On the Serializability of Transactions in Shared Dataspaces with Temporary Data

Nadia Busi
Dept. of Computer Science
Mura A. Zamboni, 7
Univ. of Bologna, Italy
busi@cs.unibo.it

Gianluigi Zavattaro
Dept. of Computer Science
Mura A. Zamboni, 7
Univ. of Bologna, Italy
zavattar@cs.unibo.it

ABSTRACT

Several coordination platforms based on the shared dataspace approach introduces, besides the typical Linda-like coordination primitives (used to produce, consume, and test for the presence/absence of data in a common repository), a transaction mechanism provided to group coordination primitives which should be executed in such a way that either all succeed or none of them is performed. In this paper we continue the investigation of the serializability of transactions in shared dataspace coordination languages that has been initiated in [2]. The new contribution consists of the analysis of the interplay between transactions and temporary data, ie., data with an associated expiration time.

Keywords

Shared dataspace coordination, temporary data, transaction serializability.

1. INTRODUCTION

In the last years we assisted to the development of middleware platforms for the coordination of dynamically reconfigurable federations of devices and processes. In this context, two relevant commercial proposals are represented by JavaSpaces [3] and TSpaces [7], produced by Sun Microsystem and IBM respectively. Both coordination middlewares are essentially based on the generative communication metaphor proposed by Linda [4]: processes communicate through production, consumption and test for presence of data in a common data repository; besides the traditional blocking production and test for presence operations, also the corresponding nonblocking versions, which terminate by signalling the absence of matching data, are provided; after its insertion in the dataspace, a datum has an independent existence, until it is not withdrawn by a consumer.

An interesting extension to the basic model, 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 [5] 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.

Recently, in [2], a formal investigation of the serializability of transactions in a process calculus containing the coordination primitives of JavaSpaces has been initiated.[1] As far as only primitives for data production, consumption and test for presence are concerned, the constraints on the semantics imposed by the JavaSpaces specifications [5] are sufficient to guarantee the serializability of transactions. However, when also the nonblocking versions of data consumption and test for presence are considered, the constraints imposed by the specifications, although necessary, no longer suffice to ensure serializability. In [2] an improved, serializable semantics, obtained by adding further constraints on data production and on test for absence operations, is proposed.

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

Another relevant extension to the basic model, permitting to avoid the accumulation of outdated information, is represented by temporary data. rather than maintaining a datum until it has been explicitly consumed, the lifetime of the datum is decided by the producer. After this time has been expired, the existence of the datum is no longer granted.

In this paper we extend the work initiated in [2], by investigating the serializability of transactions in presence of temporary data. We start our analysis of temporary data considering the strict removal policy for expired data (called leasing) of JavaSpaces: as soon as the lifetime of a datum expires, the datum can no longer be used and it is removed from the dataspace.

We show that the introduction of leased data in the basic calculus – with only data production, consumption and test for presence – does not affect serializability. When moving to the full calculus containg also nonblocking predicates, we provide some examples, showing that the constraints in [5, 2] no longer ensure serializability when leased data are taken into account. Indeed, to guarantee the serializability of the full calculus, it is necessary to delay the effective removal of an expired datum until all transactions which tested that datum for presence commit, and all previously expired data have been removed. Besides weakening the benefits of temporary data to prevent the accumulation of outdated information, this solution for leased data also presents the drawback of potentially blocking the execution of the predicates interested in data expired but not effectively removed. For these motivations, it seems more reasonable to weaken the data removal policy adopted by JavaSpaces in the following way: after the lifetime of a datum has been expired, the existence of the datum is no longer granted; however, an expired datum remains available for consumption (or test for presence) until it is not effectively removed from the dataspace by an expired-data collector.

In [1], this interpretation of temporary data alternative to leasing has been investigated under two different implementations of the collector. The first one, called *unordered collection*, removes one of the expired data, whereas the second one, called *ordered collection*, removes one of the data which expired first. To avoid that an expired datum read by an active transaction prevents the collector from removing all successively expired data, the more reasonable choice is represented by unordered collection.

We propose an improved semantics for the full calculus with temporary data and unordered collection, obtained by preventing the collector to garbage expired data that have been read by a currently active transaction, and we show that it is serializable.

The paper is structured as follows: Section 2 discusses the primitives for production, consumption, and test for presence; Section 3 considers also the test for absence operations; Section 4 reports some concluding remarks. Due to space limit the proofs of lemmas and theorems are not reported.

2. THE BASIC PRIMITIVES

In this section we introduce the basic calculus which comprises the *write*, *read*, and *take* coordination primitives plus the operations *start* and *commit* for transactions. This calculus is essentially an extension of the basic calculus reported in [2] with the *leasing* mechanism of JavaSpaces: a *write* operation specifies a time to live for the datum to be produced; when this time expires the produced datum is no more available neither for reading nor for consumption.

In the JavaSpaces specifications a typical *read* locking mechanism is considered: *"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 locking policy is necessary in order to ensure serializability of transactions as described by the following example. Consider

$$\begin{aligned}&\langle a\rangle \mid \mathit{create}(x).\mathit{read}(a).\mathit{take}(b).\mathit{commit}(x) \mid\\ &\mathit{create}(y).\mathit{take}(a).\mathit{write}(b).\mathit{commit}(y)\end{aligned}$$

representing a state of a shared dataspace system in which there is a datum a available inside the repository, a first transaction x which reads datum a and consumes b, and a second 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.

In [2] it is proved that this locking mechanism is sufficient to ensure the serializability of transactions using a calculus with persistent data. Here, we prove that serializability is ensured even in the new calculus with leased data.

2.1 The basic calculus

Let *Name* be a set of data ranged over by a, b, $\ldots$, *Const* be a set of program constants ranged over by K, K', $\ldots$, and *Txn* a set of transaction names ranged over by x, y, $\ldots$. We use capital letters X, Y, $\ldots$, 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, ie. $\{x\}$ is represented also with x.

In order to model temporary data we need to represent the passing of time. To be as general as possible, we do not fix any specific model of time. We only assume what follows: *Time*, ranged over by t, t', $\ldots$, is a set of time instants; *Inter*, ranged over by Δt, $\Delta t'$, $\ldots$, is a set of time intervals; $\leq$ is a total order on *Time* such that $t \leq t'$ means that the time instant t' follows the instant t; $+ : \mathit{Time} \times \mathit{Inter} \rightarrow \mathit{Time}$ is an addition operation such that $t + \Delta t$ is the time instant in which a time interval Δt, starting at time instant t, will finish. We make the minimal reasonable assumption that, for any time instant t and time interval Δt, $t \leq t + \Delta t$; this means that the time instant in which a time interval finishes follows the instant in which it starts.

Let *Proc* ranged over by P, Q, $\ldots$ be the set of the possible processes defined by the following grammar:

$$
\begin{array}{rcl}
P & ::= & \langle a \rangle_X^t \mid C \mid x\{P\} \mid x\!:\!C\{P\} \mid P|P \\
C & ::= & \mathbf{0} \mid \mu.C \mid C|C \mid K
\end{array}
$$

where:

$$
\begin{array}{rcl}
\mu & ::= & \mathit{write}(a, \Delta t) \mid \mathit{read}(a) \mid \mathit{take}(a) \mid \\
 & & \mathit{create}(x) \mid \mathit{commit}(x)
\end{array}
$$

Processes are the parallel composition of available data, programs, and active transactions. Available data are modelled by terms $\langle a \rangle_X^t$, where a denotes the datum, X the set of active transaction from which the datum has been read (it is usually omitted when empty) – this information is used to implement the transaction policy described above –, and t is the expiration time of the datum (it is sometimes omitted when it is not important in the current context). 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 process 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 $\mathbf{0}$ (term which is usually omitted), a prefix form $\mu.P$ guarded by a coordination primitive μ, the parallel composition of subprograms $P|Q$, or a program constant K. A prefix μ can be one of the primitives $\mathit{write}(a, \Delta t)$, which introduces a new object $\langle a \rangle$ inside the data repository with a time to live of Δt (we sometimes use the simplified prefix $\mathit{write}(a)$ when the time to live of the datum has no importance), $\mathit{read}(a)$, 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 [6].

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

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

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.

In order to model the passing of time in the system, we introduce configurations: let $\mathit{Conf} = \{[P, t] \mid P \in \mathit{Proc}, t \in \mathit{Time}\}$ be the set of the possible configurations, described by a process P (which denotes the active programs, transactions, and data available in the system) and a time instant t (which indicates the current time in the system).

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 datum will never become externally visible.

The semantics of the language is described by a labelled transition system $(\mathit{Conf}, \mathit{Label}, \longrightarrow)$ where $\mathit{Label} = \{X\!:\!\tau, X\!:\!\triangleright, X\!:\!\triangleleft, X\!:\!- \mid X \in \wp(\mathit{Txn}), |X| \leq 1\}$ (ranged over by α, β, ...) 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 last label $X\!:\!-$ indicates a step during which no explicit coordination operations are executed, but due to the passing of time data expire and they are removed from the configuration. The labelled transition relation $\longrightarrow$ is the smallest one satisfying the axioms and rules in Table 1.

In axiom (7) and rules (8) and (11) we use an auxiliary function $P \setminus t$ to remove the data inside P which expired before the time t. The function is inductively defined as follows:

$$
\langle a \rangle_X^{t'} \setminus t = \begin{cases} \langle a \rangle_X^{t'} & \text{if } t \leq t' \\ \mathbf{0} & \text{otherwise} \end{cases}
$$

$$
(P|Q) \setminus t = (P \setminus t)|(Q \setminus t) \qquad C \setminus t = C
$$

$$
(x\{P\}) \setminus t = x\{P \setminus t\} \qquad (x : C\{P\}) \setminus t = x : C\{P \setminus t\}
$$

Observe also that rule (10) makes use of the function $\mathit{Data}(Q)$ (used to denote the set of data available in the configuration Q) inductively defined as follows:

$$
\mathit{Data}(\langle a \rangle_X^t) = \{a\} \qquad \mathit{Data}(P|Q) = \mathit{Data}(P) \cup \mathit{Data}(Q)
$$

$$
\mathit{Data}(C) = \mathit{Data}(x\{P\}) = \mathit{Data}(x\!:\!C\{P\}) = \emptyset
$$

Axiom (1) indicates that $\langle a \rangle_\emptyset^{t_a}$ can be consumed by a process performing a $\mathit{take}(a)$ operation; the subscript set of transaction names should be empty because the datum should not be previously read within active transactions. The side condition imposes two constraints: (i) the first one is that

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

(1)	$[\mathit{take}(a).P \mid \langle a\rangle^{t_a}_{\emptyset}, t] \xrightarrow{\tau} [P, t']$	$t \leq t'$ and $t' \leq t_a$
(2)	$[\mathit{read}(a).P \mid \langle a\rangle^{t_a}_{X}, t] \xrightarrow{\tau} [P \mid \langle a\rangle^{t_a}_{X}, t']$	$t \leq t'$ and $t' \leq t_a$
(3)	$[\mathit{write}(a, \Delta t).P, t] \xrightarrow{\tau} [\langle a\rangle^{t'+\Delta t}_{\emptyset} \mid P, t']$	$t \leq t'$
(4)	$[\mathit{create}(x).P, t] \xrightarrow{y:\triangleright} [y\{P[y/x]\}, t']$	y fresh and $t \leq t'$
(5)	$[x : \mathit{take}(a).P\{Q\} \mid \langle a\rangle^{t_a}_{Y}, t] \xrightarrow{x:\tau} [x : P\{Q\}, t']$	$Y \subseteq \{x\}$ and $t \leq t'$ and $t' \leq t_a$
(6)	$[x : \mathit{read}(a).P\{Q\} \mid \langle a\rangle^{t_a}_{Y}, t] \xrightarrow{x:\tau} [x : P\{Q\} \mid \langle a\rangle^{t_a}_{Y \cup \{x\}}, t']$	$t \leq t'$ and $t' \leq t_a$
(7)	$[x\{\prod_i \mathit{commit}(x).P_i \mid \prod_j \langle a_j\rangle^{t_j}\} \mid \prod_h \langle b_h\rangle^{t_h}_{Y_h}, t] \xrightarrow{x:\triangleleft}$	
	$[(\prod_i P_i \mid \prod_j \langle a_j\rangle^{t_j} \mid \prod_h \langle b_h\rangle^{t_h}_{Y_h \setminus x}) \setminus t', t']$	$t \leq t'$
(8)	$[\langle a\rangle^{t_a}_{X}, t] \xrightarrow{-} [\mathbf{0}, t']$	$t \leq t'$ and $t' \not\leq t_a$
(9)	$\dfrac{[P, t] \xrightarrow{X:\alpha} [P', t']}{[P \mid Q, t] \xrightarrow{X:\alpha} [P' \mid (Q \setminus t'), t']}$	$\alpha = \tau, \triangleright, -$
(10)	$\dfrac{[P, t] \xrightarrow{\alpha} [P', t']}{[x\{P\}, t] \xrightarrow{x:\alpha} [x\{P'\}, t']}$	$\alpha = \tau$
(11)	$\dfrac{[P, t] \xrightarrow{-} [P', t']}{[x\{P\}, t] \xrightarrow{-} [x\{P'\}, t']}$	
(12)	$\dfrac{[P, t] \xrightarrow{x:\triangleleft} [P', t']}{[P \mid Q, t] \xrightarrow{x:\triangleleft} [P' \mid (Q \setminus t'), t']}$	$\mathit{Data}(Q) = \emptyset$
(13)	$\dfrac{Q \equiv P \quad [P, t] \xrightarrow{\alpha} [P', t'] \quad P' \equiv Q'}{[Q, t] \xrightarrow{\alpha} [Q', t']}$	

the current time, in the reached configuration, should not preceed the current time of the initial configuration; this constraint, which reflects the passing of time, is used also in all the other axioms; (ii) the second one, on the other hand, ensures that the read datum is not yet expired.

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 $\mathit{write}(a, \Delta t)$ operation is the production of $\langle a\rangle^{t'+\Delta t}_{\emptyset}$ where we impose that the subscript set of transaction names is initially empty, and the expiration time of the new datum is obtained by adding the time to live to the current time t' of the configuration in which the new datum is introduced.

Each active transaction is identified by a unique name; we model this naming mechanism by associating to each transaction a fresh name (ie. 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 $\mathit{create}(x).P$, 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. Observe also that in the reached configuration, the data which expire between the time t in the initial configuration and the time t' of the reached configuration are removed by exploiting the auxiliary function $P \setminus t$ described above. Axiom (8) allows for the withdrawal of expired data which are no more available neither for reading nor for consumption (see the side consitions of axioms (1), (2), (5), and (6)).

Rule (9) is the usual local rule, where the auxiliary function $P \setminus t$ is used to ensure that all the expired data are removed. Rule (10) 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. In the case of steps related to data expiration only (ie. those labeled with $-$), we do not add the transaction name to the label, because during these kind of steps no operations are executed from within any transaction (see rule (11)). Rule (12) indicates that a transaction commitment performed by the configuration P can be performed also in $P \mid Q$ 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 (13) is the standard rule for structural congruence.

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: $txn(X{:}\tau) = txn(X{:}\lhd) = txn(X{:}\rhd) = txn(X{:}t) = X$ to denote the transaction names occurring in a transition label and $actxn(P) = \{x \mid \exists C, Q \text{ s.t. } x\{Q\} \text{ or } x : C\{Q\} \text{ is a subterm of } P\}$ to denote the set of the transactions active in a configuration. Given the sequence of labels $\sigma = \alpha_1 \ldots \alpha_n$, we denote with $[P, t] \xrightarrow{\sigma} [P', t']$ the sequence of transitions $[P, t] \xrightarrow{\alpha_1} [P_1, t_1] \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_n} [P_n, t_n]$ and with σ^- we represent the sequence obtained from σ by removing all the labels $-$.

DEFINITION 2.1. *A transition sequence $[P, t] \xrightarrow{\sigma} [P', t']$, with $actxn(P) = actxn(P') = \emptyset$ and $\sigma^- = \alpha_1 \ldots \alpha_n$, is* serialized *iff $\alpha_i = x : \alpha$, with $\alpha \neq \lhd$, implies $\alpha_{i+1} = x : \beta$, for each $i = 1, \ldots, n-1$. A transition sequence $P \xrightarrow{\sigma} P'$ is* serializable *if there exists σ' such that σ'^- is a permutation σ^- and $[P, t] \xrightarrow{\sigma'} [P', t']$ 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 $[P, t] \xrightarrow{\alpha} [P'', t''] \xrightarrow{\beta} [P', t']$ with $\alpha = x : \alpha'$ where $\alpha' \neq \lhd$, and $txn(\alpha) \neq txn(\beta)$ then there exists a sequence of labels σ s. t. $\sigma^- = (\beta\alpha)^-$ and $[P, t] \xrightarrow{\sigma} [P', t']$.*

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 $[P, t]$ be a configuration and $[P, t] \xrightarrow{\sigma_1} [P', t']$ be a transition sequence such that $actxn(P) = \emptyset$.*

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

3. ADDING TEST FOR ABSENCE

In this section we extend the previous calculus with two further coordination primitives $read\exists$ and $take\exists$ which are variants of the *read* and *take* operations which additionally 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.

We start by discussing some problems related to serializability which typically occur when test for absence operations are taken into account. Consider

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

representing a state in which a datum is required to be consumed within a transaction and tested for absence outside that transaction. Consider now the following computation: $\langle a\rangle$ is consumed from within the transaction; subsequently, the test for absence outside the transaction is performed; 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 instantaneous 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 presence of test for absence operations – which is not addressed in the JavaSpaces specifications – that has been pointed out in [2]. Consider the two concurrent programs

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

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

This computation is clearly non-serializable as 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, in [2] the following

Table 2: Modified axioms and rules (symmetric rules omitted).

$(1')$	$[take(a).P \mid \langle a\rangle_{\emptyset}^{t_a}, t] \xrightarrow{\tau} [P, t']$	$t \leq t'$
$(2')$	$[read(a).P \mid \langle a\rangle_{X}^{t_a}, t] \xrightarrow{\tau} [P \mid \langle a\rangle_{X}^{t_a}, t']$	$t \leq t'$
$(3')$	$[write(a, \Delta t).P, t] \xrightarrow{\vec{a}} [\langle a\rangle_{\emptyset}^{t'+\Delta t} \mid P, t']$	$t \leq t'$
$(4')$	$[create(x).P, t] \xrightarrow{y:\triangleright} [y\{P[y/x]\}_{\emptyset}^{\emptyset}, t']$	y fresh and $t \leq t'$
$(5')$	$[x : take(a).P\{Q\}_{T}^{R} \mid \langle a\rangle_{Y}^{t_a}, t] \xrightarrow{x:\tau} [x : P\{Q\}_{T}^{R\cup\{a\}}, t']$	$Y \subseteq \{x\}$ and $t \leq t'$
$(6')$	$[x : read(a).P\{Q\} \mid \langle a\rangle_{Y}^{t_a}, t] \xrightarrow{x:\tau} [x : P\{Q\} \mid \langle a\rangle_{Y\cup\{x\}}^{t_a}, t']$	$t \leq t'$
$(7')$	$[x\{\prod_i commit(x).P_i \mid \prod_j \langle a_j\rangle^{t_j}\}_{T}^{R} \mid \prod_h \langle b_h\rangle_{Y_h}^{t_h}, t] \xrightarrow{x:\bigoplus_j a_j \triangleleft}$ $[\prod_i P_i \mid \prod_j \langle a_j\rangle^{t_j} \mid \prod_h \langle b_h\rangle_{Y_h \setminus x}^{t_h}, t']$	$t \leq t'$
$(8')$	$[\langle a\rangle_{\emptyset}^{t_a}, t] \xrightarrow{-} [\mathbf{0}, t']$	$t \leq t'$ and $t' \not\leq t_a$
$(9')$	$\dfrac{[P, t] \xrightarrow{X:\alpha} [P', t']}{[P \mid Q, t] \xrightarrow{X:\alpha} [P' \mid Q, t']}$	$\alpha = \tau, \triangleright, -$
$(10')$	$\dfrac{[P, t] \xrightarrow{\alpha} [P', t']}{[x\{P\}_{T}^{R}, t] \xrightarrow{x:\alpha} [x\{P'\}_{T}^{R}, t']}$	$\alpha = \tau$
$(11')$	$\dfrac{[P, t] \xrightarrow{-} [P', t']}{[x\{P\}_{T}^{R}, t] \xrightarrow{-} [x\{P'\}_{T}^{R}, t']}$	
$(12')$	$\dfrac{[P, t] \xrightarrow{x:\triangleleft} [P', t']}{[P \mid Q, t] \xrightarrow{x:\triangleleft} [P' \mid Q, t']}$	$Data(Q) = \emptyset$

further locking mechanism is proposed: *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, thus solving the serializability problem. However, here we point out that the new locking mechanism is not sufficient when we consider leased data. Consider the following

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

representing a slight variation of the first example of this section, where a leased datum $\langle a\rangle^{t}$ is considered. Consider now the following computation: the datum $\langle a\rangle^{t}$ is read inside the transaction; subsequently it expires and it is removed; then the operation $read\exists(a)$ operation outside the transaction fails thus activating the $write(b)$ continuation; after, the datum $\langle b\rangle$ is first produced from outside the transaction and then consumed inside the transaction; finally, the transaction commits. This computation is clearly non-serializable because the unique way for the transaction to commit is to perform the test for absence and the output operation outside the transaction interleaved with the $read(a)$ and the $take(b)$ operations inside the transaction.

A possible solution to this kind of problem could be to add the following locking mechanism: *When read, a datum is added to the set of data read by the provided transaction. If such a datum expires before the end of the transaction, no test for absence on that kind of data could be executed before the end of the provided transaction.*

Observe that this new locking mechanism blocks the execution of the $read\exists(a)$ operation, thus disallowing the undesired computation described above. A drawback of this locking policy is that, when a datum read inside an active transaction expires, it is not possible to forget definitely about that datum because it is necessary to record the kind of data on which no test for absence operation could be executed. This contrasts with one of the main aims of the leasing mechanism, ie., to free the resources allocated to expired data.

Moreover, other problems concerning serializability occur due to the fact that the leasing mechanism introduces a temporal dependency between the withdrawal of two data which have two subsequent expiration times. Indeed, consider the system

$$\langle a\rangle^{t} \mid \langle b\rangle^{t'} \mid create(x).read(a).read(c).commit(x) \mid$$
$$read\exists(b)?\mathbf{0}\_write(c)$$

with $\langle a\rangle^{t}$ which expires before $\langle b\rangle^{t'}$ (ie. $t' \not\leq t$). Consider the following computation: the transaction starts and the $read(a)$ operation is executed; then $\langle a\rangle^{t}$ and $\langle b\rangle^{t'}$ expire in the expected order; the $read\exists(b)$ operation fails thus activating the continuation $write(c)$; the datum $\langle c\rangle$ is first produced, then tested for presence inside the transaction; finally, the transaction commits. This computation is clearly non-serializable because the unique way for the transaction to commit is to execute the two read operations inside the transaction interleaved with the test for absence and the write operations outside that transaction.

In order to solve this further problem of serializability, we could introduce a locking mechanism which is a stronger version of the previous one: *When read, a datum is added to the set of data read by the provided transaction. If such a datum expires before the end of the transaction, all the data which expire subsequently are recorded. No test for absence on the kind of data which have been recorded could be subsequently executed, before the end of the provided transaction.*

It is worth noting that this locking mechanism is very restrictive, because a read primitive performed on a specific kind of data, executed from within a transaction, could lock a huge amount of test for absence operations, even if performed on a different kind of data. Instead of formalizing this second stronger version of the locking mechanism, we propose a different approach for the modeling of temporary data alternative to leasing. This alternative approach has two main advantages: on the one hand, the first lighter version of the locking mechanism is sufficient; on the other hand, it takes advantage from the fact that the locking mechanism requires to remember the expired data read inside active transactions, by leaving these data in the repository.

The alternative approach has been introduced in [1] under the name of temporary data with unordered collection and it is based on a slightly different interpretation of the time to live Δt used in the $write(a, \Delta t)$ operation: it does not indicate a time after which the datum *must* be removed (as in the leasing approach of JavaSpaces), but it denotes a time after which the datum *could* be removed if required, eg., in order to free resources. Under this alternative interpretation, data could remain available even after expiration. Moreover, it is not necessary to remove the data following their order of expiration, but a datum could be selected for withdrawal even if expired after another one which is still available. An advantage of this approach is that, when new resources are needed, it is possible to select for withdrawal exactly the datum which, among all the expired ones, better fits the actual requirements.

3.1 The full calculus

The syntax of the calculus is extended as follows: the $read\exists$ and $take\exists$ operations are introduced as guards for programs with two possible continuations:

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

where:

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

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 terms:

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

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:

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

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 2 (which are the new versions for those reported in Table 1), plus rule (13) of Table 1, and the new axioms and rules for the test for absence operations reported in Table 3.

The rules (20) and (22) 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:

$$Rem(x\{P\}_T^R) = Rem(x:C\{P\}_T^R) = R$$
$$Rem(P|Q) = Rem(P) \cup Rem(Q)$$
$$Rem(C) = Data(\langle a\rangle_X^t) = \emptyset$$
$$Tfa(x\{P\}_T^R) = Tfa(x:C\{P\}_T^R) = T$$
$$Tfa(P|Q) = Tfa(P) \cup Tfa(Q)$$
$$Tfa(C) = Tfa(\langle a\rangle_X^t) = \emptyset$$

The new axiom $(1')$ is different from the previous version because the side condition $t' \leq t$ is removed: this reflects the new interpretation of temporary data according to which data could remain available even after expiration. This difference involves also axioms (2'), (4'), (5'), and (6').

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 $\{a_j\}$). The axiom $(8')$ introduces the new policy for expired data collection; observe that the subscript set of transactions should be empty. This reflects the fact that under the new locking mechanism a datum, even if expired, should be maintained in the system until there exists no active transaction under which the datum has been read. The rules $(9')\dots(12')$ are simple adaptation of the corresponding rules; the main difference can be found in (9') and (12') where the auxiliary function $P\backslash t$ is no more used because it is no more necessary to remove all the expired data in the configuration.

Axioms (14) and (15) 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 (16) and (17). Axioms (18) and (19) are adaptations

Table 3: The new axioms and rules (symmetric rules omitted).

$$(14)\quad [\mathit{take}\exists(a)?P\_Q|\langle a\rangle^{t_a}_{\emptyset}, t] \xrightarrow{\tau} [P, t'] \qquad t \leq t'$$

$$(15)\quad [\mathit{read}\exists(a)?P\_Q|\langle a\rangle^{t_a}_{X}, t] \xrightarrow{\tau} [P|\langle a\rangle^{t_a}_{X}, t'] \qquad t \leq t'$$

$$(16)\quad [\mathit{take}\exists(a)?P\_Q, t] \xrightarrow{\neg a} [Q, t'] \qquad t \leq t'$$

$$(17)\quad [\mathit{read}\exists(a)?P\_Q, t] \xrightarrow{\neg a} [Q, t'] \qquad t \leq t'$$

$$(18)\quad [x:\mathit{take}\exists(a).P\_Q\{R\}^{R}_{T}|\langle a\rangle^{t_a}_{Y}, t] \xrightarrow{x:\tau} [x:P\{R\}^{R\cup\{a\}}_{T}, t'] \qquad Y \subseteq \{x\} \text{ and } t \leq t'$$

$$(19)\quad [x:\mathit{read}\exists(a).P\_Q\{R\}^{R}_{T}|\langle a\rangle_{Y}, t] \xrightarrow{x:\tau} [x:P\{R\}^{R}_{T}|\langle a\rangle^{t_a}_{Y\cup\{x\}}, t'] \qquad t \leq t'$$

$$(20)\quad \frac{[P, t] \xrightarrow{X:\neg a} [P', t']}{[P|Q, t] \xrightarrow{X:\neg a} [P'|Q, t']} \qquad a \notin \mathit{Data}(Q) \cup \mathit{Rem}(Q)$$

$$(21)\quad \frac{[P, t] \xrightarrow{\neg a} [P', t']}{[x\{P\}^{R}_{T}, t] \xrightarrow{x:\neg a} [x\{P'\}^{R}_{T\cup\{a\}}, t']}$$

$$(22)\quad \frac{[P, t] \xrightarrow{x:A\lhd} [P', t']}{[P|Q, t] \xrightarrow{x:A\lhd} [P'|Q, t']} \qquad \mathit{Data}(Q) = \emptyset \text{ and } A \cap \mathit{Tfa}(Q) = \emptyset$$

$$(23)\quad \frac{[P, t] \xrightarrow{\vec{a}} [P', t']}{[P|Q, t] \xrightarrow{\vec{a}} [P'|Q, t']} \qquad a \notin \mathit{Tfa}(Q)$$

$$(24)\quad \frac{[P, t] \xrightarrow{\vec{a}} [P', t']}{[x\{P\}^{R}_{T}, t] \xrightarrow{x:\tau} [x\{P'\}^{R}_{T}, t']}$$

of (14) and (15) to the case in which the operations are executed inside a transaction; in (18) the set of data removed inside the transaction is updated, while in (19) 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 (20)). 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 (21)). According to rule (22) 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 $\mathit{Data}(Q) = \emptyset$ 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 (23) 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, the output operation can be performed if executed inside a transaction (rule (24)).

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. CONCLUSION

In this work we tackled a problem with the serializability of transactions in shared dataspace languages with leased data. We obtained a serializable semantics by moving to a weaker data removal policy, consisting in leaving expired data available for consumption or test for presence until they are not effectively removed by the collector. To simplify the locking mechanism, we chose the unordered collection policy, permitting to freely choose the expired datum to remove, provided that it has not been read by an active transaction.

A final remark concerns the event notification mechanism of JavaSpaces: in [2] the interplay of transactions and event notification is studied. The interplay among transactions, temporary data and event notification will be investigated in future work.

5. REFERENCES

[1] N. Busi, R. Gorrieri, and G. Zavattaro. Temporary Data in Shared Dataspace Coordination Languages. In *Proc. of FOSSACS'01*, volume 2030 of *Lecture Notes in Computer Science*, pages 121–136. Springer-Verlag, Berlin, 2001.

[2] N. Busi and G. Zavattaro. On the Serializability of Transactions in JavaSpaces. In *Proc. of ConCoord'01*, volume 54 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2001.

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

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

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

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

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