Electronic Notes in Theoretical Computer Science 54 (2001)
URL: http://www.elsevier.nl/locate/entcs/volume54.html 14 pages

On the Serializability of Transactions in JavaSpaces

Nadia Busi and Gianluigi Zavattaro

Dipartimento di Scienze dell'Informazione, Università di Bologna, Italy.
E-mail: busi,zavattar@cs.unibo.it

Abstract

JavaSpaces is a coordination infrastructure inspired by the shared dataspace model: processes interact by introducing, consuming, and testing for the presence/absence of data in a common repository. Besides these traditional operations, an event based coordination mechanism is considered which allows for the notification of the introduction of new instances of data in the repository.

JavaSpaces also supports transactions: multiple coordination operations can be grouped into a bundle that acts as a single atomic operation. In this paper we adopt serializability as a criterion to evaluate the correctness of the JavaSpaces transaction semantics: we prove that serializability is satisfied only if we restrict to output, input, and read operations. On the other hand, in the presence of either test for absence or event notification, serializability is not satisfied; we propose an alternative semantics and we prove that it supports serializability.

1 Introduction

Coordination middlewares are emerging as suitable architectures for making easier the programming of distributed applications. JavaSpaces [3] and TSpaces [9], produced by Sun Microsystem and IBM respectively, are the most prominent examples. Both proposals borrow the main features of both the *data-driven* and the *control-driven* coordination models [8]:

- the generative communication operations of Linda [5], according to which processes communicate through production, consumption and test for presence of data in a common data repository; besides the traditional blocking input and read operations also versions which terminates by signalling the absence of matching data are provided;
- an event notification mechanism, allowing for a process to register its interest in the future arrivals of some data, and then receive communication of each occurrence of this event.

A further feature, relevant for distributed applications and supported by both the aforementioned proposals, is a transaction mechanism. A set of coordination operations can be grouped in a transaction, and executed in such a way that either all of them succeed or none of them is performed.

Consistency of the data repository in the JavaSpaces specifications [6] is ensured by requiring transactions to satisfy the so called *ACID* (atomicity, consistency, isolation and durability) properties, traditionally supported by database management systems. In particular, in this paper we are concerned with preservation of the isolation property, also called *serializability*: "Ongoing transactions should not affect each other. Any observer should be able to see other transactions executing in some sequential order".

To meet the isolation requirement for transactions, in the JavaSpaces specification the semantics of coordination operations is affected as follows. A datum produced within a transaction will become accessible from outside the transaction only when the transaction commits; data consumption or test for presence within a transaction can operate on items emitted either within the transaction or in the common dataspace. Moreover, a datum tested for presence within a transaction cannot be consumed by processes outside the transaction until the transaction commits. Concerning the test for absence operations, if the only occurrences of matching data have been withdrawn by another transaction, the operation will wait until that transaction commits before reporting an operation failure. Event notifications performed within a transaction will receive notification of data productions occurring both within the transaction and in the common dataspace. When a transaction commits, all the event notifications local to the transaction are dropped; moreover, the data produced, but not consumed, within that transaction become available in the shared dataspace, and are notified to event registrations performed outside the transactions.

In this paper we provide a formal investigation of the serializability of transactions in JavaSpaces. To this aim, we abstract away from the concrete language, by embedding the coordination primitives in a process calculus equipped with a CHAM-like [1] operational semantics. [1] The proof of serializability relies on a stronger notion, similar to *conflict serializability* in databases [4]: a pair of consecutive operations, performed within two different transactions (or the first outside any transaction and the second within a transaction), can be swapped without altering the final result.

We start our investigation with a first calculus, comprising the basic coordination primitives for data production, consumption and test for presence: in this case, the constraints on the semantics imposed by JavaSpaces specifications [6] are sufficient to guarantee serializability of transactions.

[1] To simplify the treatment, we also forbid nested transactions and we provide only successful termination (commit) of transactions.

Then, we extend the calculus with the test for absence operations, and we provide an example showing that the constraints imposed by [6] on these operations, although necessary, no longer suffice to ensure serializability. We propose an improved, serializable semantics, obtained by adding further constraints on data production and on test for absence operations.

Furthermore, to guarantee the serializability of the calculus extended with an event notification mechanism, we show that it is necessary to modify the semantics of notification operations performed within a transaction specified in [6].

In this paper we abstract away from timeouts, used in JavaSpaces to avoid undesired infinite blocking of processes. However, our examples of non-serializable transactions remain valid also in presence of timeouts.

To the best of our knowledge this is the first work concerned with transactions in a shared dataspace coordination language containing test for absence and event notification primitives. A formal treatment of transaction serializability in the slightly different setting of shared variables (hence concerned with read and write primitives only) can be found in [2].

2 Transactions and the Basic Coordination Primitives

The JavaSpaces specifications adopts the following lock mechanism for transactions: *"When read, an entry is added to the set of entries read by the provided transaction. Such an entry may be read in any other transaction to which the entry is visible, but cannot be taken"*. This policy in necessary in order to ensure serializability of transactions as described by the following example. Consider the configuration

$$\langle a \rangle \mid create(x).read(a).take(b).commit(x) \mid$$
$$create(y).take(a).write(b).commit(y)$$

containing a datum a, a transaction x which reads datum a and consumes b, and a transaction y which removes a and then produces b.

If the above policy is not taken into account, the following non-serializable computation may be executed: the datum a is first read inside the transaction x, and then consumed by the transaction y; after, the datum b is first produced inside transaction y and then consumed inside transaction x; at this point both the transactions may commit. This computation is clearly non-serializable because the two transactions cannot be executed atomically one after the other.

The remainder of this section is devoted to prove that the above policy is enough to ensure serializability of transactions in the case only the basic coordination operations read, write, and take are taken into account.

2.1 The calculus

Let *Name* be a set of data ranged over by a, b, ..., *Const* be a set of program constants ranged over by K, K', ..., and *Txn* a set of transaction names ranged over by x, y, ... We use capital letters X, Y, ..., to range over $\wp(\mathit{Txn})$ (ie. the power-set of *Txn*); we represent sets and multisets with the classical bracket notation, sometimes omitting the brackets in the case of singletons, ie. $\{x\}$ is represented also with x.

Let *Conf* ranged over by P, Q, ... be the set of the possible configurations defined by the following grammar:

$$P ::= \langle a\rangle_X \ \mid\ C \ \mid\ x\{P\} \ \mid\ x\!:\!C\{P\} \ \mid\ P|P$$

$$C ::= \mathbf{0} \ \mid\ \mu.C \ \mid\ C|C \ \mid\ K$$

where:

$$\mu ::= \mathit{write}(a) \ \mid\ \mathit{read}(a) \ \mid\ \mathit{take}(a) \ \mid\ \mathit{create}(x) \ \mid\ \mathit{commit}(x)$$

Configurations are the parallel composition of available data, programs, and active transactions. Available data are modelled by terms $\langle a\rangle_X$, where a denotes the datum and X the set of active transaction from which the datum has been read (it is usually omitted when empty); this information is necessary to implement the transaction policy described above. Programs are represented by terms C containing the coordination primitives.

Active transactions are denoted in two possible ways: on the one hand, $x\{P\}$ models a transaction with name x and involved programs and data described by the configuration P; on the other hand, $x : C\{P\}$ represents a transaction x containing a program C which is interested in performing a coordination operation requiring interaction with the environment outside the transaction. The second kind of notation is necessary to permit the interaction between operations performed inside a transactions and the environment external to the transaction: for instance, we use $x : \mathit{take}(a).P\{Q\}$ to denote a transaction x, containing a program which requires to consume a datum a outside the transaction.

To denote parallel composition we adopt the usual $|$ operator; in the following we use $\prod_i P_i$ to denote the parallel composition of the indexed terms P_i.

A program can be a terminated program $\mathbf{0}$ (which is usually omitted), a prefix form $\mu.P$, the parallel composition of subprograms, or a program constant K.

A prefix μ can be one of the coordination primitives $\mathit{write}(a)$, which introduces a new object $\langle a\rangle$ inside the data repository, $\mathit{read}(a)$, which tests for the presence of an instance of object $\langle a\rangle$, and $\mathit{take}(a)$, which consumes an instance of object $\langle a\rangle$. We consider two further operations: $\mathit{create}(x)$ to start a new transaction, and $\mathit{commit}(x)$ for successful transaction termination.

Constants are used to permit the definition of programs with infinite behaviours. We assume that each constant K is equipped with exactly one definition $K = C$; as usual we assume also that only guarded recursion is used [7].

We use a structural congruence relation on configurations to denote terms with a different syntax but representing the same configuration; this is denoted by $\equiv$ and it is defined as the smallest congruence satisfying the following axioms

$$\begin{array}{ll} (i) \quad P|\mathbf{0} \equiv P & (ii) \; P|Q \equiv Q|P \\ (iii) \; P|(Q|R) \equiv (P|Q)|R & (iv) \; C \equiv K \qquad \text{if } K = C \\ (v) \quad x\{C|P\} \equiv x : C\{P\} & \end{array}$$

comprising the standard axioms for parallel composition (i)–(iii), the standard axiom for program constants (iv), plus an axiom used to permit to a program inside a transaction to move in a position which allows it to perform a coordination operation requiring interaction with the environment outside the transaction.

A transaction is started by a create operation and it is possibly terminated by a commitment operation, performed by all the involved processes. When performed within a transaction, a read operation may test for presence either a datum produced under that transaction or a datum in the external environment. As discussed above, when a datum is read within a transaction it cannot be consumed by processes outside that transaction. A take operation behaves in a similar way, and the selected datum is withdrawn from the dataspace. A datum written within a transaction will not be visible to processes outside the transaction until the transaction commits; before commitment, this datum can be consumed by a process inside the transaction; in that case, the object will never become externally visible.

The semantics of the language is described by a labelled transition system $(Conf, \, Label, \, \longrightarrow)$ where $Label = \{X{:}\tau, X{:}\triangleright, X{:}\triangleleft \;\mid\; X \in \wp(Txn), |X| \leq 1\}$ (ranged over by α, β, $\ldots$) is the set of the possible labels; with abuse of notation we use α to denote also part of a label as in $X : \alpha$. With $x : \alpha$ we denote $\{x\} : \alpha$ and with α we represent $\emptyset : \alpha$. The label $X : \tau$ denotes a standard computation step, while $X : \triangleright$ and $X : \triangleleft$ the beginning and the end of a transaction, respectively. The labelled transition relation $\longrightarrow$ is the smallest one satisfying the axioms and rules in Table 1. Observe that rule (10) makes use of the function $Data(Q)$ (used to denote the set of data available in the configuration Q) inductively defined as follows:

$$\begin{array}{ll} Data(\langle a \rangle_X) = \{a\} & Data(P|Q) = Data(P) \cup Data(Q) \\ Data(C) = Data(x\{P\}) = Data(x : C\{P\}) = \emptyset & \end{array}$$

Axiom (1) indicates that $\langle a \rangle_\emptyset$ can be consumed by a process performing a $take(a)$ operation; the subscript set of transaction names should be empty

(1)	$take(a).P \mid \langle a \rangle_{\emptyset} \xrightarrow{\tau} P$	
(2)	$read(a).P \mid \langle a \rangle_X \xrightarrow{\tau} P \mid \langle a \rangle_X$	
(3)	$write(a).P \xrightarrow{\tau} \langle a \rangle_{\emptyset} \mid P$	
(4)	$create(x).P \xrightarrow{y:\triangleright} y\{P[y/x]\}$	y fresh
(5)	$x : take(a).P\{Q\} \mid \langle a \rangle_Y \xrightarrow{x:\tau} x : P\{Q\}$	$Y \subseteq \{x\}$
(6)	$x : read(a).P\{Q\} \mid \langle a \rangle_Y \xrightarrow{x:\tau} x : P\{Q\} \mid \langle a \rangle_{Y \cup \{x\}}$	
(7)	$x\{\prod_i commit(x).P_i \mid \prod_j \langle a_j \rangle\} \mid \prod_h \langle b_h \rangle_{Y_h} \xrightarrow{x:\triangleleft} \prod_i P_i \mid \prod_j \langle a_j \rangle \mid \prod_h \langle b_h \rangle_{Y_h \setminus x}$	
(8)	$\dfrac{P \xrightarrow{X:\alpha} P'}{P \mid Q \xrightarrow{X:\alpha} P' \mid Q}$	$\alpha = \tau, \triangleright$
(9)	$\dfrac{P \xrightarrow{\tau} P'}{x\{P\} \xrightarrow{x:\tau} x\{P'\}}$	
(10)	$\dfrac{P \xrightarrow{x:\triangleleft} P'}{P \mid Q \xrightarrow{x:\triangleleft} P' \mid Q}$	$Data(Q) = \emptyset$
(11)	$\dfrac{Q \equiv P \quad P \xrightarrow{\alpha} P' \quad P' \equiv Q'}{Q \xrightarrow{\alpha} Q'}$	

Table 1
Operational semantics for the basic calculus (symmetric rules omitted).

because the datum should not be previously read within active transactions. Axiom (2) models the read operation (in this case the subscript set of transaction names does not play any role). Axiom (3) indicates that the effect of the execution of a $write(a)$ operation is the production of the datum $\langle a \rangle_{\emptyset}$ (the subscript set of transaction names is initially empty).

Each active transaction is identified by a unique name; we model this naming mechanism by associating to each transaction a fresh name (i.e. a new name which has not been previously used in the agent). For the sake of simplicity, we do not formally model any mechanism to ensure the global freshness of names, however, standard mechanisms can be exploited which allow for the propagation of locally-fresh names.

When a new transaction is started by a program $create(x).P$, a fresh name y is used to identify uniquely the new transaction; this name must be substituted for x inside P. This is described in axiom (4) where $P[y/x]$ denotes the substitution of x with y inside P. Axioms (5) and (6) describe take and read operations, performed by processes inside a transaction, on data in the external environment; in the case of consumption, the removed datum should not be previously read within other active transactions (this is ensured by the side condition $Y \subseteq \{x\}$); in the case of read, the name of the transaction should be added to the subscript set of transaction names associated with the read datum. Axiom (7) describes transaction commitment: the processes inside the transaction must agree on the commitment operation, the data produced inside the transaction become available to the external environment, and the name of the committed transaction should be removed from the subscript set of transaction names associated to the data in the external environment.

Rule (8) is the usual local rule, while (9) is the application of the local rule to transactions: observe that the transaction name is added to the label in order to denote the transaction under which the action is taken. Rule (10) indicates that a transaction commitment performed by the configuration P can be performed also in $P|Q$ provided that Q does not contain data; this side condition is necessary in order to ensure that all the data in the environment are taken into account by the axiom (7) which introduces the transaction commitment action. Finally, rule (11) is the standard rule for structural congruence.

It is worth noting that we do not fix any constraints concerning the use of the transaction operations create and commit inside programs. For example, the set of configuration *Conf* comprises also the program $commit(y).start(y)$ which requires to commit a transaction before it is created. However, the operational semantics ensures that this kind of terms have no outgoing transitions (see the axiom (7)). Moreover, it is also worth noting that the operational semantics does not permit the execution of nested transactions, ie. transactions inside an outer transaction. Even if this a simplification, we claim that the results we prove on serializability apply also in a more general setting in which nested transactions are supported.

2.2 Serializability

Serializability is a generally accepted criterion for correctness of the execution of transactions. Given the interleaving execution of a set of transactions, it is serializable if the same result can be reached by a *serialized* execution of the transaction. An execution is serialized if all the actions taken inside the same transaction are executed sequentially, one after the other, without interleaving with actions outside the transaction:

In the following we need the following notation: $txn(X{:}\tau) = txn(X{:}\lhd) = txn(X{:}\rhd) = X$ to denote the transaction names occurring in a transition label and $actxn(P) = \{x \;\;|\;\; \exists C, Q \text{ s.t. } x\{Q\} \text{ or } x : C\{Q\} \text{ is a subterm of } P\}$ to denote the set of the transactions active in a configuration.

Definition 2.1 Given the transition sequence $P \xrightarrow{\alpha_1} P_i \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_n} P_n$ we denote it also with $P \xrightarrow{\sigma} P'$ where $\sigma = \alpha_1 \ldots \alpha_n$. The transition sequence $P \xrightarrow{\sigma} P'$, with $actxn(P) = actxn(P') = \emptyset$, is *serialized* iff $\alpha_i = x : \alpha$ implies $\alpha_{i+1} = x : \beta$ or $\alpha_i = x : \lhd$ for $i = 1, \ldots, n-1$. A transition sequence $P \xrightarrow{\sigma} P'$ is *serializable* if there exists a permutation σ' of σ such that $P \xrightarrow{\sigma'} P'$ is a serialized transition sequence.

The following lemma proves that each transition performed inside a transaction can be delayed and executed after a subsequent transition, provided that the latter is performed outside the transaction.

Lemma 2.2 *If* $P \xrightarrow{\beta} P'' \xrightarrow{\alpha} P'$ *with* $\alpha = x : \alpha'$ *where* $\alpha' \neq \triangleright$, *and* $txn(\alpha) \neq txn(\beta)$ *then there exists* P''' *such that* $P \xrightarrow{\alpha} P''' \xrightarrow{\beta} P'$.

We are now ready to present the theorem which reports the serializability result for the calculus with the basic coordination operations only.

Theorem 2.3 *Let* P *be a configuration and* $P \xrightarrow{\sigma_1} P'$ *be a transition sequence such that* $actxn(P') = \emptyset$.

- *If* $actxn(P) = \emptyset$ *then there exists a permutation* σ_2 *of* σ_1 *s.t.* $P \xrightarrow{\sigma_2} P'$ *is serialized.*
- *If* $actxn(P) = \{x\}$ *then there exist* σ_2 *and* σ_3 *s.t. for each* $\alpha \in \sigma_2$ *then* $txn(\alpha) = \{x\}$, $\sigma_2\sigma_3$ *is a permutation of* σ_1, *and* $P \xrightarrow{\sigma_2} P'' \xrightarrow{\sigma_3} P'$ *where* $actxn(P'') = \emptyset$.

3 Adding Test for Absence

In this section we extend the previous calculus with two further coordination primitives *read*∃ and *take*∃ which are variants of the *read* and *take* operations which embed the possibility to test for the absence of matching data, respectively. These operations behave like the corresponding *read* and *take* only in the case the required datum is available for reading or consumption; otherwise, they terminate by indicating the absence of the required datum. These two coordination primitives correspond to the *readIfExists* and *takeIfExists* operations of JavaSpaces.

The two operations are guards for programs with two possible continuations: $read\exists(a)?P\_Q$ and $take\exists(a)?P\_Q$, where P is the continuation chosen in the case the operation succeeds, while Q is chosen if the required datum is not available.

Before presenting the formal syntax and semantics of the extended calculus, we discuss some problems related to serializability.

Consider the following configuration in which a datum is required to be consumed within a transaction and tested for absence outside that transaction:

$$\langle a \rangle \mid create(x).take(a).take(b).commit(x) \mid read\exists(a)?\mathbf{0}\_write(b)$$

Consider now the following computation: the consumption of $\langle a \rangle$ inside the transaction occurs, subsequently the test for absence outside the transaction is performed; after, the datum $\langle b \rangle$ is first produced and then consumed inside the transaction; finally, the transaction commits. This computation is clearly non-serializable because the unique way to perform the test for absence and the output operation outside the transaction is to execute them after the $take(a)$ but before the $take(b)$ operations inside the transaction. This kind of problem is solved in JavaSpaces by avoiding the consumption of data taken within a transaction: these data are simply locked and they are removed only when the transaction commits. Locked data can be neither read nor consumed, and disallow the execution of operations testing the absence of data of that kind.

We now discuss a further problem concerning serializability in the presence of test for absence operations which is not addressed in the JavaSpaces specifications.

Consider the configuration

$$create(x).take\exists(a)?\mathbf{0}\_(take(b).commit(x)) \mid write(a).write(b)$$

and its following computation: the transaction starts, the $take\exists(a)$ operation tests the absence of a and activates the continuation $take(b).commit(x)$; subsequently the two output operations outside the transaction are executed; finally the input operation inside the transition occurs and the transaction commits.

This computation is clearly non-serializable because the unique way for the transaction to commit is that the two write operations outside the transaction are executed exactly between the test for absence and the input operation inside the transaction. To solve this problem we propose the following further lock policy: *after a test for absence is performed inside a transaction on a certain kind of data, no data of that kind can be introduced in the shared dataspace before the end of the transaction.*

This new constraint forbids the execution of the $write(a)$ operation in the computation described above. On the other hand, it does not forbid the execution of output operations performed inside transactions; indeed, data produced inside transactions are not introduced in the globally shared dataspace until the transaction commits. For example, in the configuration

$$\begin{array}{l} create(x).take\exists(a)?\mathbf{0}\_(take(b).commit(x)) \mid \\ create(y).write(a).take(a).write(b).commit(y) \end{array}$$

the $write(a)$ operation inside the transaction y could be executed even after the test for absence in transaction x.

Consider now a similar configuration in which the datum $\langle a \rangle$ produced inside the transaction y is not removed before the transaction commits:

$$\begin{array}{l} create(x).take\exists(a)?\mathbf{0}\_(take(b).commit(x)) \mid \\ create(y).write(a).write(b).commit(y) \end{array}$$

In this case the transaction y cannot commit if the test for absence inside transaction x has been already performed due to the lock policy we have adopted; indeed, if the transaction commits, the emitted datum $\langle b \rangle$ should be introduced in the globally shared dataspace and this cannot happen due to the lock introduced by the previously executed test for absence operation.

$$
\begin{array}{lll}
(3') & write(a).P \xrightarrow{\vec{a}} \langle a\rangle|P & \\
(4') & create(x).P \xrightarrow{y:\triangleright} y\{P[y/x]\}_{\emptyset}^{\emptyset} & y \text{ fresh} \\
(5') & x : take(a).P\{Q\}_T^R|\langle a\rangle_Y \xrightarrow{x:\tau} x : P\{Q\}_T^{R\cup a} & Y \subseteq \{x\} \\
(7') & x\{\prod_i commit(x).P_i|\prod_j\langle a_j\rangle\}_T^R|\prod_h\langle b_h\rangle_{Y_h} \xrightarrow{x:\oplus_j a_j \triangleleft} & \\
 & \prod_i P_i|\prod_j\langle a_j\rangle|\prod_h\langle b_h\rangle_{Y_h\setminus x} & \\
(12) & take\exists(a)?P\_Q|\langle a\rangle \xrightarrow{\tau} P & \\
(13) & read\exists(a)?P\_Q|\langle a\rangle_X \xrightarrow{\tau} P|\langle a\rangle_X & \\
(14) & take\exists(a)?P\_Q \xrightarrow{\neg a} Q & \\
(15) & read\exists(a)?P\_Q \xrightarrow{\neg a} Q & \\
(16) & x : take\exists(a).P\_Q\{R\}_T^R|\langle a\rangle_Y \xrightarrow{x:\tau} x : P\{R\}_T^{R\cup a} & Y \subseteq \{x\} \\
(17) & x : read\exists(a).P\_Q\{R\}_T^R|\langle a\rangle_Y \xrightarrow{x:\tau} x : P\{R\}_T^R|\langle a\rangle_{Y\cup x} & \\
(18) & \dfrac{P \xrightarrow{X:\neg a} P'}{P|Q \xrightarrow{X:\neg a} P'|Q} & a \notin Data(Q) \cup Rem(Q) \\
(19) & \dfrac{P \xrightarrow{\neg a} P'}{x\{P\}_T^R \xrightarrow{x:\neg a} x\{P'\}_{T\cup\{a\}}^R} & \\
(20) & \dfrac{P \xrightarrow{x:A\triangleleft} P'}{P|Q \xrightarrow{x:A\triangleleft} P'|Q} & Data(Q) = \emptyset \text{ and } A \cap Tfa(Q) = \emptyset \\
(21) & \dfrac{P \xrightarrow{\vec{a}} P'}{P|Q \xrightarrow{\vec{a}} P'|Q} & a \notin Tfa(Q) \\
(22) & \dfrac{P \xrightarrow{\vec{a}} P'}{x\{P\}_T^R \xrightarrow{x:\tau} x\{P'\}_T^R} &
\end{array}
$$

Table 2
Operational semantics for test for absence (symmetric rules omitted).

We are now ready to present the formal syntax and semantics of the calculus with test for absence. Formally, the two new operations are introduced as guards for programs with two possible continuations:

$$C ::= \dots \mid \eta?C\_C$$

where:

$$\eta ::= read\exists(a) \mid take\exists(a)$$

Moreover, we have to add two kinds of information to active transactions: the set of data tested for absence and those removed during the transaction. This is achieved by using the new configurations:

$$P ::= \ldots \mid x\{P\}_T^R \mid x:C\{P\}_T^R$$

where $R, T \in \wp(Name)$ are two sets of data representing the kind of data removed and tested for absence inside the transaction, respectively.

The new set of configurations is denoted by $Conf_\exists$; while the new set of labels is denoted by $Label_\exists = Label \cup \{X{:}\neg a, X{:}\vec{a}, X{:}A\lhd \mid X : \wp(Txn), a \in Name, A \subseteq Name\}$. The first label is used to model test for absence operations on datum a, the second label denotes the execution of a $write(a)$ operation, while the third label is the new label for transaction commitment indicating also the multisets of data which have been produced, but not consumed, during the transaction and should be introduced in the shared repository after transaction commitment.

The rule (v) of the structural congruence $\equiv$ should be modified according to the new syntax:

$$(v')\ x\{C|P\}_T^R \equiv x:C\{P\}_T^R$$

The operational semantics is defined by the labelled transition system $(Conf_\exists, Label_\exists, \longrightarrow)$ where the labelled transition relation $\longrightarrow$ is the smallest one satisfying the axioms and rules in Table 1 and in Table 2 where $(3')$, $(4')$ $(5')$, and $(7')$ are substituted for the corresponding rules in Table 1. The rules (18) and (20) use the two functions $Rem(P)$ and $Tfa(P)$, denoting the set of data removed and those tested for absence inside transactions active in the configuration P, respectively. They are inductively defined as follows:

$$Rem(x\{P\}_T^R) = Rem(x:C\{P\}_T^R) = R \qquad Tfa(x\{P\}_T^R) = Tfa(x:C\{P\}_T^R) = T$$

$$Rem(P|Q) = Rem(P) \cup Rem(Q) \qquad Tfa(P|Q) = Tfa(P) \cup Tfa(Q)$$

$$Rem(C) = Data(\langle a \rangle_X) = \emptyset \qquad Tfa(C) = Tfa(\langle a \rangle_X) = \emptyset$$

Axiom $(3')$ introduces the new label $\vec{a}$ denoting the execution of a $write(a)$ operation. Axioms $(4')$ and $(5')$ are the adaptations of the corresponding rules to the new syntax; in particular, $(5')$ updates the set of data removed from the environment by input operations inside the transaction. Axiom $(7')$ introduces the new label $X{:}A\lhd$ (the notation $\oplus_j a_j$ denotes the multiset union of all the singletons $\{a_j\}$).

Axioms (12) and (13) describe the successful execution of the new $take\exists(a)$ and $read\exists(a)$ operations, respectively. These new operations fail when no datum $\langle a \rangle$ is found in the environment; this is modelled by the label $\neg a$ introduced by the axioms (14) and (15). Axioms (16) and (17) are adaptations of (12) and (13) to the case in which the operations are executed inside a transaction; in (16) the set of data removed inside the transaction is updated, while in (17) the subscript set of transaction names associated to the read datum is extended with the name of the current transaction.

A transition labelled with $\neg a$, representing a test for absence of a, can be performed only if the environment does not contain any $\langle a \rangle$ and also no $\langle a \rangle$ have been previously consumed inside an active transaction (see rule (18)). Moreover, when a test for absence is performed inside a transaction, the subscript set T of data tested for absence must be updated (see rule (19)). According to rule (20) a transaction can commit only if the data it introduces in the shared repository are not currently tested for absence inside other active transactions; moreover, the side condition $Data(Q) = \emptyset$ ensures that all the data available in the environment when a transaction commits are taken into account by the rule $(7')$ (which introduces transaction commitment). Rule (21) ensures that an output operation of $\langle a \rangle$ is performed only if active transaction exists which already tested for the absence of that kind of datum. On the other hand, this output operation can be performed if executed inside a transaction (see rule (22)).

The lock policy that we propose ensures the serializability of transaction; this is formally proved by the fact that the Lemma 2.2 and the Theorem 2.3 hold also in the new calculus extended with test for absence.

4 Adding Event Notification

In this section we extend the calculus with an event notification mechanism inspired by the *notify* primitive of JavaSpaces.

The syntax of the kernel language is simply extended with a new prefix:

$$\mu ::= \ldots \quad | \quad notify(a, C)$$

The new program $notify(a, C).P$ can register its interest in the future incoming arrivals of the data of kind a, and then receive communication of each occurrence of this event. When this event occurs, a new instance of the program C is activated as reaction to the event. This behaviour can be modelled by introducing a new term $on(a, C)$, which is a listener that spawns an instance of program C every time a new object $\langle a \rangle$ is introduced in the dataspace. Formally we extend the configurations as follows:

$$P ::= \ldots \quad | \quad on(a, P)$$

The new set of configurations is denoted with $Conf_n$.

According to the JavaSpaces specification *"a notify ... applies to write operations that are committed to the entire space. A notify performed under a ... transaction additionally provides notification of writes performed under that transaction."* Following this approach a listener inside a transaction has visibility of new data produced also outside that transaction. For example in

$$(23) \quad notify(a,P).Q \xrightarrow{\tau} on(a,P)|Q$$

$$(3'') \quad write(a).P|\prod_i on(a,P_i) \xrightarrow{\vec{a}} \langle a\rangle|P|\prod_i(P_i|on(a,P_i))$$

$$(7'') \quad x\{\prod_i commit(x).P_i|\prod_j\langle a_j\rangle|\prod_k on(c_k,R_k)\}_T^R|\prod_h\langle b_h\rangle_{Y_h}|\prod_l on(d_l,Q_l)$$
$$\xrightarrow{x:\oplus_j a_j \lhd} \prod_i P_i|\prod_j\langle a_j\rangle|\prod_h\langle b_h\rangle_{Y_h\backslash x}|\prod_l(on(d_l,Q_l)|\prod_{(\oplus_j a_j)(d_l)} Q_l)$$

$$(20'') \quad \frac{P \stackrel{x:A\lhd}{\longrightarrow} P'}{P|Q \stackrel{x:A\lhd}{\longrightarrow} P'|Q} \qquad Data(Q)=\emptyset \text{ and } A\cap Tfa(Q)=\emptyset \text{ and } A\cap On(Q)=\emptyset$$

$$(21'') \quad \frac{P \xrightarrow{\vec{a}} P'}{P|Q \xrightarrow{\vec{a}} P'|Q} \qquad a\notin Tfa(Q)\cup On(Q)$$

Table 3
Operational semantics for event notification (symmetric rules omitted).

the configuration

$$create(x).notify(a,commit(x)) \mid write(a)$$

the transaction may commit. Indeed, consider the following computation: first the transaction is started, then the *notify* operation produces the listener $on(a,commit(x))$; then the output operation outside the transaction is executed and activates as reaction the program $commit(x)$ inside the transaction; at this point the transaction may commit. This computation is clearly non-serializable because the transaction can commit only if the output operation outside the transaction is performed after the *notify* operation inside the transaction.

To tackle this problem, we propose to change the semantics of event notification performed inside transactions: *a notify operation performed under a transaction applies only to write operations committed inside the same transaction.*

In order to prove that transaction serializability is now satisfied we define the following formal semantics: $(Conf_n, Label_n, \longrightarrow)$ where the labels are the same as those of the calculus with test for absence $Label_n = Label_\exists$ and the transition relation $\longrightarrow$ is defined by the axioms and rules in Table 1, in Table 2, and in Table 3, where $(3'')$, $(7'')$, $(20'')$, and $(21'')$ are substituted for the corresponding rules in the previous tables.

The unique new axiom is (23) which produces a listener as effect of the execution of a *notify* operation. The other rules are adaptations of previous rules: $(3'')$ notifies to listeners the occurrence of a write operation; whereas, $(7'')$ notifies the introduction of all the data produced by a transaction which is currently under commitment. As for $(7')$ the notation $\oplus_j a_j$ denotes the multiset union of all the singletons $\{a_j\}$; here, we use also $(\oplus_j a_j)(d_l)$ to denote the number of occurrences of d_l inside the multiset $(\oplus_j a_j)$.

The rules $(20'')$ and $(21'')$ use a new function $On(P)$ which returns the set of data on which there exist listeners active in the configuration P; it is inductively defined as follows:

$$On(on(a, C)) = \{a\} \qquad On(P|Q) = On(P) \cup On(Q)$$
$$On(C) = On(\langle a \rangle_X) = On(x\{P\}^R_T) = On(x\!:C\{P\}^R_T) = \emptyset$$

This function is used in order to ensure that no other listeners exist in the environment which could be interested in data produced by a write operation (rule $(20'')$) or by a committed transaction (rule $(21'')$).

The semantics we propose ensures the serializability of transaction as the Lemma 2.2 and the Theorem 2.3 hold also in the complete calculus comprising both the test for absence and the event notification operations.

References

[1] G. Berry and G. Boudol. The Chemical Abstract Machine. *Theoretical Computer Science*, 96:217–248, 1992.

[2] N. De Francesco, U. Montanari, and G. Ristori. Modelling Concurrent Accesses to Shared Data via Petri Nets. In *Proc. Programming Concepts, Methods and Calculi (PROCOMET)*, pages 403–422. Elsevier Publisher, 1994.

[3] E. Freeman, S. Hupfer, and K. Arnold. *JavaSpaces Principles, Patterns, and Practice.* Addison-Wesley, 1999.

[4] H. Garcia-Molina, J.D. Ullman, and J. Widom. *Database System Implementation.* Prentice-Hall, 2000.

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

[6] Sun Microsystems. JavaSpaces Service Specification, available at `http://java.sun.com/products/javaspaces`. 1998.

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

[8] G.A. Papadopoulos and F. Arbab. Coordination Models and Languages. *Advances in Computers*, 46:329–400, 1998.

[9] P. Wyckoff, S. McLaughry, T. Lehman, and D. Ford. T spaces. *IBM Systems Journal*, 37(3):454–474, 1998.