Verification of JavaSpaces™ Parallel Programs*

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 illustrate a formal verification method for distributed JavaSpaces applications by analyzing a non-trivial fault tolerant algorithm that solves a typical coordination problem. The problem consists of the computation of an extensive task, performed in parallel by splitting it into smaller and more manageable parts. The proposed solution, based on JavaSpaces coordination primitives, transactions and time-outs, is verified by translating it to the formal language μCRL, together with the previously developed μCRL-model of the JavaSpaces architecture, and by using model checking techniques.

Keywords*: software architecture (JavaSpaces), Formal analysis and verification, Parallel computing, Distributed termination problem.*

1 Introduction

The construction of reliable distributed systems is a difficult problem. The functional correctness of such systems is hard to establish, due to the combinatorial number of possible states and interactions that a number of separate processes can exhibit. Besides this, non-functional requirements, such as efficiency, real-time behaviour, robustness and fault tolerance, require sophisticated algorithms. A possible solution to these problems is the use of *coordination architectures* [22].

In this paper, we present a framework and a methodology for validating applications built using the shared dataspace architecture, JavaSpaces. In previous works [18, 19], we have introduced a formal model of the architecture and, now, we use this model to verify a JavaSpaces application. The problem consists of the parallel summation of a multiset of values, and the proposed solution is a fault tolerant algorithm on which an arbitrary number of workers perform simple additions and one manager detects termination.

Therefore, the main contribution of this paper is the technical validation of the reusable framework, that covers all the main characteristics of the JavaSpaces architecture, and the verification method on a, far from trivial, example. Furthermore, the proposed algorithm can be generalized in order to cover an important class of typical JavaSpaces applications: computationally intensive problems, that are broken down into a number of smaller tasks that can be executed in parallel.

JavaSpaces™ [23], being a coordination architecture, provides strong high-level primitives for communication and synchronization, guaranteeing global consistency. This middleware is a Sun Microsystems, Inc. coordination architecture based on the Linda paradigm [7] and implemented as a Jini™ [24] service. JavaSpaces facilitates the easy implementation of parallel Java programs. JavaSpaces applications are, basically, composed by agents that communicate and synchronize by sharing objects through a shared repository. Objects can be written, looked up, and taken away. A transaction mechanism is provided, which can be used to achieve global consistency and fault tolerance. Time-out and leasing mechanisms are used to avoid that processes can claim resources forever. We shortly introduce the main features of JavaSpaces in Section 2.

Although using such an architecture makes life easier for the application programmer, it remains difficult to establish the correctness of non-trivial examples. Testing a distributed system is hard, due to the non-determinism introduced by different interleavings of the components' executions. Formal methods [20, 9] have been advocated as an alternative to testing. These methods use algorithms from mathematical logic (theorem proving [21] and model checking [8]) in order to guarantee the correctness of a mathematical model.

In order to verify a JavaSpaces application, we developed a formal model of the JavaSpaces architecture, in the process algebra μCRL [14]. A JavaSpaces system of n application programs is modeled by n μCRL processes plus a

*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.

separate μCRL process which models the shared data space. The μCRL toolset [3] can be used to generate the full state space for the system, and subsequently the CADP-model checker [12, 17] can be used to verify correctness conditions on this state space. Of course, this data space model can be reused in new verifications. So we obtain a generic method for verifying distributed JavaSpaces applications.

The example selected to illustrate our method is a typical coordination problem, called *parallel summation*. Initially, the space is filled with a multiset of numbers. These numbers shall be replaced by their total sum, and upon completion this sum shall be reported. We stipulate that the solution shall be efficient and fault tolerant, and shall allow transparent replication.

Our design consists of a number of workers, which repeatedly and independently replace two numbers by their sum, and a manager, which is a special worker that can also detect termination. In this setting, fault tolerance means that the workers and the manager can crash and recover at any time, without doing any harm. Transparent replication means that more workers can be added at any time to assist in the summation process. The actual implementation is based on JavaSpaces, and uses a tricky combination of transactions and time-outs.

The correctness of our solution is not obvious at all. We formally verify in Section 4 that, for several scenarios if the manager is up sufficiently long, then the total sum will eventually be reported. The source code 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. The basic primitives are writing and looking up objects in the repository. JavaSpaces, as a Jini service, also provides support for distributed events, leasing and transactions. Figure 1 presents an overview of the architecture.

A *write* operation places a copy of an entry into the space. Entries are granted for a fixed period of time (called *lease*). The space automatically removes the objects when their lease expires. Entries can be located by "associative lookup" implemented by matching *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*. Read requests are blocking in principle, but processes can limit the amount of time they are willing to wait for a matching entry. If this time expires without finding a matching entry, a *null* entry is returned. *ReadIfExists* is similar to *read*, but it only blocks if there exist matching objects in the space that are involved in some transaction (see below). *Take* and *takeIfExists* are the *destructive* versions of *read*: once an object has been returned, it is removed from the space.

Transactions ensure 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. After the creation of a transaction, a process can either *abort* it, or *commit*. Transactions are also subject to leasing. If the lease expires the space automatically aborts the transactions and raises an *exception*. Transactions affect the behavior of the primitives. E.g., an object written within a transaction is not externally accessible until the transaction commits, if the transaction aborts the insertion will never be visible. Transactions provide a means for enforcing consistency. JavaSpaces' transactions are claimed to preserve the ACID properties: Atomicity, Consistency, Isolation and Durability.

The space also handles some distributed events, in particular: a process can inform the space about its interest in future incoming entries by using the *notify* primitive. The space will notify by an event when a matching object arrives into the space.

All the basic information about JavaSpaces needed to understand the rest of the paper has been included in this section, nevertheless we refer to [13, 23, 15] for a complete introduction to the architecture. In section 4.1 we briefly introduce the formal specification of the main features of the architecture in μCRL.

3 Parallel Summation

We now address the problem of parallel summation of a multiset of numbers. The main difficulty of this application is how to determine when the summation is completed. Among the different possibilities to solve the problem, we propose an algorithm consisting of a number of identical processes (*Workers*) that independently perform simple additions and a *Master*, that is a special worker, who is charged to publish the result when the complete sum is accomplished. First, we present a naive (and wrong) implementation of this idea. And after, we will impose to the system our extra functional requirements and give a correct solution to the problem.

The following two fragments of Java code implement the non terminating solution. Note that we do not show the definition of auxiliary classes and the initialization of the system. The entries in the space are instances of the class `Number` which encapsulates a natural number:

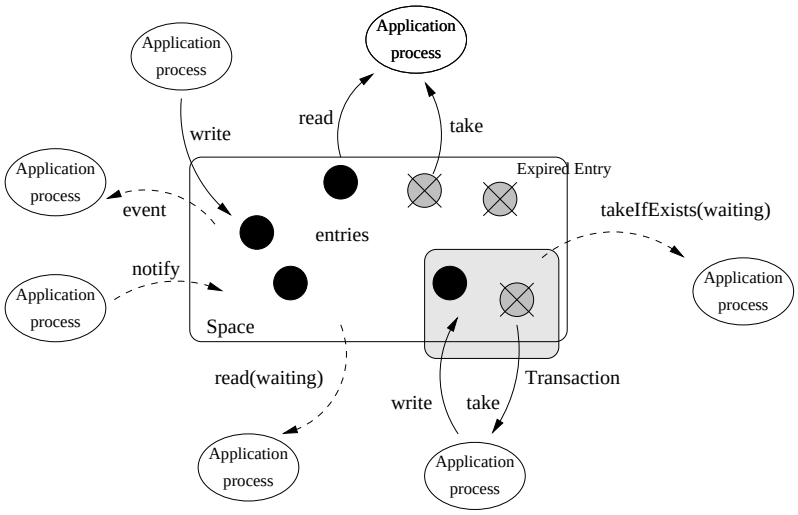

Figure 1. JavaSpaces architecture overview

Listing. Naive Workers' loop

```
for(;;){
  e1 = (Number) space.take(anyNumber, NULL, 0);
  if(e1 == null){return;}
  e2 = (Number) space.take(anyNumber, NULL, 0);
  if(e2 == null){
    space.write(e1, NULL, Lease.FOREVER);
    return;
  }
  space.write(e1.plus(e2), NULL, Lease.FOREVER);
  println("Worker wrote: " + e1.plus(e2));
}
```

A *Worker* first tries to take two entries, one after the other, by performing non-blocking takes matching any number in the space. If he succeeds then he writes the addition and loops, otherwise he undoes the changes (if needed) and halts. The calls to the method `take` get three parameters: first, the template which is also an instance of the class `NUMBER`. Second, the reference to the transaction which is equals to `NULL` as the action is not performed within a transaction. And finally, the time-out of the action, `0` means that if there are not entries in the space the method will not wait. The method `write` receives the entry to be stored, the transaction reference and the lease. The constant `FOREVER` is used to place objects that will never be removed by the space. Let us, now, present the code of the *Master*:

Listing. Naive Master's loop

```
for(;;){
  e1 = (Number) space.take(anyNumber, NULL, 0);
  if(e1 == null){return;}
  e2 = (Number) space.take(anyNumber, NULL, 0);
  if(e2 == null){
    System.out.println("Master publish:"+e1);
    return;
  }
  space.write(e1.plus(e2), NULL, Lease.FOREVER);
  println("Master wrote: " + e1.plus(e2));
}
```

The *Master* is similar to the *Workers*, but if the second take does not succeed the *Master* publishes the value of the first entry as the final result. In case both takes return an entry he performs the normal addition and continues. Note that this solution will lead to incorrect publications, for example: a *Worker* may take the last but one item, and while it is busy, the *Master* might take the last one and think that the algorithm has terminated. Before examining how the *Master* can be sure that he took the last element, let us consider another important issue.

We have imposed to the system a fault tolerance constraint, i.e. any process may suddenly stop and restart. But if a worker crashes after taking a number, this number would be lost. This forces us to consider the use of transactions. To guarantee the non corruption of the data due to the failure of a process, the critical operations have to be encapsulated by a transaction. In case a *Worker* halts or fails in the middle of an operation, the space will automatically recover to a stable state, undoing the modifications.

Recall that transactions are subject to leasing. Now the lease on the *Workers*' transactions provides an upper bound to the duration of a simple summation operation. We choose this time-out (from now denoted t_{op}) to be sufficiently large to perform one addition. It can be approximated by the estimated duration of a simple addition plus the latencies of the coordination primitives. We can, now, propose a mechanism that guarantees the exclusive access of the *Master* to the data:

1. When the *Master* is willing to check termination, he starts a transaction and writes a special entry (*lock*) to prevent *Workers* to start new operations. Thus, *Workers* have to check the non existence of the *lock* entry in the space before starting a new addition.

2. Then, the *Master* waits until the end of all possible active operations, i.e. he has to block more than the upper bound of the *Workers*' operations (at least $t_{op}+1$ time units). We denote this time-out with t_{opM}.

3. Now, he is sure he has exclusive access, i.e no *Worker* has an entry, and no *Worker* can take any of them. Then:

   - If there is only a single entry in the space the *Master* publishes it, commits the transaction and halts.
   - If there are two entries he performs a simple addition, puts the result in the space, removes the lock, commits the transaction and waits until he decides to restart the termination test again.

The *Master*'s operations have to be executed also under a transaction to prevent problems such, for example: the failure of the *Master* after having locked the space will forbid any more progress by *Workers*. The time-out of the transaction (t_{ma}) has to be sufficient to guarantee that the process can perform the steps of the protocol. t_{ma} can be under approximated by: t_{opM} plus the estimated time to perform a simple sum and plus the latencies of the involved coordination primitives. Now, let us, first, see the body of the *Master* process and then the explanation of some details of the implementation:

Listing. Master's loop – correct solution

```
for(;;){
  try{
    Transaction txn = createTransaction(t_ma);
    space.write(lock, txn, Lease.FOREVER);
    space.take(noEntry, txn, t_opM);
    e1 = (Number) space.take(anyNumber, txn, 0);
    e2 = (Number) space.take(anyNumber, txn, 0);
    if(e2 == null){
      txn.commit();
      println("Master publish: " + e1);
      return;
    }
    space.write(e1.plus(e2), txn, Lease.FOREVER);
    space.take(anyLock, txn, 0);
    txn.commit();
    println("Master wrote: " + e1.plus(e2));
  } catch (Exception e){} //loop
  Thread.sleep(t_wait);
}
```

The *Master* first creates a transaction and locks the space by writing a *lock* entry. The *lock* is written inside the transaction, therefore it will not be externally visible except that it blocks *IfExists* actions. Then, the *Master* has to wait until the end of the active operations (t_{opM}). Since we make no assumptions on the relative speed of the clock of different processes, we cannot use local primitives as `Thread.sleep(T_opM)` to perform the wait. However, we can use synchronization between the *Master* process and the space, by reading with a template that matches nothing (`noEntry`). This operation will always block during t_{opM} time units. After the *null* return of this primitive, he has exclusive access and can test the completion of the algorithm.

The behavior of the generic *Workers* is similar to the naive version. But now the operations are executed under a transaction, leased for t_{op} time units. In case a *Worker* receives an exception due to the expiration of the transaction's time-out, he just restarts the algorithm again. Another change is that before every addition, a worker has to check whether the space is locked or not by the *Master*. This test is done by means of a *ReadIfExists* primitive which only blocks if there are matching entries with conflicting transaction locks. Note that the *Master* is never going to free the *lock* before removing it, so, this operation can only result in a *null* return, which allows *Workers* to continue their tasks, or a transaction exception which will force them to restart.

Listing. Workers' loop – correct solution

```
for(;;){
  try{
    Transaction txn = createTransaction(t_op);
    space.readIfExists(lock, txn,Long.MAX_VALUE);
    e1 = (Number) space.take(anyNumber, txn, 0);
    if(e1 == null){
      txn.abort();
      return;
    }
    e2 = (Number) space.take(anyNumber, txn, 0);
    if(e2 == null){
      txn.abort();
      return;
    }
    space.write(e1.plus(e2), txn, Lease.FOREVER);
    space.take(anyLock, txn, 0);
    txn.commit();
    println("Worker wrote: " + e1.plus(e2));
  } catch (Exception e){} //loop
}
```

Since all the critical actions are encapsulated in transactions, all agents can arbitrary fail and restart without corrupting the information of the system. However, to detect the completion of the sum one *Master* should be alive sufficiently long.

The application allows to have any number of running *Workers*. However, replication of the *Master* would lead to incorrectness. Nevertheless, it's possible to imagine a complete replicable application in which all the processes are equal and the role of *Master* or *Worker* is assigned by a special entry or token. In this case, if the actual *Master* dies the *Workers* will compete for the token. Note that this solution will require to manage two different transactions, one as in the previous processes and another to deal with the new token.

The proposed algorithm tries to maximize the efficiency of the computation by allowing as many operations in parallel as possible. Note that *Workers* do not compete between

each other for any resource, so they run completely in parallel. But, performance of the system depends on the selection of accurate upper bounds of the simple additions, the rate of test for termination, which is given by the time the *Master* waits between consecutive loops (t_{wait}) and, of course, on the number of active *Workers* and the reliability of the processes. t_{wait} can be tuned according to the estimated total duration.

If we knew a priori the amount of numbers in the space, we might use a counter, storing this number and decreasing it after every successful operation, this solution will strongly reduce the performance of the system due to the concurrent access to this shared entry. Other similar solutions based on a shared data structure will suffer from the same handicap.

Another possible approach to solve the problem could be based on the dispatch of notification events after successful additions which would allow processes to control the number of entries left in the space and determine the termination. This solution presents difficulties due to the, by specification, unreliable distribution of events, i.e. events may be lost, duplicated or unordered, which would let unfeasible the verification of the algorithm.

Even if the basic idea of the algorithm is rather simple, the proposed solution deals with quite complicated features: mutual exclusion, transactions and relations between time-outs. Therefore we cannot immediately claim that the algorithm is correct. In the following section we are going to use a formal procedure to prove the correctness.

4 Verification by Model Checking

In this section we describe how the case study is modeled in μCRL and how it can be verified. μCRL [14] is a combination of abstract data types and process algebra. Abstract data types are used to describe data structures (like entries, templates, transaction sets) and their operations (like comparing, matching, member test). Process algebra is then used to describe the behaviour of a system, in terms of its data structures, and the atomic actions that it can perform. From process algebra, we will use the following notation: `p.q` (perform p and then perform q); `p + q` (perform arbitrarily either p or q); `sum(x:D,p(x))` (perform $p(d)$ with an arbitrarily chosen d of sort D); `p` $\lhd$ `b` $\rhd$ `q` (if b is true, perform p, otherwise perform q); `p || q` (run processes p and q in parallel).

4.1 Formal model of a JavaSpaces system

The complete description of the formal specification of JavaSpaces can be found in two previous works: *"Formal specification of JavaSpaces™ architecture using μCRL"* [18] and *"μCRL specification of event notification in JavaSpaces™"* [19]. The formal model includes the main features of the architecture: coordination primitives, transactions, leasing and events, only small details, like nested transactions or the lease renewal mechanism, have been omitted. During the process of building the formalism we had to face several difficulties due to existence of unclear or ambiguous issues and to the lack of detail and precision of the JavaSpaces specification, these problems have been solved by making some assumptions about the behavior and by capturing as close as possible the functionalities of the architecture.

We only explain here the essentials of the formal model. A JavaSpaces system is modeled as the parallel system $JavaSpace \,||\, Application_1 \,||\, \cdots \,||\, Application_n$. Here *JavaSpace* is a μCRL process, which arguments represent the internal *state* of the space, and communicates with the applications by synchronous *atomic actions*, derived from the JavaSpaces API. The state of the *JavaSpace* process contains a.o. the following:

- the entries stored in the space, with their lease.
- pending read and take requests, and notification registrations.
- active transactions.
- a system clock, used to remove entries and abort transactions whose lease expired, and to unblock lookup requests whose time-out expired.

The notion of "time" is modeled as a centralized clock, described by a counter, actions occur ordered in between two clock ticks. To guarantee progress in the system the space can only increase the counter if there are no pending internal actions. Transactions are implemented by using local sets of entries; every transaction has three sets in which it stores respectively the written, taken and read entries. The entries stored in these sets cannot be used outside the transaction until the transaction commits.

Applications can synchronize with the space by the following atomic actions[1]:

- *write(id, entry, txn, lease)*
- *read(id, txn, timeout, tmpl)* and *readE(id, txn, timeout, tmpl)*
- *take(id, txn, timeout, tmpl)* and *takeE(id, txn, timeout, tmpl)*
- *Return(id, entry)*
- *create(id, txn, lease)*, *abort(id, txn)* and *commit(id, txn)*

[1] We exclude the primitives related with the notification mechanism, as it is not used in this paper.

- *Exception(txn)*

Here *id* denotes a unique application identifier used to control the access of the agents to the space. *readE* and *takeE* are abbreviations of *readIfExists* and *takeIfExists*. A transaction (*txn*) can be NULL (no transaction) or some transaction-id generated by a *create*. Once a process has created a transaction it has to add to every action the possibility to receive an expiration exception (see examples in the appendix). A *lease* or *time-out* value can be any natural number, or FOREVER. Finally, *tmpl* denotes a template. The lookup-primitives are split in two actions: a request (read, take, ...) and a *Return*. The returned value can be an entry that matches the provided template or a NULL (*entryNull*) if the associated time-out has expired. The other primitives are considered atomic and we assume that they always succeed.

We show, below, the μCRL equivalent of the *Naive Master*'s loop. Remember that this code does not represent the correct solution of the problem, we present it here to illustrate the relation between the Java and the μCRL code. The sources of the processes that are analyzed in the next section are included, in the Appendix. We obtained the process expressions by a systematic, manual translation of the Java code of Section 3.

```
proc  Naive_Master(id:Nat) =
take(id, NULL, tt(0), anyNumber)
.sum(e1:Entry,
   Return(id, e1)
   .(
   take(id, NULL, tt(0), anyNumber)
   .sum(e2:Entry,
      Return(id, e2)
      .(write(id, number(plus(value(e1), value(e2))),
                                  NULL, FOREVER)
      ◁ not(eq(e2, entryNull)) ▷
      M_publish(value(e1)).delta)
   )
   ◁ not(eq(e1, entryNull)) ▷ delta
   )
)
.Naive_Master(id)
```

Once the system is specified in μCRL, the toolset allows to perform symbolic analysis. In general, a specification is transformed into linear format in which an equivalent single process replaces the parallel composition of the processes of the system. Techniques can be applied to this linear model to, for example, eliminate redundant or dead code or to reduce the specification by proving confluence which will lead to a reduction in the state space. The tool gives also the possibility to build a control flow graph (see appendix) and a data dependencies graph which present a compositional view of the system and can be used, for example, to guide finding desirable abstractions.

Other techniques that can be fruitful in order to prove the correctness of the application are simulation, state space visualization, invariant checking and model-checking. The last one is presented in the following section.

The size of the complete model is around 1700 lines of code, the bigger (1500 lines) and more complex part of which corresponds to the JavaSpaces architecture specification that can be re-used by users to analyze different distributed applications based on JavaSpaces with a minimal effort.

4.2 Model Checking

Model checking techniques are (mostly) restricted to finite state systems. Therefore, we have to limit our system by fixing the number of workers, and the amount of entries initially included in the repository. Actually, we also assumed in the verification that the *Workers* can fail once and if they do they cannot recover anymore and the *Master* can also fail once and if he does he recovers and he will not fail anymore. The last assumption is needed in order to guarantee that the *Master* will detect termination in the end.

The correctness claim of the algorithm is that successful termination is detected, i.e. the *Master* publishes the correct result. Therefore, we would like to prove a property meaning that on all possible paths, starting from the initial state, the action `M_publish(n)` will be reached, with `n` representing the correct result. This is expressed in the temporal logic regular alternation-free μ-calculus [17] by the library pattern "inevitable reachability", which is written as follows:

```
mu X.(<true> true
            and [not 'M_publish(n)'] X)
```

The formula will be satisfied if the initial state is included, in the variable `X`, in one of the least fix point iterations. Starting from the empy set of states ($X = \emptyset$), the first iteration will actualize the variable `X` by adding all the states that have at least one outgoing transition (`<true> true`) and all transitions different to `M_publish(n)` go to the empy set, in order words the states that have `M_publish(n)` transitions only. The following iterations will add the states that only have `M_publish(n)` transitions or transitions going to the previous set of states.

The tool support for the verification is provided by the combination of the μCRL tool set [3] and the Evaluator of the CÆSAR ALDÉBARAN DEVELOPMENT PACKAGE (CADP) [12, 17]. First, a labeled transition system representing the full state space is generated by the μCRL tool set. After hiding all internal actions, the state space can be

	# data entries			
# processes	2	3	4	5
1	155	753	4785	33545
2	3738	26082	225463	2200478
3	180881	1464665	15224547	–
4	25648290	–	–	–

Figure 2. Sizes of the generated state space

minimized and visualized, in order to allow visual verification of the external behaviour. The Evaluator tool from the CADP package is used to check properties from temporal logic on the state space.

We checked our system for various numbers of application processes (1,2,3,4) and several numbers of values (2,3,4,5). The size of the state space is displayed in Figure2. In all cases, the formula above is satisfied. The naive implementation was refuted by our method and a counterexample was given. In Figure 3 we visualize the reduced state space, of the correct solution, (only the external behavior of the processes is displayed), in the case of a master and two workers (id 1 or 2), summing up three values (1+2+3). We see various intermediate states. In state number 2 all processes are still alive, however in state 3 and 4 *Worker* 1 and 2 have failed, respectively; in state 5 both of them have failed. We observe that the desired result is always achieved (just before state 6) and the intermediate additions can either be done by the *Workers* or by the *Master*.

The state space generations (see table) have been done in a cluster which consists of 8 nodes, each node is a dual AMD Athlon MP (1.4Gz) system with 2GB of RAM. The generation of the system with 3 processes and 3 data values lasted around 2 hours while the biggest presented state space (4p, 2e) was generated in 24 hours.

5 Related Work

This application builds upon our previous work [18, 19], in which the formal JavaSpaces model was developed. For a gentle introduction to JavaSpaces we refer to the books [13, 15], and SUN's documentation on JavaSpaces [23] and Jini [24]. The basic ideas of JavaSpaces go back to the coordination language Linda [7].

Some work on the formalization of JavaSpaces (or other Linda-like languages) exist, notably [5, 6]. Those authors aim at the semantics of JavaSpaces, its expressiveness, and results on serializability of transactions. Verification of individual JavaSpaces programs was not aimed at.

Our technical approach is similar to the research in [11, 16]. In these papers, programs written under the Splice architecture [4] are verified. Both papers give an operational model of Splice in μCRL, and use the μCRL and CADP tool sets to analyse Splice programs. Splice systems are quite different, because Splice provides a distributed data storage, as opposed to the central storage in JavaSpaces.

There are several efforts to apply formal methods to the verifications of Java programs, among them we point to the Loop Project's tool [1] which analyzes single threaded Java programs by translating the code into PVS and Bandera [10] which deals with multi-threaded systems and it also makes a translation of the original source into a specification language such as Promela. Our approach is mainly focused on the study of the coordination part of distributed systems and encapsulates the details of the communication into the model of JavaSpaces.

6 Conclusion

Through the analysis of a non-trivial example we validated our general framework to verify JavaSpaces applications. The studied problem is a representative example of coordination problems that JavaSpaces is aimed at. The μCRL language and its related tool sets have been shown to be suitable to model check this class of applications. We conclude that small but non-trivial JavaSpaces applications can be effectively verified using our technology. This technology could still be scaled up, by using abstraction and symmetry techniques. In the parallel summation example, the set of values can be abstracted to just its size. Also, there is a lot of symmetry because all workers are identical.

We have seen in section 4.1 how we manually translate the Java code into μCRL. We think that the automatic translation of the code is very important from a methodological point of view and for the "industrial" application of the verification technique, however this process is completely orthogonal to our research and can be carried out by applying techniques implemented in the Bandera or Loop Project's tool, cited above.

Besides scaling up and mechanized translation, future work could be to use the μCRL model of JavaSpaces in order to investigate its meta-properties, and simulate proposed modifications and extensions of JavaSpaces.

By model checking we can only check correctness for a fixed number of processes, entries, and failures. We think that in order to verify general correctness for arbitrary num-

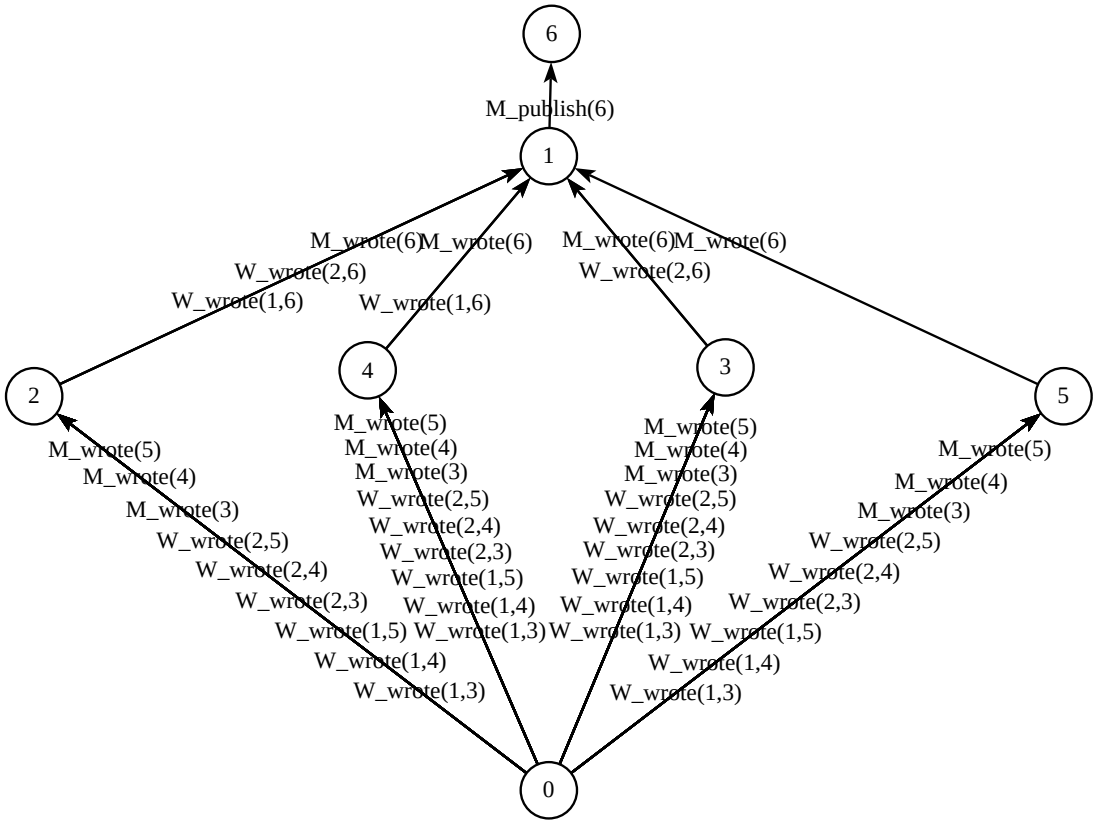

Figure 3. Three processes sum the items (1, 2, 3)

bers, a time consuming interaction with a theorem prover, like PVS [21] is indispensable. Nevertheless, we think that the model checking approach that we propose is complementary to testing. With testing, *some* executions for *some* inputs can be checked. With theorem proving, *all* executions on *all* inputs are checked. The model checking approach we propose is in between: *all* executions on *some* inputs are checked. This is currently the best available automated approach. In [11] this approach is called *scenario based verification*.

As we claimed in the introduction, the proposed algorithm can be generalized in order to solve other similar coordination problems. Furthermore, we can see this generalization as an alternative fault tolerant distributed implementation of the so-called chemical abstract machines[2]. The JavaSpaces will store the chemical molecules and the *Workers* will perform in parallel the chemical reactions.

References

[1] J. van den Berg, M. Huisman, B. Jacobs, and E. Poll. A type-theoretic memory model for verification of sequential Java programs. In D. Bert and C. Choppy, editors, *Recent Trends in Algebraic Development Techniques (WADT'99)*, number 1827 in LNCS. Springer, 2000.

[2] Gerard Berry and Gerard Boudol. The chemical abstract machine. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 81–94. ACM Press, 1990.

[3] 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.

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

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

[8] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. MIT Press, 1999.

[9] E.M. Clarke and J.M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, 1996.

[10] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Păsăreanu, Robby, and Hongjun Zheng. Bandera: extracting finite-state models from java source code. In *International Conference on Software Engineering*, pages 439–448, 2000.

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

[12] 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.

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

[14] 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.

[15] Steven L. Halter. *JavaSpaces example by example*. Prentice Hall PTR, 2002.

[16] 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*, pages 351–358, Madrid, 2002. ACM press.

[17] Radu Mateescu and Mihaela Sighireanu. Efficient on-the-fly model-checking for regular alternation-free mu-calculus. *Science of Computer Programming*, 46(3):255–281, March 2003.

[18] J.C. van de Pol and M. Valero Espada. Formal specification of JavaSpaces™ architecture using μCRL. In *Proc. of COORDINATION*, pages 274 – 290. Springer, 2002.

[19] J.C. van de Pol and M. Valero Espada. μCRL specification of event notification in JavaSpaces™. In *Actas de las X Jornadas de Concurrencia, Jaca, Spain*, pages 191–204. Universidad de Zaragoza, 2002.

[20] J.M. Rushby. Formal methods and their role in the certification of critical systems. Technical Report CSL-95-01, CSL, 1995.

[21] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M.K. Srivas. PVS: Combining specification, proof checking, and model checking. In R. Alur and T.A. Henzinger, editors, *Proc. of CAV'96*, LNCS 1102, pages 411–414. Springer Verlag, 1996.

[22] M. Shaw and D. Garlan. *Software Architecture: Perspectives of an Emerging Discipline*. Prentice-Hall, 1996.

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

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

7 Appendix

In the μCRL specifications of JavaSpaces applications catching an exception corresponds the ability to always synchronize with the `Exception` action. `Excp` abbreviates the expression `Exception(trc).Master(id)`. The *Master* does not include the wait action after every loop since is not necessary to prove termination.

Figure 7 shows the control graph of a *Worker* process, note that the labels are abbreviated: `C` corresponds to `Create`, `E` to `Exception`, `A` to `abort` and so on.

```
proc  Master(id:Nat) =
sum(trc:Nat, create(id, trc, tt(T_ma))
   .(write(id, lock, trc, FOREVER) + Excp)
   .(take(id, trc, tt(T_opM), noEntry) + Excp)
   .(sum(n:Entry, Return(id, n)) + Excp)
   .(take(id, trc, tt(0), anyNumber) + Excp)
   .(sum(e1:Entry, Return(id, e1)
    .((take(id, trc, tt(0), anyNumber) + Excp)
    .(sum(e2:Entry, Return(id, e2)
     .((commit(id,trc) + Excp)
      .M_publish(value(e1)).Finish)
     ◁ eq(e2, entryNull) ▷
     .((write(id, number(plus(value(e1), value(e2))),
                              trc,FOREVER) + Excp)
     .(take(id, trc, tt(0), anyLock) + Excp)
     .(sum(l:Entry, Return(id, l)) + Excp)
     .(commit(id, trc) + Excp)
     .M_wrote(plus(value(e1), value(e2)))))
    + Excp)))
   + Excp))
.Master(id)

proc  Worker(id:Nat) =
sum(trc:Nat, create(id, trc, tt(T_op))
   .(readE(id, trc, FOREVER, anyLock) + Excp)
   .(sum(lock:Entry, Return(id, lock)) + Excp)
   .(take(id, trc, tt(0), anyNumber) + Excp)
   .(sum(e1:Entry, Return(id, e1)
    .((take(id, trc, tt(0), anyNumber) + Excp)
    .(sum(e2:Entry, Return(id, e2)
     .((write(id, number(plus(value(e1), value(e2))),
                              trc,FOREVER) + Excp)
     .(commit(id, trc) + Excp)
     .W_wrote(id, plus(value(e1), value(e2))))
     ◁ not(eq(e2, entryNull)) ▷
      (abort(id, trc).δ + Excp))
    + Excp))
    ◁ not(eq(e1, entryNull)) ▷
     (abort(id, trc).δ + Excp))
   + Excp))
.Worker(id)
```

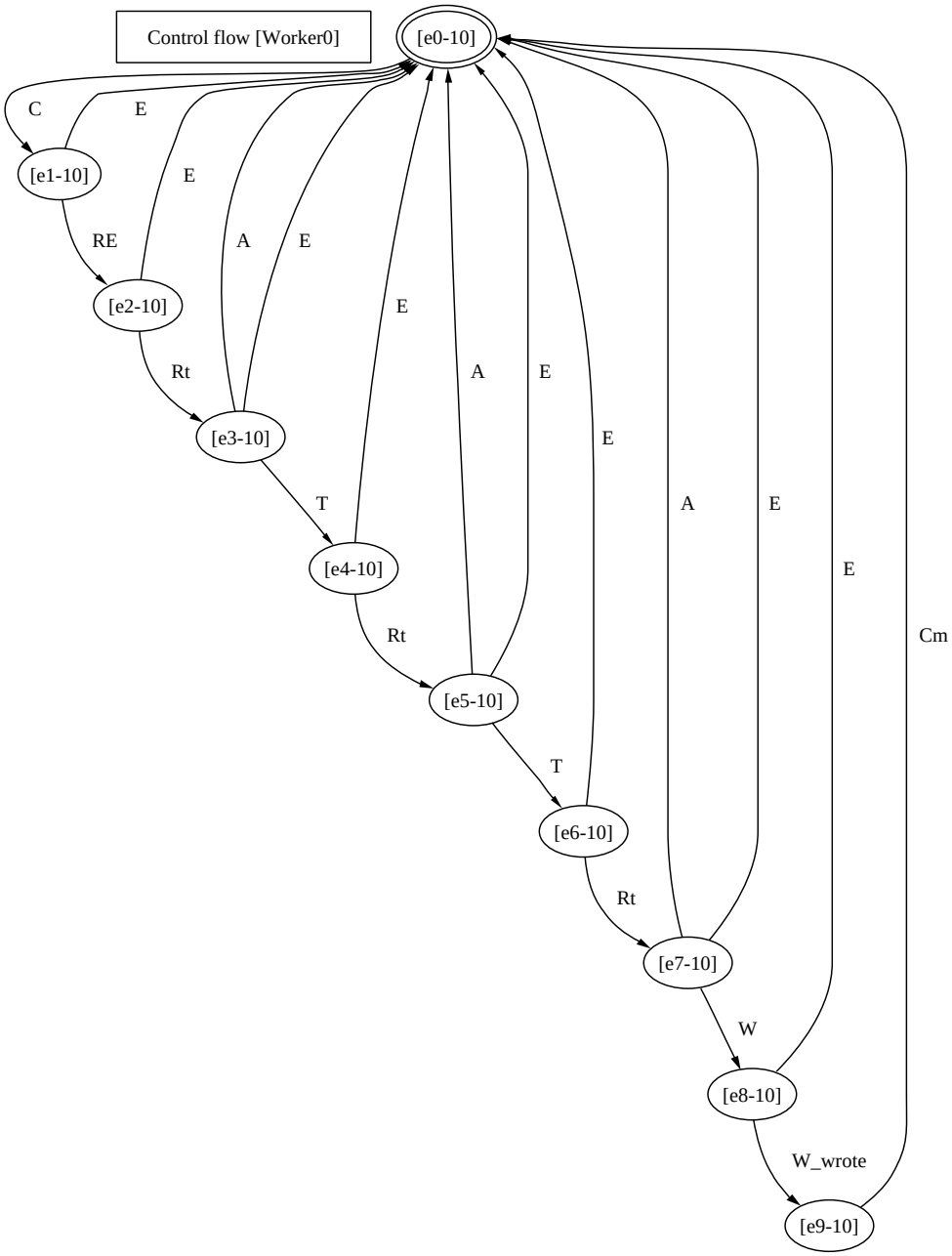

Figure 4. Stategraph of the *Worker* process