Formal Specification of JavaSpaces^{TM} Architecture Using μCRL*

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. We study a formal specification of the shared data space architecture, JavaSpaces. This Java technology provides a virtual space for entities, like clients and servers, to communicate by sharing objects. We use μCRL, a language that combines abstract data types with process algebra, to model an abstraction of this coordination architecture. Besides the basic primitives write, read and take, our model captures transactions and leasing. The main purpose of the proposed formalism is to allow the verification of distributed applications built under the JavaSpaces model. A simple case study is analyzed and automatically model checked using the μCRL and CADP tool sets.

1 Introduction

It is well known that the design of reliable distributed systems can be an extremely arduous task. The parallel composition of processes with a simple behavior can even produce a wildly complicated system. A distributed application has to face some important challenges: it has to facilitate communication and synchronization between processes across heterogeneous networks, dealing with latencies, partial failures and system incompatibilities. The use of coordination architectures is a suitable way to manage the complexity of specifying and programming large distributed applications.

Re-usability is one of the most important issues of coordination architectures. Once the architecture has been implemented on a distributed network, different applications can be built according to the requirements without any extra adaptation. Programmers implement their systems using the interface provided by the architecture, which consists of a set of primitives or operators.

In this paper we study the JavaSpaces^{TM} [14] technology that is a Sun Microsystems, Inc. architecture based on the Linda coordination language [6]. JavaSpaces is a Jini^{TM} [15] service that provides a platform for designing distributed computing systems. It gives support to the communication and synchronization of external processes by setting up a common shared space. JavaSpaces

* 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.

is both an application program interface (API) and a distributed programming model. The coordination of applications built under this technology is modeled as a flow of objects. The communication is different from traditional paradigms based on message passing or method invocation models. Several remote processes can interact simultaneously with the shared repository, the space handles the details of concurrent access. The interface provided by JavaSpaces is essentially composed by insertion and lookup primitives. In the following section we present some details of the technology specification.

The goal of our research is to verify the correctness of applications built using JavaSpaces services. Therefore we propose a formal model of the architecture which will allow to prototype these distributed applications. We use the language μCRL [11] to create an operational and algebraic definition of the technology. μCRL is a language based on the process algebra ACP [9], extended with equational abstract data types. Its tool set [1] combined with the CÆSAR ALDÉBARAN DEVELOPMENT PACKAGE (CADP) [8] allows the automatic analysis of finite systems.

This paper is structured as follows. After this introduction, we complete the description of JavaSpaces and we present the μCRL language. We continue with the study of a model of JavaSpaces in the formal language μCRL. Then we present a simple case study showing the main features of the proposed specification and the model checker. Before the conclusions, we provide pointers to some related work. The formal specification and some examples can be found at: "http://www.cwi.nl/~miguel/JavaSpaces/".

2 JavaSpaces

Components of applications built under the JavaSpaces model are "loosely coupled", they do not communicate with each other directly but by sharing information via the common repository. They execute primitives to exchange data with the shared space. Figure 1 presents an overview of the JavaSpaces architecture.

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 is returned, it is removed from the space.

JavaSpaces also provides support to distributed events, leasing and transactions, from the Jini architecture [15]:

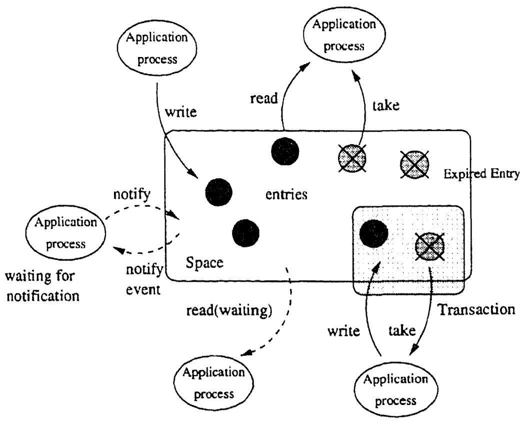

Fig. 1. JavaSpaces architecture overview

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 on 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.

The space manages some distributed events, in particular: a process can inform the space its interest in future incoming objects, by using the *notify* primitive. The space will notify by an event when a matching object arrives to the space. Notification will not be studied in this paper.

To know more about JavaSpaces, please consult the references [10,14].

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:$\rightarrow$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 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:

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

4 Formal Specification

The μCRL model we propose supports the main features of the JavaSpaces specification introduced in previous sections. The choices made on the implementation of the model try to keep it as compliant as possible with the specification. However, some concepts have been abstracted away trying to keep the model simple and suitable to do model checking.

First we present the architecture from the application point of view focusing on the API, going later into specific details of the implementation.

4.1 Application Point of View

The space is modeled as a single process called *javaspace* (we have not experimented with using multiple spaces). User applications are implemented as external processes executed in parallel with the space. External applications exchange data between them by transferring entries through the shared space. The communication between the *javaspace* process and the external applications is done by means of a set of synchronous actions, derived from the JavaSpaces API. A JavaSpaces system is specified in μCRL as follows:

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

The arguments of the *javaspace* process represent the current state of the space. They are composed by: stored objects, active transactions, the current time, et cetera... These arguments are explained in detail in the following section.

External processes have unique identification number. They have to add it as parameter to every invocation of a primitive. The space uses this *id* 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^{TM} 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. Data fields, from standard sorts (naturals, booleans,...) or new sorts, and operators can be included. The sort must include the equality (*eq*) function, and the constructor *entryNull* because they are necessary to perform the look up actions. The following code presents the definition of a simple counter:

```
sort  Entry
func  entryNull:→Entry
      counter: Nat→Entry
map   eq: Entry×Entry→Bool
      value:Entry→Nat
      inc:Entry→Entry
var   n,n': Nat
rew   eq(entryNull, entryNull) = T
      eq(counter(n), counter(n')) = eq(n, n')
      eq(entryNull, counter(n)) = F
      eq(counter(n), entryNull) = F
      value(counter(n)) = n
      inc(counter(n)) = counter(S(n))
```

The insertion of an entry into the space is done by means of the *write* action. This primitive is defined as follows:

```
sort  Nat, Entry
act   write: Nat×Entry×Nat×Nat
```

The arguments of the action *write* are: the process identification number, the entry, the transaction identifier and the requested lease. The behavior of the action depends on whether it is executed under a transaction or not. If it is not joined to any transaction, the *transaction id* parameter is equal to 0 or *NULL*, then the insertion is instantaneously updated in the space. In our model there are no possible exceptions thrown during the operation. It means that when a *write* has been executed the entry is successfully inserted. Different *write* invocations will place different objects in the repository, even if the data values are equal. The use of transactions is explained further in the present section.

When a user performs a *write*, he can associate a lease to the entry. An entry is automatically removed from the space when its lease expires. A lease is a natural number from 0 to *FOREVER*. The null value (0) means that the entry is deleted at the same unit of time that it is placed in the space. The *FOREVER* value says the entry will never be removed. Our model differs from the JavaSpaces specification because the lease requested is always granted by the space and it cannot be canceled or renewed.

An example of *write* invocation in which the application process inserts a null counter in the space without transaction and leased for one time tick, is defined as follows:

```
proc  p(id:Nat) =
      ...
      .write(id, counter(0), NULL, S(0))
      ...
```

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 don't 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.

Let's see an example of Query **sort** that has two possible queries: *any* that will match any entry in the space and *equal* that match any entry with a data field, accessed via the operator *value*, is equal to a given parameter:

```
sort  Query
func  any: →Query
      equal: Nat→Query
map   test: Query×Entry→Bool
var   e: Entry
      n: Nat
rew   test(any, e) = T
      test(equal(n), e) = eq(n, value(e))
```

An entry of the space will match a look up action if it satisfies the associated query, as indicated by the *test* predicate.

There are implemented four look up primitives: *read*, *take*, *readIfExists* and *takeIfExists*. All of them take the following arguments: process identification number, transaction identification number, timeout and query.

The execution of a look up primitive is done by means of two atomic actions. First the external process invokes the primitive (*read*, *take*, ...), then the space communicates the result of the request by returning a matching entry if the operation was successfully performed or an *entryNull* if the timeout has expired and no objects satisfied the query.

The μCRL specification of the actions is:

```
sort   Nat, Entry, Query
act    read, take, readIfExists,takeIfExists: Nat×Nat×Nat×Query
       Return:Nat×Entry
```

Let's see an example program using the *take* operation, which request any entry of the space and blocks for one time step:

```
proc   p(id:Nat) =
       ...
       .take(id, NULL, S(0), any)
       .(Σ_{e:Entry} Return(id, e)
          .(...◁ not(eq(e, entryNull)) ▷...))
       ...
```

The behavior of the four primitives depends on how the space is updated after the action (whether the entry is removed or not), and whether the action performs a test of presence (if there are matching objects with conflicting locks). The behavior is different if the actions are executed under a transaction.

In our model the instantiation of a transaction is done by the action *create*, which has the arguments: process identification number, transaction identification number (assigned by the space), and lease. The space allocates a new resource and returns to the user the identification number of the created transaction. Once the transaction has been created, operations join to it by passing its *id* number to the primitives.

A transaction can complete by the explicit actions *commit* and *abort*, or by being automatically aborted when its lease expires. If the last case happens the space informs to the creator of the transaction the expiration of the transaction by "sending an exception". We model the exceptions by a communication action called *Exception* parameterized with the *id* of the transaction and the *id* of the process to whom the exception is directed to. If a process creates a transaction it has to add the possibility of receiving an exception on all the actions executed until the commitment of the transaction.

In our model, we restrict to the case that a transaction can only be used by a single process. Only the creator can join primitives and receives the timeout exception. The following example shows how the transaction model can be used in external processes:

```
act    create: Nat×Nat×Nat
       commit, abort, Exception:Nat×Nat
proc   p(id:Nat) =
       ...
       $\sum_{trc:Nat}$ (create(id, trc, S(0))
         .(write(id, ..., trc, ...) + Exception(id, trc).handle_actions)
         .(take(id, ..., trc, ...) + Exception(id, trc).handle_actions)
         .(commit(id,trc) + Exception(id, trc).handle_actions))
           or
         .(abort(id, trc) + Exception(id, trc).handle_actions))
       ...
```

The Jini's transaction model has been simplified, for example our model doesn't support nested transactions or transactions over multiple processes.

Transactions make changes on the semantics of the primitives, e.g. when a *write* action is performed under a transaction, the entry will be externally visible only after the transaction commits, if the transaction is aborted no changes will be updated in the space. If a process inserts an entry under a transaction, and meanwhile another process executes a *readIfExists*, the second process blocks waiting for the commitment of the transaction (or for the timeout), if the entry is the only in the space that satisfies the query.

We have introduced the main features of the specification. Although all the JavaSpaces services have not been implemented, the proposed framework is suitable to model and verify many interesting JavaSpaces applications. Now let's present some details about the space implementation.

4.2 Implementation Point of View

The *javaspace* process handles the concurrent access of the external applications to the common repository. It manages a data base storing the shared entries, the active transactions, and other data structures like pending actions. The process has also to manage some timeouts of leased resources.

To support leasing and the timeouts, the space has to deal with the notion of "time". We propose the implementation of a centralized clock. This is appropriate, because in reality a Javaspace server has a centralized implementation, in multiple JavaSpaces each space would have its own clock. There is no clock synchronization between the space process and the external applications, so we have not made any assumption about the relative speed of the processes.

The clock is implemented by a discrete counter. More than one service can be processed in the same time unit. So between two units the *javaspace* process can treat several communication primitives. Externally we can say several actions are performed in parallel (in the sense of interleaving).

The *javaspace* process increments arbitrarily a counter that determines the duration of time from the startup of the system to the present state. The μCRL tool set can only analyze finite instances of the specification so we have to limit the time duration of the system. For this reason we use a constant which indicates after how many time steps the system must halt. The process will run from 0 to *FOREVER* clock ticks.

```
map FOREVER:→Nat
rew FOREVER = S(...S(0))
```

Now, we are going to analyze the most important issues of the specification of *javaspaces*.

In the first part of the specification we define a number of standard data types which will allow us to define more complex structures. We can find at the top of the specification the sorts: booleans (*Bool*) and naturals numbers (*Nat*) with their usual operations. 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 *Bool* specification has been presented in Section 3. Naturals have two constructors: *0* for the null value and *S(n)*, for the successor of a natural.

Entries are internally encapsulated in the *Object* sort, which includes the entry, the requested lease and an identification number. The space automatically assigns a fresh *id* to every new entry. The signature of the *Object* sort is defined as follows:

```
sort  Object
func  object: Nat×Entry×Nat→Object
map   eq: Object×Object→Bool
      id:Object→Nat
      entry:Object→Entry
      lease:Object→Nat
```

Objects are stored in a data base that has the structure of a set. The *javaspace* process manages this data base by inserting, removing and looking up entries. The entries are organized without any order, so when the space executes a search action, all of the matching entries have the same possibility to be selected.

The data base is defined by the *ObjectSet* sort. It has two constructors the *emO* that creates a new empty set and *in* that inserts an object in a set. It has other operators to locate entries, remove, compare et cetera...

When the space receives an external invocation of a *write*, it creates a new *object* and it inserts it in the set. The following fragment of code corresponds to a *write* action without transaction.

```
proc  javaspace(t:Nat, M:ObjectSet,..., objectsIDs:Nat,...) =
        ...
        +
        $\sum_{processID:Nat}( \sum_{e:Entry}( \sum_{trcID:Nat}( \sum_{lease:Nat}($
         Write(processID, e, trcID, lease)
         .javaspace(t, in(object(objectIDs, e, plus(lease,t)),M),.., S(objectIDs),...)
        $\lhd$
         and(eq(trcID, NULL),...)
        $\rhd \delta$))))
        + ...
```

In the code, M represents the object set, t is the current time, and *objectIDs* is a counter used to assign a fresh *id* to every new entry.

Regarding the look up primitives: when the space receives a search request first it creates a *pending action*. A *pending action* includes: the *id* of the process that executed the primitive, the type (*read*, *take*, ...), the transaction *id*, the time the process wants to block and the query. The *pending actions* are stored in a set of the sort *ActionSet* whose definition is similar to the object set's one.

If there is an object that matches one of the pending actions then the space returns the entry to the corresponding external process by means of the *return* action. An entry matches an action if the execution of the *test* operation of the associated query returns true. If there is a pending action with an expired timeout the space returns the *entryNull* Figure 2 shows how this mechanism works.

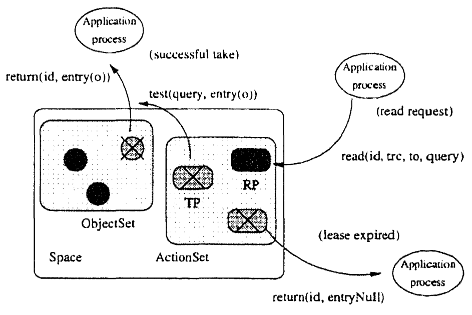

Fig. 2. Look up mechanism

We can see below the sort that defines a transaction:

sort `Transaction`
func transaction: `Nat`×`Nat`×`ObjectSet`×`ObjectSet`×`ObjectSet`→`Transaction`
map eq: `Transaction`×`Transaction`→`Bool`
`id`:`Transaction`→`Nat`
`timeout`:`Transaction`→`Nat`
`Wset, Rset, Tset`:`Transaction`→`ObjectSet`

Every new transaction receives a fresh identification number, 0 is reserved for the *NULL* transaction. Transactions have three object sets. The sets are used to trace the changes performed by the operations joined to the transaction:

- *Wset*: stores the entries written under a transaction. After a commit the objects are placed in the space set.
- *Tset*: after a *take* the object is removed from the space and is placed in the transaction set. If the transaction commits the *Tset* is deleted, if it aborts the objects are put back in the space.

- *Rset*: stores the entries read under the transaction. When an object is in a *Rset* it cannot be taken outside the transaction.

When a process executes a *readIfExists* or *takeIfExists* and there is no matching object in the space, we check in the *Wsets* and *Tsets* of the other transactions to decide if the process has to block or not. When the lease of a transaction expires the space aborts it and informs the user by executing the action *Exception*.

In summary, the *javaspace* process can ever:

- Receive request of services: look ups or insertions.
- Match entries with pending actions, sending the result to the external processes.
- Perform actions related to the lease or timeout expirations: to remove old entries, abort expired transactions, unblock process waiting for an entry.
- Increment the clock by one unit until the time limit, leaving unchanged the state of the system. This action is only possible if there are no matched actions or expired timeouts (or leases) in the system.

5 Verification

We are going to formalize a simple JavaSpaces application to show the possibilities of the μCRL tool set for system verification. The system is inspired by the classical arcade game Ping-Pong, in which two players throw one ball from one to the other. This example has been taken from the chapter 5 of the book "JavaSpaces^{TM} Principles, Patterns, and Practice" [10]. The players are modeled by two processes called Ping and Pong which communicate by means of an entry that encapsulates the ball. In the first section we propose a very simple version of the game, in the second we did some small changes to the game rules, that allow us to use more functionality of the specification.

5.1 Simple Ping-Pong

In this version, players can only catch and throw the ball. The system halts when players have sent the ball a fixed number of rows or when the space life time expires.

The Entry **sort** (ball) is defined as follows:

```
sort  Entry
func  entryNull:→Entry
      ball: Name→Entry
map   eq: Entry×Entry→Bool
      receiver:Entry→Name
var   e: Entry
      n,n': Name
rew   eq(...,...)
      receiver(ball(n)) = n
```

The only field the entry has is the name of the player whom the ball is directed to. The name is from the sort *Name*, that has two constructors *Ping* and *Pong*, and one function (*other*) used to switch from one to the other (*other(Ping) = Pong*). To get the ball from the space, a player uses a query:

```
sort  Query
func  forMe: Name→Query
map   test: Query×Entry→Bool
var   e: Entry
      n,n': Name
rew   test(forMe(n), e) = eq(n, receiver(e))
```

The code of both players is the same. It has as arguments: the given name, the identification number, and the number of player rows:

```
proc  player(id:Nat,name:Name,round:Nat) =
        take(id, NULL, FOREVER, forMe(name))
        .Σe:Entry (Return(id,e)
           .print(name)
           .write(id, ball(other(name)), NULL, FOREVER))
        .player(id,name,S(round))
        ◁ lt(round, maxRounds) ▷ δ
```

Print is an external action used to communicate to the environment that a player has catched the ball and is going to throw it back. In the initial state the space includes a ball directed to *Ping*. The values of the other main data structures (TransactionSet, PendingActionSet,. . .) are initialized to empty. The system instantiation is as follows:

$$
\begin{aligned}
System = {} & \tau_{\{W,E,Rt\}}\partial_{\{write,Write,take,Take,return,Return\}} \\
& (javaspace(0, in(object(0, ball(Ping), FOREVER), emO), \\
& emT, emA, S(NULL), S(0), 0) \\
& ||\, player(0, Ping, 0)\, ||\, player(S(0), Pong, 0))
\end{aligned}
$$

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 a two rows game reduced by tau equivalence.

The fair execution of the game is 0-3-1-2-4. If the time reaches the bound the system halts, and it's possible that the system stops before all the rows have been completed, this behavior corresponds to the transitions 0-4, 3-4 and 1-4.

Some properties can be automatically verified by the Evaluator tool from the CADP package. These properties are expressed in temporal logic. We used the regular alternation-free μ-calculus formulas [13]. For example, a safety property expresses the prohibition of "bad" execution sequences. The following formula means that the player *Ping* cannot throw the ball twice in a row:

[true* . *"print(Ping)"* . (not *"print(Pong)"*)* . *"print(Ping)"*] false

The tool set verifies if the formula holds or not. In the same way we can verify invariants, liveness or fairness properties et cetera...

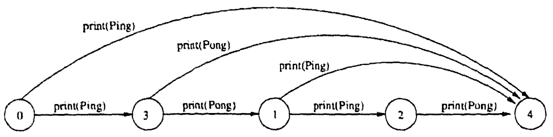

Fig. 3. External behavior of 2 rows simple Ping-Pong game

5.2 More Complex Ping-Pong

We introduce a small change in the rules of the game. In this version, once a player catched the ball, he has one time unit to put it back in the space, otherwise he looses the game. We model this approach by using transactions. After a player has performed the *take*, he creates a transaction leased for one second. Once the *write* operation is done, the transaction can commit. Let's see the process code:

```
proc  player(id:Nat,name:Name,round:Nat) =
        take(id, NULL, FOREVER, forMe(name))
        .$\sum_{e:Entry}$ (Return(id,e)
           .$\sum_{trc:Nat}$ (create(id, trc, S(0))
             .(write(id, ball(other(name)), trc, FOREVER)
               + Exception(id, trc).looser(name))
             .(print(name) + Exception(id, trc).looser(name))
             .(commit(id, trc)) + Exception(id, trc).looser(name)))
        .player(id,name,S(round))
        $\lhd$ lt(round, maxRounds) $\rhd$ $\delta$

proc  looser(name:Name) = I_am_the_looser(name).$\delta$
```

Figure 4 shows the reduced LTS associated to the system. In the state number 6 the game has finished. The system can halt due to the end of the match or because one of the players has lost or the time has expired.

From the state 0 there are 5 possible transitions:

- *print*(*Ping*) to the state 3. The player Ping has taken the ball and the transaction expires. Ping looses.
- *print*(*Ping*) to the state 6. The player displays the message and the system halts (because $t = FOREVER$). Nobody looses.
- *I_am_the_looser*(*Ping*) to the state 6. The player takes the ball but the transaction expires before printing. *Ping* looses.

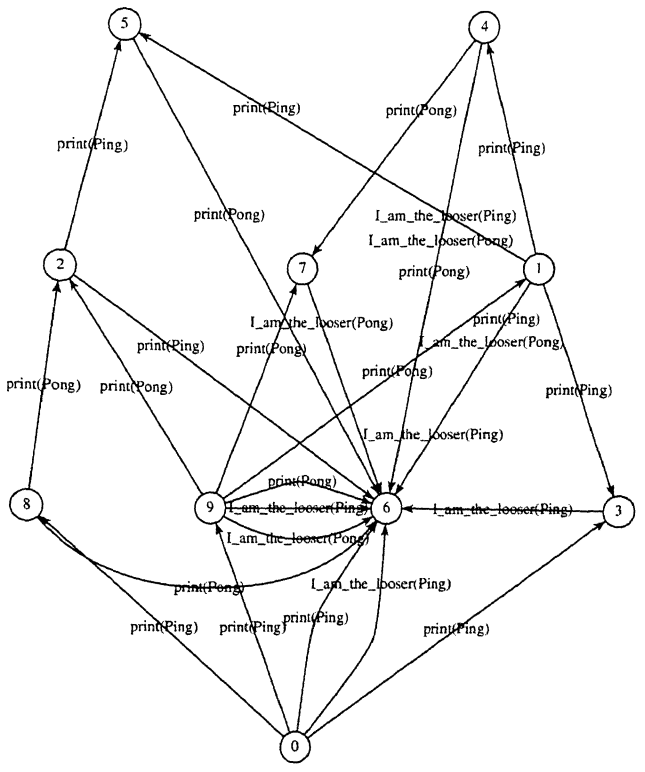

Fig. 4. External behavior of 2 rows complex Ping-Pong game

- *print(Ping)* to the state 9. Normal execution, player has catched and thrown the ball. The fair game execution path is 0-9-1-4-6.
- *print(Ping)* to the state 8. The player throws the ball correctly but in the last time step of the system so the transaction can not expire anymore. In the states 8, 2 and 5 the transactions timeouts are always greater than *FOREVER*. Following this path nobody can loose.

We can see that the property *"A player has one unit of time to throw the ball after he catched it"* doesn't hold, because after the return of the *take* action the player can wait as long as he wants before creating the transaction. We prove this by showing that the following formula doesn't hold for the system (without hiding internal actions):

[true* . *'T.*'*.(not (*'E.*'* or *'print.*'*)*) . *"clock"* . (not (*'E.*'* or *'print.*'*)*) . *"clock"*] false

The formula says that after a take communication (*T*), we can not have two consecutive clock ticks without an intermediate exception (*E*) or a *print*. The model checker gives a path that doesn't satisfy the formula. Next, we rewrite the property in the following way: *"A player has one unit of time to throw the ball after he created a transaction, otherwise he receives an exception"*. This is expressed by the formula:

[true* . 'C(.*)'.(not ('E.*' or 'Cm.*')*) . "clock" . (not ('E.*' or 'Cm.*')*) . "clock"] false

After the creation of the transaction (*C*), players have to commit (*Cm*) in one time step otherwise they get an exception. The system indeed satisfies this formula.

6 Related Work

Our information on JavaSpaces is based upon the book [10], and the documentation from Sun on JavaSpaces [14] and Jini [15]. 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 [6].

Some work on the formalization of JavaSpaces (or other Linda-like languages) exist, notably [3,4,5]. 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 [11]. This allows us to use the existing μCRL tool set [1] and the CADP tool set [8] 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 [7,12]. In these papers, programs written under the Splice architecture [2] 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 provide a framework to verify distributed applications built using the JavaSpaces architecture. We have modeled in μCRL a formal specification of the main features of the coordination architecture that allow to prototype and analyse JavaSpaces applications. The language μCRL is expressive enough ıupport all the functionality of JavaSpaces. The main features have been im-mented: primitives, leases, timeouts, transactions and events. We foresee no jor problems in the specification of the remaining services.

The last part of the paper is dedicated to the study of a very simple JavaS-:es application. Although we cannot verify the correctness of the proposed ·cification, we can see in small examples, that the behavior corresponds to the raSpaces specification. Together with the μCRL simulator this provides some idation of the model. We also present some ideas of how to verify properties applications. In the same way of the example we can study more complex ›blems.

There are many possibilities for future work. First we can extend the specifi-;ion by including the remaining features: notify primitive, lease renewal, nested ınsactions, multiple spaces, etc. We can also analyse properties of the formal ecification or the transactional model, do formal comparision with other ap-›aches [4,5], or to go further in the verification of applications by studying real ırld applications.

eferences

1] 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.

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

3] 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.

4] 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.

5] 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.

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

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

[8] 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.

[9] W. J. Fokkink. *Introduction to Process Algebra.* Texts in Theoretical Computer Science. Springer-Verlag, 2000.

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

11] 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.

12] 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.

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

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

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