Formal specification of JavaSpaces$^{\text{TM}}$ architecture using μCRL

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

2000 Mathematics Subject Classification: 68M14, 68N30, 68Q85
Keywords and Phrases: JavaSpaces, shared data space, distributed systems, coordination, formal methods, μCRL
Note: Research carried out for the STW-project CES.5009: "Formal Design, Tooling, and Prototype implementation of a Real-Time Distributed Shared Data Space"

1. Introduction

It is well known that the design of reliable distributed system 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$^{\text{TM}}$ [13] technology that is a Sun Microsystems, Inc. architecture based on the Linda coordination language [6]. JavaSpaces is a Jini$^{\text{TM}}$ [14] 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 is both an application program interface (API) and a distributed programming model. Protocols built under the technology are modeled as a flow of objects. The communication is different from traditional 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 the JavaSpaces in μCRL formal model. 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.

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. These primitives are defined on the JavaSpaces API:

```
public interface JavaSpace {
   Lease write(Entry entry, Transaction txn, long lease)
      throws ...;
   long NO_WAIT = 0;
   Entry read(Entry tmpl, Transaction txn, long timeout)
      throws ...;
   Entry readIfExists(Entry tmpl, Transaction txn, long timeout)
      throws ...;
   Entry take(Entry tmpl, Transaction txn, long timeout)
      throws ...;
   Entry takeIfExists(Entry tmpl, Transaction txn, long timeout)
      throws ...;
   EventRegistration notify(Entry tmpl, Transaction txn,
         RemoteEventListener listener, long lease,
         MarshalledObject handback)
      throws ...;
}
```

A *write* operation places a copy of an entry passed as argument into the space. A *read* request returns a copy of an object from the space that matches the *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 possible 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.

A process that executes a *notify* primitive informs the space its interests in future incoming objects. The space will notify by an event when a matching object arrives. Notification will not be studied in this paper. To know more about JavaSpaces, please consult the references [10, 13].

Objects in the space are located by "associative lookup". Processes find the objects they are interested in by expressing constraints about their contents without having any information about the object identification, owner or location. The space also provides a reliable persistent object storage, written objects remain in the repository until an explicit remove action, even if the writer process has terminated. Picture 1 presents an overview of the JavaSpaces architecture:

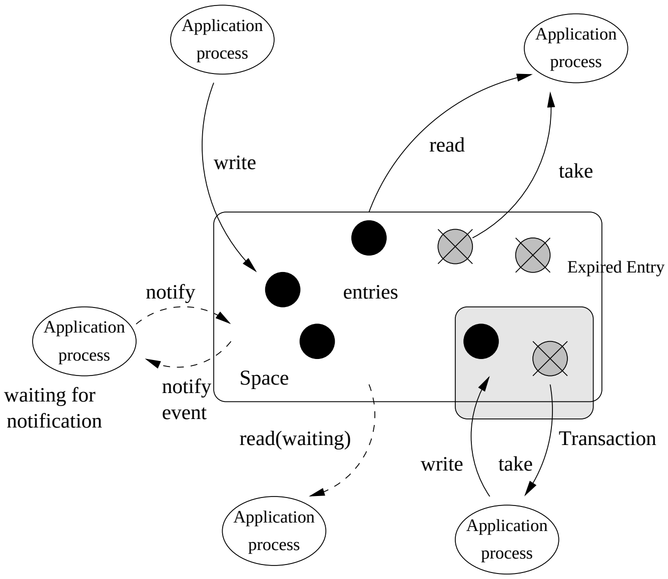

Figure 1: JavaSpaces architecture overview

JavaSpaces also provides support to distributed events, leasing and transactions, from the Jini architecture [14]:

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 be never visible if the transactions aborts. Transactions provide a means for enforcing consistency. Transactions on JavaSpaces preserve the ACID properties: Atomicity, Consistency, Isolation and Durability.

The space can determine the time during which an object can be stored in the repository before being automatically removed. 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 manages some distributed events, for example: an exception is sent when the lease of a transaction has been expired, a notification event communicates that a entry has arrived to a process that has executed a notify operation.

Before presenting the formal model, we are going to introduce the μCRL language.

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:$\to$`Bool`
map `and`: `Bool`$\times$`Bool`$\to$`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**) 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 parametrized. Actions in μCRL are considered atomic. There are two predefined actions: 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 **encap** (∂_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

Some fragments of code are included during the introduction of the specification, in the following section:

4. Formal Specification

The μCRL model we propose supports the main features of the JavaSpaces specification introduced in previous sections. We will abstract the concepts trying to be as compliant as possible with the specification, although not all of the primitives and services have been implemented. The remaining services will be studied in further work.

First we present the architecture from the application point of view, going later into specific details of the model.

4.1 Application point of view

The space is modeled as a single process called *javaspace*. 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 abstracted from the JavaSpaces API. A JavaSpaces system is specified in μCRL as follows:

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, 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. On 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 only restriction is that the sort must include the equality (*eq*) function, because it is necessary in order to perform the look up operations. The following code presents an example of definition:

```
sort  Entry
func  counter: Nat→Entry
map   eq: Entry×Entry→Bool
      value:Entry→Nat
      inc:Entry→Entry
var   e,e': Entry
      n: Nat
rew   eq(e, e') = eq(value(e), value(e'))
      value(counter(n)) = n
      inc(counter(n)) = counter(S(n))
```

This *Entry* sort represents a counter. It has only one natural field and two operators, besides the equality function, one to access to the value and another to increment it with one unit.

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
      % Arguments:
        % process identification number
        % entry
        % transaction identification number
        % 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 for 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   eq: Query×Query→Bool
      test: Query×Entry→Bool
var   e: Entry
      n,n': Nat
rew   test(any, e) = T
      test(equal(n), e) = eq(n, value(e))
      % eq operators...
```

An entry of the space will match a look up action if it satisfies the associated query.

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 query using one of the following two actions:

- *nonNull*: The look up operation was successfully performed and the space returns a matching entry.
- *null*: It's sent when there are no objects in the space that satisfy the query and the timeout expires.

The μCRL specification of the actions is:

```
sort  Nat, Entry, Query
act   read, take, readIfExists, takeIfExists: Nat×Nat×Nat×Query
      % Arguments:
        % process identification number
        % transaction identification number
        % timeout
        % query
act   nonNull:Nat×Entry
      % Arguments:
        % process identification number
        % entry
```

act null:Nat
% Arguments:
% process identification number

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)
.($\sum_{e:Entry}$ (nonNull(id, e)) + null(id))
...

The behavior of the four primitives depends on how the space is updated after the action (whether the entry is removed or not), and if the action performs a test of presence (if there are 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, a user can join operations 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 external process the expiration of the transaction by "sending an exception". We express this event mechanism creating a communication action called *Exception* parametrized with the *id* of the transaction. 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, a transaction can only be used by a single process, only the process that creates the transaction can join primitives or can receive the timeout exception. The following example shows how the transaction model can be used in external processes:

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

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

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. Not all of the JavaSpaces services have been implemented, but the other features can be modeled in a similar way. The framework we have presented shows that we can model JavaSpaces applications. Now let's present some details about the space implementation.

4.2 System 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 distributed events.

To support leasing and the timeouts, the space has to deal with the notion of "time". We propose the implementation of a centralized logical clock. 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))

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).

There is no clock synchronization between the space process and the external applications, so we cannot make any assumption about the relative speed of the processes.

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 the 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
```

The *Object* constructor has three arguments: identification number, entry and lease. The definition also includes functions to access the data and the obligatory equality operator.

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. The other operators are: equality function (*eq*), "if-then-else" auxiliary function (*if*), remove a given object (*rem*), union of two sets (U), check if the set is empty (*empty*), and some functions to locate entries:

- *test*: check if a given object has been inserted into the space.
- *testExpired*: check if a object is in the space and if its lease has expired.
- *existsExpired*: check if there are any expired objects in the set.

- *existsSatandNotExp*: check if there are any no expired object in the space that satisfy a given query.

The signature of this sort is:

```
sort  ObjectSet
func  emO:→ObjectSet
      in:Object×ObjectSet→ObjectSet
map   eq: ObjectSet×ObjectSet→Bool
      if: Bool×ObjectSet×ObjectSet→ObjectSet
      rem: Object×ObjectSet→ObjectSet
      test: Object×ObjectSet→Bool
      empty: ObjectSet→Bool
      U: ObjectSet×ObjectSet→ObjectSet
      testExpired: Object×ObjectSet×Nat→Bool
      existsExpired: ObjectSet×Nat→Bool
      existsSatandNotExp: ObjectSet×Nat×Query→Bool
```

When the space recives 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),...)
$\triangleleft$
% Action under NULL transaction
% and there are still fresh ids
$\triangleright\ \delta))))$
+
...

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. To keep finite the instance of the space we allow the use of a limited number of entries defined by a constant:

map maxObjects:→Nat
rew maxObjects = S(...S(0))

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 transaction *id*, the time the process wants to block, and the query.

The *pending actions* are stored in some sets of the sort *ActionSet*. We have one set for each class of look up primitives: *readPendings*, *takePending*, *readIfExistsPendings*, *takeIfExistsPendings*. The definition of these sets is similar to the object set.

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 *nonNull* 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 executes a *null* action. Picture 2 shows how this mechanism works.

We can see below the sort that defines a transaction:

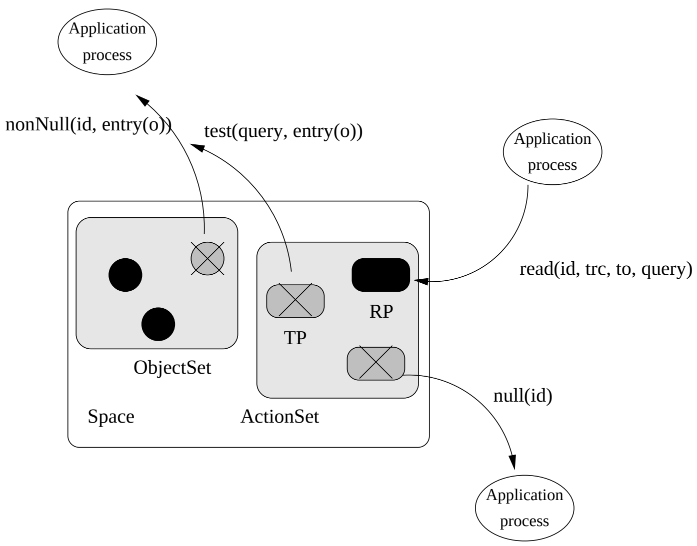

Figure 2: Look up mechanism

sort `Transaction`
func transaction: `Nat`$\times$`Nat`$\times$`ObjectSet`$\times$`ObjectSet`$\times$`ObjectSet`$\rightarrow$`Transaction`
map eq: `Transaction`$\times$`Transaction`$\rightarrow$`Bool`
`id`:`Transaction`$\rightarrow$`Nat`
`timeout`:`Transaction`$\rightarrow$`Nat`
`Wset, Rset, Tset`:`Transaction`$\rightarrow$`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 transactions commit the $Tset$ is deleted, if it aborts the objects are put back in the space.
- *Rset*: stores the entries read under a transaction. When an object is in a *Rset* it cannot be taken outside the transaction.

When a process executes a *readIfExists* of *takeIfExists* and there are not 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 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 is there are no expired timeouts or leases in the system.

The following fragment of code shows an overview of the *javaspaces* process:

```
proc  javaspace(t:Nat, M:ObjectSet, Trc:TransactionSet, TP:ActionSet,
        RP:ActionSet, TEP:ActionSet, REP:ActionSet, trcIDs:Nat, objetIDs:Nat) =
      (
      %coordination primitives
      Write(...).javaspace(...) %inserts new object
      +
      Read(...).javaspace(...) %creates new pending action
      +
      NonNull(...).javaspace(...) %matches an entry with one pending action
      +
      ...
      +
      %process expired actions
      garbage_collection.javaspace(...)%removes an expired entry
      +
      Null(...).javaspace(...)%unblocks a process
      +
      exception(...).javaspace(...)%aborts a transaction; sends the corresponding exception
      +
      ...
      +
      %Increment clock
      clock_tick.javaspace(S(t), data0, data1, ...) ◁ there are no timeout actions to do ▷ δ
      )
      ◁ t ≤ FOREVER ▷ δ
```

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 through 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 time expires.

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

```
sort  Entry
func  entry: Name→Entry
map   eq: Entry×Entry→Bool
      receiver:Entry→Name
```

```
var   e,e': Entry
      n: Name
rew   eq(e,e') = and(eq(reveiver(e), receiver(e')))
      receiver(entry(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 2 constructors $Ping$ and $Pong$, and one funtion ($other$) 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   eq: Query×Query→Bool
      test: Query×Entry→Bool
var   e: Entry
      n,n': Name
rew   eq(forMe(n), forMe(n')) = eq(n, n')
      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))
  $\sum_{e:Entry}$ ((nonNull(id, e) + null(id).δ)
    .print(name)
    .write(id, entry(other(name)), NULL, FOREVER))
  .player(id,name,S(round))
  $\lhd$ lt(round, maxRounds) $\rhd$ δ

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

$$
\begin{aligned}
System = \tau_{\{W,E,nnu,nu\}}\partial_{\{write,Write,take,Take,nonNull,NonNull,null,Null\}}
&(javaspace(0, in(object(0, entry(Ping), FOREVER), emO),\\
&emT, emA, emA, emA, emA, S(0), S(NULL))\\
&\,||\, player(0, Ping, 0) \,||\, player(S(0), Pong, 0))
\end{aligned}
$$

To each μCRL specification belongs a directed graph, in which the states are process terms, and the edges are labelled with actions. If this transition system has a finite number of states the tool set can be used in combination with the CÆSAR ALDÉBARAN DEVELOPMENT PACKAGE (CADP) to generate, visualise and analyse this transacion system. The picture 3 shows the generated labelled transition system (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 reachs 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 trasitions 0-4, 3-4 and 1-4.

Formulas can be automatically verified by the CAPD tool set. We can use these formulas to prove properties of the system. 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 ...

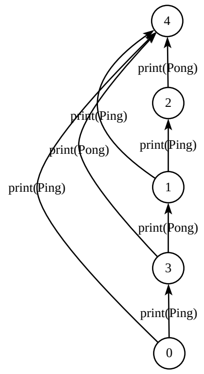

Figure 3: External behavior of 2 rows simple Ping-Pong game

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 reformed 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))
$\sum_{e:Entry}$ ((nonNull(id, e) + null(id).δ)
$\sum_{trc:Nat}$ (create(id, trc, S(0))
.(write(id, entry(other(name)), trc, FOREVER) + Exception(trc).looser(name))
.(print(name) + Exception(trc).looser(name))
.(commit(id, trc)) + Exception(trc).looser(name)))
.player(id,name,S(round))
$\lhd$ lt(round, maxRounds) $\rhd$ δ

proc looser(name:Name) = I_am_the_looser(name).δ

The 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.
- $print(Ping)$ to the state 9. Normal execution, player has catched and thrown the ball. The fair game execution path is 0-9-1-4-6.

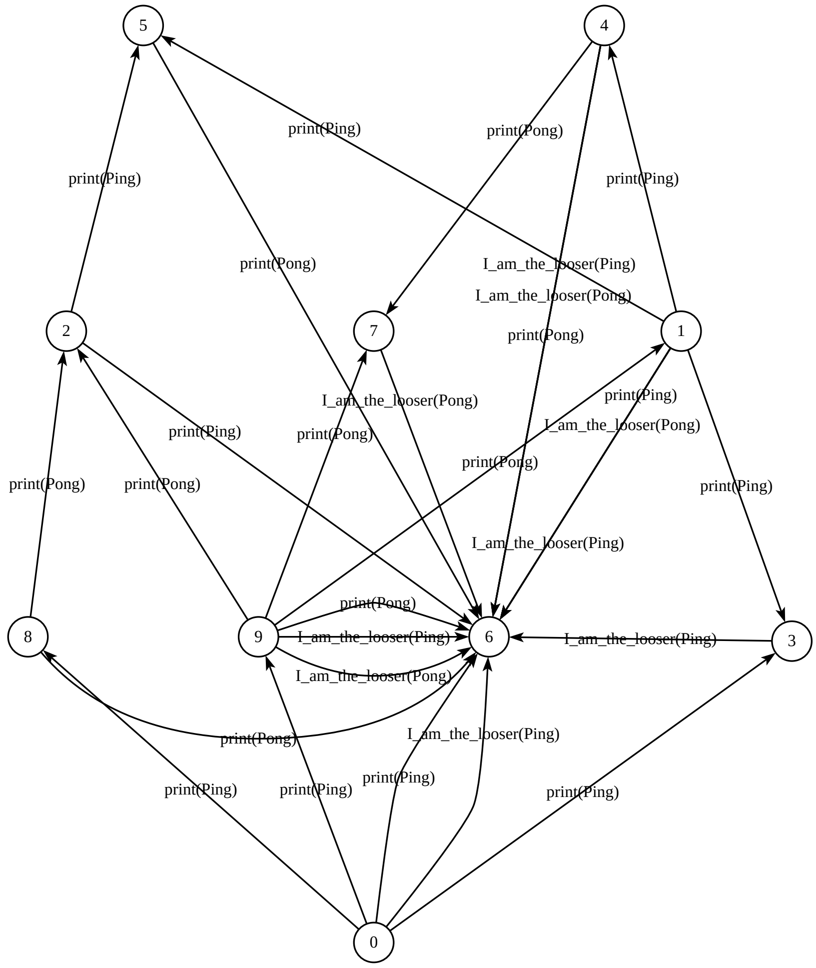

Figure 4: External behavior of 2 rows complex Ping-Pong game

- $print(Ping)$ 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 transaction leases 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 before he catched"* doesn't hold, because after the return of the *take* action the player can wait as long as he wants before create the transaction. We prove this by showing that the following formula doesn't hold for the system:

[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 to throw the ball after he creates a transaction otherwise he receives an exeception"*. 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 [13] and Jini [14]. 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 expression 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 languge μCRL is expressive enough to support all the functionnality of JavaSpaces. The main features have been implemented: primitives, leases, timeouts, transactions and events. The remaining services can be easily implemented and they will probabily be included in future versions.

The last part of the paper is dedicated to the study of a very simple JavaSpaces application. Although we have not proven the correctness of the proposed specification, we can see in small examples, that the behavior corresponds to the JavaSpaces specification. We also present some ideas of how to verify properties of applications. In the same way of the example we can study more complex problems.

There are many possibilities for future work. First we can extend the specification by including the remaing features: notify primitive, lease renewal, etc. We can also analyse properties of the formal specification, the transactional model, or to go further in the verification of applications by studying real world applications.

References

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. Hooman and J. van de Pol. Verifying replication on a distributed shared data space with time stamps. In *Proc. 2nd Workshop on Embedded Systems*, pages 107–120. STW, Utrecht, NL, 2001.
13. SUN Microsystems. *JavaSpaces*$^{\text{tm}}$ *Service Specification*, 1.1 edition, October 2000. See `http://java.sun.com/products/javaspaces/`.
14. SUN Microsystems. *Jini*$^{\text{tm}}$ *Technology Core Platform Specification*, 1.1 edition, October 2000. See `http://www.sun.com/jini/specs/`.