible computations. The unique difference is in the encoding of the read operations, which are no longer executed atomically, but obtained as composition of an input and an output operation. It is worth noting that this approach is correct because we do not consider any test for absence operator able to discriminate between a read operation and an input followed by an output.

$$
\begin{array}{l}
(\mathbf{sub}(\mathbf{e},\mathbf{A}').\mathbf{A} \oplus Ag, LS) \longrightarrow (\mathbf{A} \oplus Ag, LS \oplus \langle \mathbf{e}, \mathbf{A}' \rangle) \\
(\mathbf{unsub}(\mathbf{e},\mathbf{A}').\mathbf{A} \oplus Ag, LS \oplus \langle \mathbf{e}, \mathbf{A}' \rangle) \longrightarrow (\mathbf{A} \oplus Ag, LS) \\
(\mathbf{pub}(\mathbf{e}).\mathbf{A} \oplus Ag, LS) \longrightarrow (\mathbf{A} \oplus Ag \oplus \{\mathbf{A_i} \text{ for each occurrence of } \langle \mathbf{e}, \mathbf{A_i} \rangle \text{ in } LS\}, LS) \\
(\mathbf{new}(\mathbf{A}').\mathbf{A} \oplus Ag, LS) \longrightarrow (\mathbf{A}' \oplus \mathbf{A} \oplus Ag, LS) \\
\dfrac{(\mathbf{A}, \mathbf{LS}) \longrightarrow (\mathbf{A}', \mathbf{LS}')}{(\mathbf{K} \oplus Ag, \mathbf{LS}) \longrightarrow (\mathbf{A}' \oplus Ag, \mathbf{LS}')} \quad \text{if } K = A
\end{array}
$$

Table 2. The semantics of the publish/subscribe calculus $\mathcal{PS}$.

4 Reducing publish/subscribe to shared dataspace

In the previous section, we provided an encoding of $\mathcal{DS}$ on $\mathcal{PS}$. In this section, we show that the reverse encoding can be obtained only at the price of extending the calculus $\mathcal{DS}$ with a new primitive, permitting to "count" the exact number of occurrences of a datum in the dataspace.

The impossibility result of giving an encoding of $\mathcal{PS}$ in $\mathcal{DS}$ is obtained by providing a property of agents that is decidable for $\mathcal{DS}$ but not for $\mathcal{PS}$. The property we consider is the existence of a finite computation (called henceforth *termination*), which is decidable for $\mathcal{DS}$ but turns out to be undecidable for $\mathcal{PS}$. The decidability of termination in $\mathcal{DS}$ is obtained by providing an encoding of $\mathcal{DS}$ on finite place/transition Petri nets which preserves the behaviour of agents; as termination is decidable for this class of Petri nets, we obtain decidability of termination also for $\mathcal{DS}$. (the technical details of this encoding can be found in [1]). On the other hand, in Appendix A we show the undecidability of termination in $\mathcal{PS}$ by providing a termination-preserving encoding of Random Access Machines (RAM) [12], a Turing powerful formalism, in $\mathcal{PS}$.

Now we show how to extend the calculus $\mathcal{DS}$ with a set of primitives sufficiently powerful to encode the calculus $\mathcal{PS}$. Actually, it is sufficient to extend $\mathcal{DS}$ with an adaptation of the *copy-collect* primitive proposed in [11], suitable for our calculus. This primitive collects all the data satisfying a certain pattern; in our setting, we introduce a $ccollect(a, A)$ operation, which realizes how many occurrences of datum a are currently available, and for each of these occurrences spawns a copy of agent A.

We introduce the new calculus $\mathcal{DS}cc$, obtained from $\mathcal{DS}$ by extending the action prefixes μ in the following way:

$$\mu \ ::= \ \ldots \mid \ ccollect(a, A)$$

The semantics of $\mathcal{DS}cc$ is obtained by extending the rules for $\mathcal{DS}$ with the following rule, dealing with the $ccollect$ primitive:

$$
\begin{array}{l}
(\mathbf{ccollect}(\mathbf{a},\mathbf{A}).\mathbf{A}' \oplus Ag, DS) \longrightarrow \\
(\mathbf{A}' \oplus \bigoplus_{\mathbf{DS(a)}} \mathbf{A} \oplus Ag, DS)
\end{array}
$$

Now we present the encoding of $\mathcal{PS}$ on $\mathcal{DS}cc$, starting with an auxiliary definition.

Suppose we want to provide the encoding of agent A with a set of constant definitions $K_i = A_i$, for $i = 1, \ldots, n$.

We need to define the set of agents that, at some point in the computation, could be contained in listeners of an event e. We define the set of *possible agents in listeners* for e on A as the set of agents $\mathcal{PAL}_e = PAL_e(A) \cup PAL_e(A_1) \cup \ldots \cup PAL_e(A_n)$, where PAL_e is defined by induction on the structure of agents and prefixes:

$$
\begin{array}{lcl}
PAL_e(\mathbf{0}) & = & \emptyset \\
PAL_e(\mu.A) & = & PAL_e(\mu) \cup PAL_e(A) \\
PAL_e(K) & = & \emptyset \\
PAL_e(pub(e)) & = & \emptyset \\
PAL_e(sub(e, A)) & = & \{A\} \cup PAL_e(A) \\
PAL_e(unsub(e, A)) & = & \emptyset \\
PAL_e(new(A)) & = & PAL_e(A)
\end{array}
$$

Note that the set of possible agents in listeners of an agent is always finite.

Pursuing the analogy between listeners and data proposed in the previous sections, we represent a listener on event e, that produces a copy of agent A when waked, by a datum s_{eA} in the dataspace. Hence, subscription corresponds to the production of a new datum s_{eA}, whereas unsubscription corresponds to a consumption of the same datum.

The most involved part of the encoding is concerned with the publish primitive, which makes the auxiliary definition of $\mathcal{PAL}_e$ and the $ccollect$ primitive to come into play. For each possible agent A in a listener of event e, for each occurrence of datum s_{eA} a copy of agent A is spawned through a $ccollect$ operation. All these operations make the encoding of a publish operation non-atomic; to avoid undesired behaviours these operations must be performed in mutual exclusion with other publish or subscribe operation on the event e. To obtain mutual exclusion we assume the presence of a datum m_e in the dataspace for each event e; this datum should be removed before the execution of a sequence of operations in mutual exclusion, and reproduced at the end.

The encoding is formally defined as follows:

$$\begin{array}{rcl}
[\![\mathbf{0}]\!] & = & \mathbf{0} \\
[\![K]\!] & = & K \\
[\![new(A).A']\!] & = & eval([\![A]\!]).[\![A']\!] \\
[\![sub(e,A).A']\!] & = & in(m_e).out(s_{eA}).out(m_e).[\![A']\!] \\
[\![unsub(e,A).A']\!] & = & in(m_e).in(s_{eA}).out(m_e).[\![A']\!] \\
[\![pub(e).A]\!] & = & in(m_e). \\
 & & ccollect(s_{eA_{e1}}, [\![A_{e1}]\!]).\ldots. \\
 & & ccollect(s_{eA_{en_e}}, [\![A_{en_e}]\!]). \\
 & & out(m_e).[\![A]\!]
\end{array}$$

where $\{A_{e1}, \ldots, A_{en_e}\} = \mathcal{PAL}_e$ for each event e; moreover, we assume a definition $K = [\![A]\!]$ in $\mathcal{DS}cc$ for each definition $K = A$ in $\mathcal{PS}$.

The correctness of the encoding is proved by the fact that given an initial configuration $(Ag, \emptyset)$ of $\mathcal{PS}$, the corresponding initial configuration $([\![Ag]\!], DS)$ of $\mathcal{DS}cc$, obtained by subsituting each agent $A \in Ag$ with its mapping $[\![A]\!]$ and taking DS as the set of the data m_e necessary for mutual exclusion, has exactly the same possible computations. In this case, the unique difference is in the publish operation which is no longer atomic but implemented in terms of a sequence of *ccollect* operations; however, no undesired behaviours are introduced because these operations are executed in mutual exclusion with potentially interfeering primitives.

5 Conclusion

In this paper we have initiated a rigorous comparison of the publish/subscribe and the shared dataspace interagent coordination infrastructures. More precisely, we have defined two simple process calculi based on the two approaches, and we have exploited them in order to investigate their interchangeability, that is, the possibility to replace one infrastructure in terms of the other without altering the internal bahaviour of the agents.

The interesting result is that a basic shared dataspace infrastructure with only output, input, and read coordination operations can be modeled using the publish/subscribe architecture, while the vice versa does not hold. In order to simulate a publish/subscribe architecture exploiting a shared dataspace, it is necessary to add global coordination primitives such as, e.g., the *copy-collect* of [11].

We plan to continue our investigation by considering also other global operations; for example, we claim that the predicative input *inp* of Linda [6] is another candidate to cover the gap of expressiveness between the publish/subscribe and the shared dataspace model. Other interesting primitives are the so-called reaction-based dataspace operations: see, e.g., the *notify* of JavaSpaces [5], the *monitor* of WCL [10], or the *reactsTo* of Lime [9] which permit to observe events occurring inside the dataspace such as, e.g., the introduction of data satisfying a certain pattern. Concerning the expressiveness of the JavaSpaces $notify$ primitive some preliminary results can be found in [2].

References

[1] N. Busi, R. Gorrieri, and G. Zavattaro. On the Expressiveness of Linda Coordination Primitives. *Information and Computation*, 156:90–121, 2000.

[2] 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.

[3] G. Cabri, L. Leonardi, and F. Zambonelli. Reactive Tuple Spaces for Mobile Agent Coordination. In *Proc. 2nd Int. Workshop on Mobile Agents*, volume 1477 of *Lecture Notes in Computer Science*, pages 237–248. Springer-Verlag, Berlin, 1998.

[4] G. Cugola, E. D. Nitto, and A. Fuggetta. Exploiting an event-based infrastructure to develop complex distributed systems. In *Proc. of the 20th ICSE*, 1998.

[5] E. Freeman, S. Hupfer, and K. Arnold. *JavaSpaces Principles, Patterns, and Practice*. Addison-Wesley, 1998.

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

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

[8] M. Minsky. *Computation: finite and infinite machines*. Prentice-Hall, 1967.

[9] G. Picco, A. Murphy, and G.-C. Roman. Lime: Linda meets mobility. In *Proc. of the 21th ICSE*, 1999.

[10] A. Rowstron. Wcl: A co-ordination language for geographically distributed agents. *World Wide Web Journal*, 1(3):167–179, 1998.

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

[12] J. Shepherdson and J. Sturgis. Computability of recursive functions. *J. ACM*, 10:217–255, 1963.

A Proof of the undecidability of termination in $\mathcal{PS}$

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

The execution of the program begins with the first instruction and continues by executing the other instructions in sequence, unless a jump instruction is encountered. The execution stops when an instruction number higher than the length of the program is reached.

If the program terminates, the result of the computation is the contents of the registers. In [8] it is shown that the following two instructions are sufficient to model every recursive function:

$Succ(r_j)$: adds 1 to the content of register r_j;

$DecJump(r_j, s)$: if the content of r_j is not zero, then decreases it by 1 and goes to the next instruction, otherwise jumps to instruction s.

We restrict to RAMs which start and finish their computation (in the case they terminate) with all the registers empty; it is a well known result that this restriction does not decrease the expressiveness of the formalism.

Now we present the encoding of RAM in $\mathcal{PS}$. The encoding is *nondeterministic*, as it introduces some extra infinite computation; nevertheless, it is ensured that a RAM terminates if and only if the corresponding encoding has a terminating computation. As termination cannot be decided in Turing equivalent formalisms, the same holds also for $\mathcal{PS}$.

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

Given the RAM program R composed by the instructions $I_1 \dots I_k$, the corresponding encoding is the agent below:

$$\begin{array}{l} sub(loop_1, pub(loop_1)).sub(loop_2, pub(loop_2)). \\ sub(p_1, P_1). \dots .sub(p_k, P_k).pub(p_1) \end{array}$$

If the i-th instruction is an increment of register r_j, the constant P_i is defined as

$$P_i = sub(isz_j, pub(loop_2)).pub(p_{i+1})$$

If the i-th instruction is a decrement of register r_j, or jump to instruction s, the constant P_i is defined as

$$\begin{array}{rl} P_i = & sub(c_i, \mathbf{0}). \\ & eval(unsub(c_i, \mathbf{0}).pub(loop_1). \\ & \quad unsub(isz_j, pub(loop_2)). \\ & \quad unsub(loop_1, pub(loop_1)). \\ & \quad sub(loop_1, pub(loop_1)).pub(p_{i+1})). \\ & eval(unsub(c_i, \mathbf{0}).pub(isz_j).pub(p_s)) \end{array}$$

If we observe the agent encoding the RAM, we note that, before starting simulating the execution of instructions, two listeners, $\langle loop_1, pub(loop_1)\rangle$ and $\langle loop_2, pub(loop_2)\rangle$, are produced. If, during the execution of the encoding of instructions, an event $loop_1$ (resp. $loop_2$) is published, the listener will produce an agent $pub(loop_1)$ (resp. $pub(loop_2)$), thus obtaining an infinite loop that can be broken only by an unsubscribe operation.

The basic idea of this encoding is to represent the actual content of each register r_j with a corresponding number of listeners $\langle isz_j, pub(loop_2)\rangle$. Moreover, every time an increment (or a decrement) on the register r_j is performed, a new listener $\langle isz_j, pub(loop_2)\rangle$ is spawned (resp. unsubscribed). In this way it is possible to check if the actual content of a register r_j is zero by verifying if there are no occurrences of listener $\langle isz_j, pub(loop_2)\rangle$.

There are two possible wrong choices that can be performed during the computation: (i) a decrement on a register containing zero or (ii) a jump for zero on a non-zero register.

In the case (i), the agent

$$\begin{array}{l} pub(loop_1).unsub(isz_j, pub(loop_2)). \\ unsub(loop_1, pub(loop_1)). \\ sub(loop_1, pub(loop_1)).pub(p_{i+1}) \end{array}$$

is activated with no listener $\langle isz_j, pub(loop_2)\rangle$ available. Thus, the agent publishes event $loop_1$ and blocks trying to execute $unsub(loop_1, pub(loop_1)$. However, event $loop_1$ wakes up the listener $\langle loop_1, pub(loop_1)\rangle$, that produces an agent $pub(loop_1)$, thus starting an infinite loop.

In the case (ii), the agent $pub(isz_j).pub(p_s)$ is activated when there is at least one occurrence of listener $\langle isz_j, pub(loop_2)\rangle$. When event isz_j is published, the above listener(s) produces one (or more) occurrences of event $loop_2$, thus starting an infinite loop with listener $\langle loop_2, pub(loop_2)\rangle$.

Finally, it may happen that, even if we execute a decrement on a non-zero register, the agent takes a divergent computation. In fact, consider the agent

$$\begin{array}{l} pub(loop_1).unsub(isz_j, pub(loop_2)). \\ unsub(loop_1, pub(loop_1)). \\ sub(loop_1, pub(loop_1)).pub(p_{i+1}) \end{array}$$

First, it starts an infinite loop with listener $\langle loop_1, pub(loop_1)\rangle$, then, after removing a listener on event isz_j, it breaks the loop by removing the listener on $loop_1$; at this point, a new listener on $loop_1$ is created: if the agent $pub(loop_1)$ is not executed before creating the new listener, the infinite loop will restart with the new listener. However, this fact does not create troubles, because we are interested only in preserving terminating computations.