μCRL specification of event notification in JavaSpaces™*

Jaco van de Pol and Miguel Valero Espada

Centrum voor Wiskunde en Informatica,
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
{Jaco.van.de.Pol, Miguel.Valero.Espada}@cwi.nl

Abstract. In this paper, we extend the formal specification of the JavaSpaces architecture presented in [18] with the event notification mechanism. Processes running on a JavaSpaces system can register their interest in incoming entries. The space informs the arrival of matching entries by sending events to the registered processes. We use μCRL, a language that combines abstract data types with process algebra, to model a formal abstraction of this mechanism. The purpose of this work, in combination with the previous one, is to verify properties of the JavaSpaces technology and to allow automatic model checking of distributed applications built under it.

1 Introduction

The parallel composition of simple behavior agents can produce complicated systems. Distributed applications have to manage with the communication and synchronization between processes across heterogeneous networks, dealing with latencies, partial failures and system incompatibilities. Hence coordination architectures attempt to assist programmers at the difficult task of designing and implementing reliable distributed systems.

JavaSpaces™ [16] technology is a Sun Microsystems, Inc. coordination architecture, implemented as a Jini™ [17] service. It gives support to two programming styles of processes coordination: the shared dataspace (Linda [8] like style) and a reactive style. External agents communicate by sharing objects through the space, by means of some basic primitives. They can basically write and look up objects but they can also express their interest in incoming entries, by registering using the *notify* primitive. Then the space is charged to inform the agents the presence of suitable entries by sending events. The external processes "react" to the arrival of new entries in the space.

In a previous paper [18], we studied the basic features of the shared dataspace style, now we are going to present the formal specification of the notification mechanism using μCRL [13, 11], a language which merges the standard process

* Partially supported by PROGRESS, the embedded systems research program of the Dutch organisation for Scientific Research NWO, the Dutch Ministry of Economic Affairs and the Technology Foundation STW, grant CES.5009.

algebra ACP [1] and abstract data types. By extending the model with the new operation, we allow to prototype and verify more JavaSpaces applications. The verification of the system is done by using the combination of the μCRL tool set [2] and the Cæsar Aldébaran Development Package (CADP) [10].

During the implementation of the μCRL model we had to face several difficulties in interpretation of the JavaSpaces specification. There are some issues that JavaSpaces specification leaves unclear or ambiguous and that are actually solved in the implementation. In our work, we attempt to clarify and resolve this lack of precision and detail.

This paper is structured as follows. After this introduction, we present the JavaSpaces specification and the μCRL language. We continue with the study of the formalism of the JavaSpaces architecture focusing on the notify primitive. Then, we illustrate the specification by modeling and model checking a simple application. The paper finishes with the conclusion and some references to other related works. The specification and some examples can be found at: "http://www.cwi.nl/~miguel/JavaSpaces/".

2 JavaSpaces

JavaSpaces is both an application program interface (API) and a distributed programming model. Agents can interact simultaneously with a shared dataspace of objects, the space handles the details of concurrent access to the data. Agents of applications are "loosely coupled", they do not communicate with each other directly but by sharing information via the common space. They use a small set of primitives described in Figure 1:

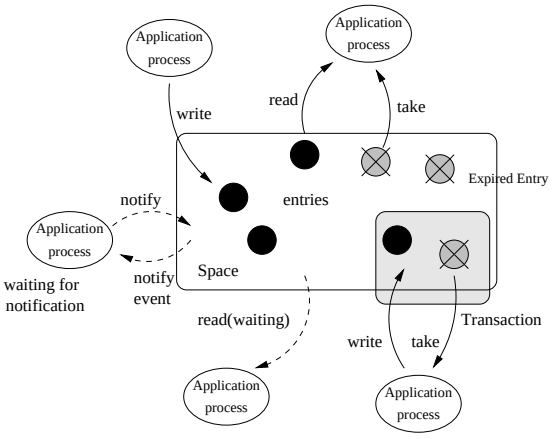

Fig. 1. JavaSpaces architecture overview

A *write* operation places a copy of an entry into the space. Entries can be located by "associative lookup" implemented by *templates*. Processes find the

entries they are interested in by expressing constraints about their contents without having any information about the object identification, owner or location. A *read* request returns a copy of an object from the space that matches the provided *template*, or *null* if no object has been found. If no matching entries are in the space, then *read* may wait a user-specified amount of time (*timeout*) until a matching entry arrives in the space. *ReadIfExists* performs exactly like *read*, but it only blocks if there are matching objects in the space but they have conflicting locks from one or more other transactions. *Take* and *takeIfExists* are the *destructive* versions of *read* and *readIfExists*: once an object has been returned, it is removed from the space.

The *notify* primitive is used to express interest in future incoming objects. The agent provides a template and the space will notify the agent when a matching object has arrived, by means of an event. Three entities are involved in the notification mechanism: The space is the source of events, it fires an event when an entry matches a registration. The destinations, called listeners, wait for the arrival of events and "react" to them. And the application process which register the listeners to be notified. The registration is done by the synchronous action *notify* based on the JavaSpaces specification [12].

```
public interface JavaSpace {
   ...;
   EventRegistration notify(Entry tmpl, Transaction txn,
         RemoteEventListener listener, long lease,
         MarshalledObject handback)
      throws RemoteException, TransactionException;
   ...;
}
```

The primitive gets as arguments the template to match entries, the reference to a transaction, the reference to the remote event listener, the lease and a handback used to pass information from the application process to the listeners. The space returns an *eventRegistration* object, which includes the registration identification number (the space assigns a identification number to any new registration), the granted lease and the initial sequence number for events generated from the *notify* registration. Every matching entry will increase by one the sequence number of the registrations. And newly generated event will contain a sequence number greater than the previous one.

JavaSpaces also provide support to leasing and transactions, from the Jini architecture [17]:

JavaSpaces supports a transactional model ensuring that a set of grouped operations are performed on the space atomically, in such a way that either all of them complete or none are executed. Transactions affect the behavior of the primitives, e.g. an object written within a transaction is not externally accessible until the transaction commits, the insertion will never be visible if the transactions aborts. Transactions provide a means for enforcing consistency. Transactions in JavaSpaces preserve the ACID properties: Atomicity, Consistency, Isolation and Durability.

JavaSpaces allocates resources for a fixed period of time, by associating a lease to the resource. The lease model is beneficial in distributed systems where partial failures can produce waste of resources. The space determines the time during which an object can be stored in the repository before being automatically removed. Also transactions are subject to leasing, an exception is sent when the lease of a transaction has been expired. Leases can always be renewed or canceled.

In the paper [18] we have presented the μCRL specification of the primitives: *write, read, take, takeIfExists and readIfExists*, leases and transactions. Now we are going to focus on the *notify* operation.

To know more about JavaSpaces, please consult the references [12, 16].

3 Introduction to μCRL

A μCRL specification is composed by two parts. First, the definition of the data types, called **sorts**. A sort consists of a signature in which a set of function symbols, and a list of axioms are declared. For example, the specification of the booleans (*Bool*) with the conjunction operator (*and*) is defined as follows:

```
sort  Bool
func  T,F:→Bool
map   and: Bool×Bool→Bool
var   b: Bool
rew   and(T, b) = b
      and(F, b) = F
```

The keyword **func** denotes the *constructor* function symbols and **map** is used to declare additional functions for a sort. We can add equations using variables (declared after **rew** and **var**) to specify the function symbols. The declaration of the sort *Bool* must be included in every μCRL specification because booleans are used for modeling the guards in the "if-then-else" construction.

The second part of the specification consists of the process definition. The basic expressions are actions and process variables. *Actions* represent events in the system, are declared using the keyword **act** followed by an action name and the sorts of data with which they are parameterized. Actions in μCRL are considered atomic. There are two predefined constants: δ which represents deadlock, and τ which is a hidden action. *Process variables* abbreviate processes, and are used for recursive specifications. *Process operators* define how the process terms are combined. We can use:

- The sequential, alternative and parallel composition $(.,+,\|)$ process operators.
- **sum** $(\sum)$ to express the possibility of infinite choice of one element of a sort.
- The conditional expression "if-then-else" denoted $p \lhd b \rhd q$, where b is a boolean expression, p and q process terms. If b is true then the system behaves as p otherwise it behaves like q.

They keyword **comm** specifies that two actions may synchronize. If two actions are able to synchronize we can force that they occur always in communication using the operator ∂_H. The operator τ_I hides enclosed actions by renaming into τ actions. The initial behavior of the system can be specified with the keyword **init** followed by a process term:

$\mathsf{System} = \tau_I \partial_H (p_0 \parallel p_1 \parallel ...)$
init System

4 μCRL Specification

The space is modeled as a single process called *javaspace*. External agents are implemented as separate processes executed in parallel with the space. A JavaSpaces system is specified in μCRL as follows:

$$\mathsf{System} = \tau_I \partial_H (javaspace(...) \parallel external\_P_0(id_0 : Nat, ...) \parallel external\_P_1(id_1 : Nat, ...) \parallel ...)$$

The arguments of the *javaspace* process represent the current state of the space. They are composed by: stored objects, active transactions, the current time, active operations, notify registrations, et cetera... External processes interact with the space by means of a set of synchronous actions, derived from the JavaSpaces API. Every process has a unique identification number used by the space to control the access to the common repository. Processes use the sort *Entry* to encapsulate the shared data. In the JavaSpaces specification, an entry corresponds to a serializable Java™ object which implements the public interface Entry (with some other restrictions). In our model, entries are represented by a **sort**. Users can define their own data structure according to the application requirements. The insertion of a new entry into the space is done with the action *write* which has four arguments: the process identification number of the sort *Nat* (naturals), the entry of the sort *Entry*, the lease of naturals and the reference to a transaction (*null* if it is not submitted to any one). When the space receives a write request, it automatically encapsulates the entry, with its lease and the reference to the transaction, in an new data sort (*Object*) and stores it in the database which has the structure of a *Set*.

Look up primitives could be classified as: *destructive* and *non-destructive*, depending on whether the item is removed or not after the execution of the action, and in *blocking* and *non-blocking* depending on whether the process waits until it receives the requested item. We can invoke destructive look ups (*take*) or non-destructive (*read*), setting up the time during which the action blocks.

The JavaSpaces specification says that a look up request searches in the space for an Entry that matches the template provided in the action. If the match is found, a reference to a copy of the matching entry is returned. If no match is found, *null* is returned. We do not use templates to model the matching operation but by adding to every invocation one predicate, as argument, which determines if an Entry matches or not the action. This predicate belongs to the

sort *Query*, defined by the user according to the specification of the *Entry*. The sort must include the operator *test* used to perform the matching. An entry of the space will match a look up action if it satisfies the associated *test* predicate. The look up operations are not atomic. They are done by two synchronous actions; first the process makes the request and blocks waiting for an entry or for the timeout expiration. First, the space stores this request in a set with other pending requests and afterward the space returns a matching entry or the null value.

The behavior of all the primitives would be slightly different depending on whether they are executed under a transaction or not. Before focusing on the *notify* primitive let's see a small example of code illustrating the presented operations. The example is a recursive process which renames entries of type A to type B. It performs the operation under a transaction leased for one time unit. If the timeout of the transaction expires the space raises an exception and sent it to the process by means of a synchronous action (*exception*), then the process deadlocks:

proc ren(id:Nat) =
$\cdot\sum_{trc:Nat}$ (create(id, trc, S(0))
.(take(id, trc, FOREVER, isTypeA) + Exception(id, trc).δ)
.$\sum_{e:Entry}$ ((Return(id,e) + Exception(id, trc).δ)
.(write(id, renameToB(e), trc, FOREVER)+ Exception(id, trc).δ))
.(commit(id, trc) + Exception(id, trc).δ))
.ren(id).δ

Now, we are going to focus on the specification of the notify mechanism introduced in Section 2.

For simplicity, we have abstracted away the lease, the transaction and the handback but we will comment the inclusion of the first two fields later in this section. The template is replaced by a query. We assume the registration is done atomically, thus no events can be fired between the begin of the registration and its return. Therefore the initial sequence number of events will be zero. Due to these abstractions, the space only returns a single value representing the registration identification number. This operation is performed reliably so it cannot throw any exception. The action signature is:

sort Nat, Query
act notify: `Nat×Nat×Query×Nat`

The arguments are: the process identification number, the listener identification number, the query and the event registration identification number (provided by the space).

When the space synchronizes with a *notify* action, it stores the registration in a set. For each newly written entry it will check every registration to know whether it has to be notified or not. In other words, the space marks the registrations whose query matches the new entry. It also increments by one the event sequence number. The specification says that the space makes a "best effort"

to deliver the notifications, a notification event will be *eventually* sent to the registered listeners. The space does not guarantee the generation of an event for every matching entry, so several matching entries can be stored before the space decides to fire a message. The sequence numbers are useful to keep track of the events, as we will see on a small example in Section 5.

An event listener is an object that reacts to the reception of an event and that may be running remotely. The listener has a method (*notify*[1]) invoked whenever it receives a notification event. According to the JavaSpaces specification the *notify* call is synchronous so the space waits on a listener until the call finishes, but the JavaSpaces implementations are multi-threaded hence many different notify calls can be done concurrently. We modeled the *notify* operation with our single *javaspace* process, assuming that we have an implementation with enough threads to manage all the notifications of the system. The μCRL space delivers the event and doesn't wait until the end of the method call of the listener. This policy will not be admissible if there are too many listener registrations or if the *notify* methods are very slow (block the space for long periods) or never return. Our model would help programmers to take care about preserving the desirable behavior of the system, for example we will show one application in Section 5 the detection of a problem in a listener which arrives to a non desired blocked state.

The event sent by the space contains some data values. It includes the registration identification number, to allow a listener to distinguish the event as belonging to a particular registration and the sequence number of events, which can be used by listeners to know the number of events occurred from last notification.

In our model, listeners are going to be modeled as separate processes. μCRL does not allow the instantiation of processes on running time so listeners have to be defined at the beginning, according to the needs of the application. A listener has the following structure:

proc listener(id:Nat, d_0:D_0,...,d_n:D_n) =
$\sum_{registrationID:Nat}(\sum_{seq:Nat}$ (_Notify(id, registrationID, seq)
.*do_work*
.listener(id)))

Where d_0:D_0,...,d_n:D_n are the user defined arguments, and *_Notify* the action for receiving the event. The *.do_work* operation may be composed of any computation or any communication with other processes or with the space.

Messages travel over the network from the event source (the space) to the event destination (the listener); they are not delivered instantaneously nor reliably. Hence events may be lost and never reach their destinations. They may also arrive unordered because events can follow different paths. The event source can always duplicate messages because it cannot be sure whether the delivered events have arrived or not. To model this non-deterministic behavior, we specify

[1] Do not confuse with the registration notify method.

a separate process which represents the network situated between the source and the destination. This process stores the events in a fifo list, and can always:

- Receive: The network process receives and event from the space by means of the synchronous action $\_Notify$.
- Loose: It removes the first object of the list.
- Swap: It changes the order of the first two events that have the same destination.
- Duplicate: It replicates the first element of the list.
- Deliver: The networks dispenses an event to a listener using the action $\_notify$.

We parameterized the network process with a field counting the number of errors (looses, duplications and swaps). To keep finite the system we only allow a maximum number of errors. A reliable network would have a maximum number of errors equal to zero.

The complete system is composed of the parallel composition of the application processes, listeners, the space and the network. Figure 2 illustrates the model.

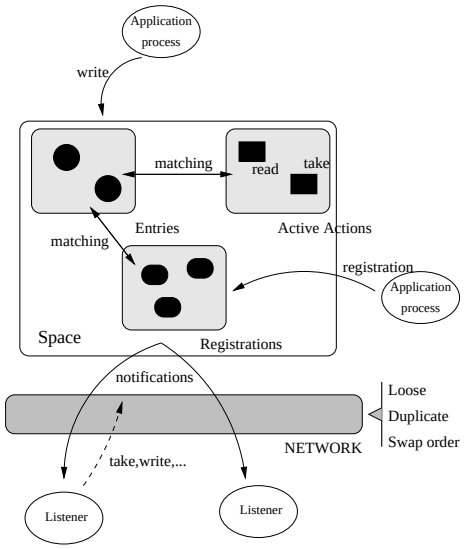

Fig. 2. Notification architecture

We can also add leasing to the registration mechanism. We proceed in the same way as for the look up primitives. The application process passes the requested lease to the space, it includes this value in a data field of the registration object. The space manages a centralized clock implemented as a discrete counter. The *javaspace* process increments this clock arbitrarily. Using this counter we can manage the expiration of the leases. When the registration lease expires the space automatically removes the object from the data base. Listeners can receive

events even if the registration has been removed, because the messages may be delayed on the network.

Notify can also be joined to a transaction. The space will send events when a matching entry is written under the same transaction of the registration or under the *null* transaction. If a transaction expires the joined registrations will be removed. We have not implemented these two issues (transactions and leasing) for the *notify* primitive, but according to its specification for the other primitives we foresee no major difficulties to do it. This concludes the presentation of the μCRL model, now let's analyze a simple application.

5 Example

In this section we are going to illustrate the use of the proposed specification to model check JavaSpaces applications by analyzing a simple example. The system consists of a process which registers a listener, expressing interest in any new data of the type *message*. When the listener receives the event, it just takes the message and prints *"Hello World"* (see the example in Chapter 8 of "JavaSpaces Principles, Patterns, and Practice" [12]). First we specify the sort *Entry*, which only has two possible values the compulsory null entry, and the message (we don't care about its content). The μCRL code is as follows:

```
sort  Entry
func  entryNull:→Entry
      message: →Entry
map   eq: Entry×Entry→Bool
var   e: Entry
rew   eq(entryNull,entryNull) = T
      eq(message, message) = T
      eq(entryNull, message) = F
      eq(message, entryNull) = F
```

We define two queries: *any* which matches any entry and *isMessage* which matches the entries of type *message*. Let's see the code:

```
sort  Query
func  any: →Query
      isMessage: →Query
map   test: Query×Entry→Bool
      eq: Query×Query→Bool
var   e: Entry
rew   test(any, e) = T
      test(isMessage, message) = T
      eq(any,any) = T
      eq(isMessage,isMessage) = T
      eq(any,isMessage) = F
      eq(isMessage,any) = F
```

The user application is composed by two processes. *Apps* executes the action *notify* registering the listener and the listener, that first gets the event and then tries to take the entry from the space. If the take is successful it does the action *HelloWorld*. Note that we have simplified the primitives *take* and *write* by removing the lease and the transaction. The code is:

proc apps(id: Nat, listenerID: Nat) =
$\sum_{registrationID:Nat}$ (notify(id, listenerID, isMessage, registrationID))
.write(id, token).δ

proc listener(id:Nat) =
$\sum_{registrationID:Nat}(\sum_{seq:Nat}$ (_Notify(id, registrationID, seq)))
.take(id, isMessage)
.waiting
.$\sum_{e:Entry}$ (Return(id,e))
.HelloWorld
.endNotify(id, registrationID, seq)
.listener(id)

Finally, the complete system is composed by the parallel composition of the space, the *apps* process, the listener and the network, which is allowed to commit one error.

System = $\partial_{\{write,Write,notify,Notify,\_notify,\_Notify,...\}}$
$(javaspace(0, emN, emA, 0)||Network(emE, 0, S(0))\,||$
$apps(0, S(0))\,||\,listener(S(0)))$

To each μCRL specification belongs a labeled transition system (LTS) being a directed graph, in which the nodes represent states and the edges are labeled with actions. If this transition system has a finite number of states the μCRL tool set can automatically generate this graph. Subsequently, the CÆSAR ALDÉBARAN DEVELOPMENT PACKAGE (CADP) can be used to visualise and to analyse this transition system. Figure 3 shows the generated LTS of the simple HelloWorld application where the action N corresponds to the communication between the *notify* action of the application process and the *Notify* action of the space, W to the write actions, $\_N$ corresponds to send an event from the space to the network, $\_\_N$ to deliver it from the network to the listener, T corresponds to a take request and Rt is the return of the *take*. *Duplicate, loose, HelloWorld, waiting* and *endNotify* are external actions informing about the execution of the system. Remark that the action *endNotify* is just a "printed" message, listeners do not synchronize with the space at the end of the *notify* invocation.

We can see in the figure 3 a desirable execution following the path: 0-1-2-3-4-7-9-11-13-15) and two undesired behaviors. The first is when the network looses the data (path: 0-1-2-3-5), thus the listener doesn't receive the message. The other is when the network duplicates the event(0-1-2-3-6-8-10-12-14-15-16-17-18-19-20), then listener tries to take two times the message. In this case the listener gets blocked *waiting* for a return that will never happen unless another

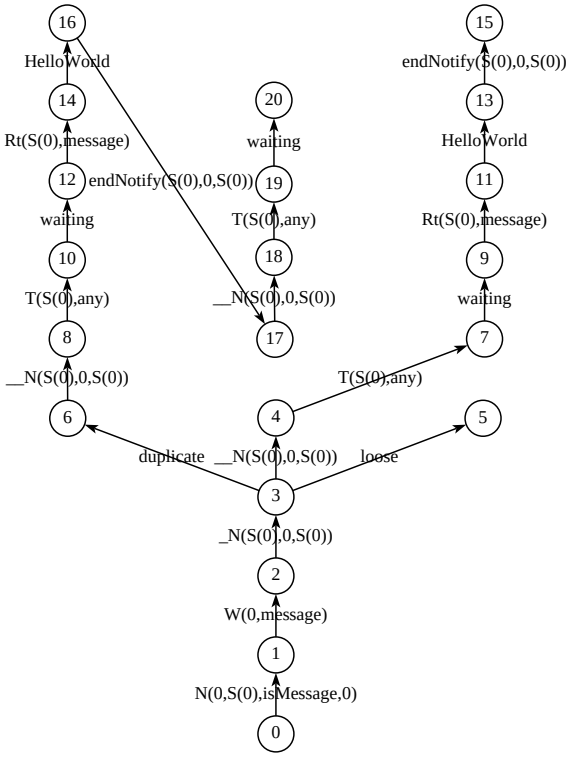

Fig. 3. LTS of HelloWorld in a non reliable network

process writes a new message. We can avoid the second undesired behavior by checking the event sequence number before trying to perform the take primitive. This is modeled with the following code:

proc listener(id:Nat, last:Nat) =
$\sum_{registrationID:Nat}(\sum_{seq:Nat}$ (_Notify(id, registrationID, seq)
.(take(id, isMessage)
.waiting
$\sum_{e:Entry}$ (Return(id,e))
.HelloWorld $\lhd$ gt(seq, last) $\rhd$ do_nothing)
.endNotify(id, registrationID, seq)
.listener(id)))

The listener only tries to take the message if the sequence number (*seq*) is greater than the sequence number of the last notification (*last*), otherwise it assumes that the event has been duplicated and finishes.

We can automatically verify some properties of the system using the Evaluator tool from the CADP package. These properties are expressed in temporal logic. We used the regular alternation-free μ-calculus formulas [15]. For example, the following formula means that every *notify* invocation of a listener finishes, in other words: after a *__N* there is always an *endNotify* with the same arguments:

```
[true*.'__N(.*)']mu X.(<true>true and [not 'endNotify(\(.*\))']X)
```

This formula does not hold for the first example of listener. The evaluator analyzes it and gives the counter example corresponding to already pointed path. However the second listener, which checks the sequence number of events satisfies the property.

6 Related Work

As we said, this work is an extension of the [18]. Our information of JavaSpaces is based upon the book [12], and the documentation from Sun on JavaSpaces [16] and Jini [17]. The latter document describes a.o. the concepts of leasing, transactions and distributed events. The basic ideas of JavaSpaces go back to the coordination language Linda [8].

Some work on the formalization of JavaSpaces (or other Linda-like languages) exist, notably [4, 5, 6, 7]. In these papers, an operational semantics of JavaSpaces programs is given by means of derivation rules. In fact, in this approach JavaSpaces programs become expressions in a special purpose process algebra. Those authors aim at general results, i.e. comparison with other coordination languages, expressiveness, and results on serializability of transactions. Verification of individual JavaSpaces programs wasn't aimed at.

Although we also take an operational approach, our technique is quite different. We model the Javaspace system, and the JavaSpaces programs as expressions in the well-known, general-purpose process algebra, μCRL [13]. This allows us to use the existing μCRL tool set [2] and the CADP tool set [10] for the verification of individual JavaSpaces programs. In our model, the JavaSpaces programs communicate with the JavaSpaces system synchronously.

Our technical approach is similar to the research in [9, 14]. In these papers, programs written under the Splice architecture [3] are verified. Both papers give an operational model of Splice in μCRL, and use the μCRL and CADP tool sets to analyse Splice programs. One of the main purposes of the Splice architecture is to have a fast data distribution of volatile data. To this end, the data storage is distributed, as opposed to the central storage in JavaSpaces. In Splice, data items are distributed by a publish/subscribe mechanism. Newer data items simply overwrite outdated items.

7 Conclusion

In this paper we studied the specification of the notify mechanism of the JavaSpaces architecture. We have found several difficulties in interpretation that we tried to solve. Some of these problems are solved in the implementation of JavaSpaces but not in its specification.

First, the specification says that an event can be duplicated by the event source. This issue is source of several questions: How and when does the space decide to send twice the same message? Has the *notify* call a timeout? Can listeners be notified during a notification? Other problem comes from the interpretation of "best effort"; the space will "eventually" send a event after a

write. But when does the space send a event? and when does it compress several matches in one notification?.

The *notify* call is "synchronous", so the space blocks until the end of the remote method. Which actions are listeners allowed to do in the *notify* method? What will happen if a listener never returns? What will be the difference between a single-threaded and a multi-threaded space?

We attempted to solve the unclear details by making assumptions about the behavior of the system. Our informations are not only based on the JavaSpaces specification, sometimes ambiguous, but also in the archives of the discussion group where some of these have been treated. See, for example:

`http://archives.java.sun.com/cgi-bin/wa?A2=ind9904&L=javaspaces-users&P=R3468&D=0&H=0&O=T&T=1`

`http://archives.java.sun.com/cgi-bin/wa?A2=ind0106&L=javaspaces-users&P=R2562&D=0&H=0&O=T&T=1`

The last part of the paper is dedicated to the study of a very simple JavaSpace applications. Although we cannot verify the correctness of the proposed model, we can see, in small examples, that the behavior corresponds to JavaSpaces specification. Together with the μCRL simulator this provides some validation of the model. We also present some ideas of how to verify properties of applications. In the same way we can study more complex problems.

The μCRL model of the notification mechanism may be used not only to model check JavaSpaces applications but also to study the architecture itself and resolve all kinds of unclear or ambiguous points.

References

[1] Jan A. Bergstra and Jan Willem Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 37:77–121, 1985.

[2] S.C.C. Blom, W.J. Fokkink, J.F. Groote, I.A. Langevelde, B. Lisser, and J.C. van de Pol. μCRL: a toolset for analysing algebraic specifications. In *Proc. of CAV*, LNCS 2102, pages 250–254. Springer, 2001.

[3] M. Boasson. Control systems software. *IEEE Trans. on Automatic Control*, 38(7):1094–1106, July 1993.

[4] M.M. Bonsangue, J.N. Kok, and G. Zavattaro. Comparing coordination models based on shared distributed replicated data. In *Proc. of SAC*, pages 146–155. ACM, 1999.

[5] N. Busi, R. Gorrieri, and G. Zavattaro. Process calculi for coordination: From Linda to JavaSpaces. In *Proc. of AMAST*, LNCS 1816, pages 198–212. Springer, 2000.

[6] N. Busi and G. Zavattaro. On the serializability of transactions in JavaSpaces. In U. Montanari and V. Sassone, editors, *Electronic Notes in Theoretical Computer Science*, volume 54. Elsevier Science Publishers, 2001.

[7] Nadia Busi and Gianluigi Zavattaro. Publish/subscribe v.s. shared dataspace coordination infrastructures. In *Workshop on Web-based Infrastructures and Coordination Architectures for Collaborative Enterprises (WETICE-2001)*. IEEE Computer Society Press, 2001.

[8] N. Carriero and D. Gelernter. *How to Write Parallel Programs: A First Course*. MIT Press, 1990.

[9] P.F.G. Dechering and I.A. van Langevelde. The verification of coordination. In *Proc. of COORDINATION*, LNCS 1906, pages 335–340. Springer, 2000.

[10] J.-C. Fernandez, H. Garavel, A. Kerbrat, L. Mounier, R. Mateescu, and M. Sighireanu. CADP – a protocol validation and verification toolbox. In *Proc. of CAV*, LNCS 1102, pages 437–440. Springer, 1996.

[11] W. Fokkink. *Introduction to Process Algebra.* Texts in Theoretical Computer Science. Springer, 2000.

[12] E. Freeman, S. Hupfer, and K. Arnold. *JavaSpaces principles, patterns, and practice.* Addison-Wesley, Reading, MA, USA, 1999.

[13] J.F. Groote and M.A. Reniers. Algebraic process verification. In J.A. Bergstra et al., editor, *Handbook of Process Algebra*, chapter 17. Elsevier, 2001.

[14] J.M.M. Hooman and J.C. van de Pol. Formal verification of replication on a distributed data space architecture. In *Proceedings ACM SAC, Coordination Models, Languages and Applications*, page (to appear), Madrid, 2002. ACM press.

[15] R. Mateescu. *Verification des proprietes temporelles des programmes paralleles.* PhD thesis, Institut National Polytechnique de Grenoble, 1998.

[16] SUN Microsystems. *JavaSpaces$^{\text{tm}}$ Service Specification*, 1.1 edition, October 2000. See http://java.sun.com/products/javaspaces/.

[17] SUN Microsystems. *Jini$^{\text{tm}}$ Technology Core Platform Specification*, 1.1 edition, October 2000. See http://www.sun.com/jini/specs/.

[18] J.C. van de Pol and M. Valero Espada. Formal specification of JavaSpaces$^{\text{TM}}$ architecture using μcrl. In *Proc. of COORDINATION*, page (to appear). Springer, 2002.