UML-SPACES: A UML Profile for Distributed Systems Coordinated Via Tuple Spaces

E. Astesiano - G. Reggio
DISI Università di Genova - Italy
astes,reggio@disi.unige.it

Abstract

Coordination via tuple spaces is a well known and accepted technique for designing distributed systems; originally introduced in Linda, it has been very recently adopted within the Java environment as underlying mechanism of the JavaSpaces$^{\mathrm{TM}}$ technology.

Here we explore the possibility of using UML in the development of systems using such technique. While tuple spaces are modelled quite naturally in UML as particular passive classes and objects, state machines do not seem adequate for modelling active objects interacting via tuple spaces. A solution is to use the UML lightweight extension mechanism to propose an appropriate extension of state machines by adding transition triggers corresponding to successful and failed execution of lookup operations on a tuple space.

The result is a UML profile, and thus a visual notation, for modelling autonomous distributed systems based on tuple spaces.

Keywords: UML - Autonomous distributed systems - Coordination - Tuple spaces

1 Introduction

The Unified Modeling Language (UML) [13, 11] is an industry standard visual notation largely used for specifying software systems. This language is unique and important for several reasons:

- UML is an amalgamation of several, in the past competing, notations for object-oriented modelling.
- Compared to other pragmatic modelling notations in Software Engineering, UML is very precisely defined and contains large portions that are similar to a formal specification language, as the OCL language used for the constraints.
- UML is quite flexible intended as extensible, adaptable, and modifiable. From the preface of the proceeding of last <<UML>> conference [5]:

  Flexibility is needed if the UML is to be used in a variety of application domains. Tailoring of UML syntax and adaption of UML semantics to system domains is highly desiderable. Incorporating domain-specific concepts into the language *will yeld modelling languages that more effectively support system development in these domains. Tayloring may involve determining a subset of the UML that is applicable to the domain, extending or modifying existing language elements, or defining new language elements.*

Furthermore, UML itself provides the mechanisms, summarized in [8], to define its various variants. UML Real Time [12] is an example of a variant for modelling embedded real-time systems.

Here, we want to extend UML for modelling systems made of distributed autonomous processes. This is a relevant task, because UML has not a natural provision for handling such systems, as pointed out also in [6].

At the moment, there is not a standard prominent mechanism for the design of distributed systems, so we have chosen to consider "coordination via tuple spaces", the mechanism originally introduced in Linda [3]. We think that the choice is sensible, because such mechanism is quite simple, easy to learn and to use, well-known and accepted; moreover it has been very recently adopted within the Java environment as the underlying mechanism of the JavaSpaces^{TM} technology, see [7].

A *tuple* is just a heterogeneous record, a *tuple type* is a record type, and a *tuple space*, or shortly *space*, is an abstract shared storage device containing tuples of the same type. Processes using a tuple space may *write* a tuple in such space (the tuple is added to the space). They can also look up tuples by using a pattern (or template), that is a tuple where some of the fields are specific values and the others are left as wildcards. There are two kinds of lookup operations: *read* and *take*. A read operation returns either a tuple that matches the pattern, i.e., the specified fields must be the same (successful execution), or an indication that a match was not found (failed execution). A take operation works as a read, but if a match is found the tuple is removed from the space.

We can sensibly model tuples and tuple spaces within UML by means of standard passive objects, whereas tuple types and space types are modelled by classes. Furthermore, we can slightly extend the UML notation, with some stereotypes of class and of association, to get a convenient way to describe such items.

At this point the proposed extension seems to concern only the class diagrams, but we have to consider what is its impact on the use of the most common kinds of UML diagrams, precisely sequence/collaboration and state chart (state machine). While such extension is quite compatible with sequence/collaboration diagrams, we have found problematic the use of state machines. The main reason is that such diagrams are intended for modelling the behaviour of reactive systems; see, e.g., [13] page 30 "*State machine may be used to describe user interface, device controllers, and other reactive subsystems*" and page 186 "A *class .... It may have a state machine that specifies its reactive behaviour – that is, its response to the reception of events.*". Processes using tuple spaces are not reactive because they behave autonomously by looking up and writing the tuple spaces.

Therefore, we need a different notation for modelling the behaviour of distributed (autonomous) processes using tuple spaces. We have devised two possible solutions:

- a quite lightweight extension of state machines, consisting of just adding new transition triggers corresponding to successful/failed executions of read and take operations on a tuple space;

- a heavyweight extension, introducing a new kind of diagram, *behaviour diagrams* originated from the labelled transition systems, to model in general the behaviour of processes (including autonomous, interactive aspects).

In this paper we present the first solution, while the second one, which is not specifically intended for processes using tuple spaces, will be presented in [2].

Technically, the new triggers are simply obtained by adding a new kind of events, *ghost events*, always available for dispatching, and so this extension is nicely integrated with the other features of the state machines, and it will be easy to update tools supporting them.

The UML variant for distributed systems proposed in this paper can be precisely presented as a UML profile [8], that we name UML-SPACES.

Here we present UML-SPACES using a simple case study: the invoice system (taken from [1]).

> *The problem under study is an information system for a company selling products to clients to support the management decisions. The clients send orders to the company, where an order contains one and only one reference to an ordered product in a given quantity. The status of an order will be changed from "pending" to "invoiced" if the ordered quantity is less or equal to the quantity of the corresponding referenced product in stock. New orders may arrive, others may be cancelled, and there may be some arrivals of products in some quantity in the stock.*

2 Tuples and Tuple Spaces in UML

Tuples are particular data types, roughly corresponding to records, and so we can model them in UML by using the DataType element (a special kind of classes).

A tuple space is a kind of abstract shared storage device characterized by a precise structure and a precise way to access it, storing tuples all of the same type. Thus, we can model it, quite naturally, using UML as a passive object.

However, we think that it is convenient to introduce appropriate stereotypes (of DataType and of Class respectively) for such structures, at least to get an appropriate clear and concise notation.

In the following two subsections we present our stereotypes for tuples and for tuple spaces.

2.1 Tuples in UML

We think to model a tuple datatype with n fields, whose types are $T_1, \ldots, T_n$ respectively, as a DataType classifier with n attributes of types $T_1, \ldots, T_n$ respectively.

However, to accommodate the tuple patterns (used by the read and take operations) the value of a field of a tuple may be also the special "wild card" (denoted by '_'). Thus, the types of the attributes of the tuple DataType will be $\mathrm{W}(T_1), \ldots, \mathrm{W}(T_n)$, where for a given type T, $\mathrm{W}(T)$ roughly corresponds to $T \cup \{\_\}$.

The main operation of the tuple DataType is match, a predicate checking whether a normal tuple matches a tuple pattern.

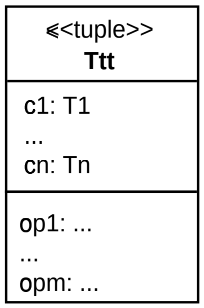


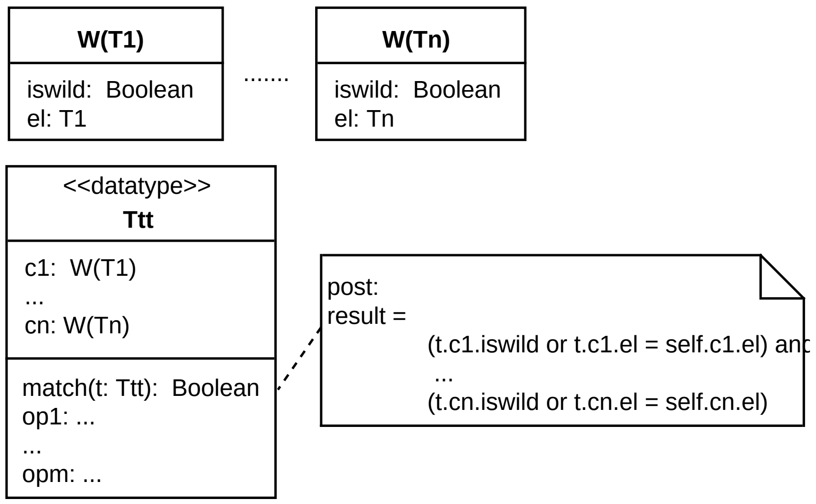
Figure 1: The stereotype <<tuple>>

The stereotype of DataType corresponding to tuple types is named <<tuple>>, and the corresponding notation is given in Fig. 1, where: n ≥ 1, c1, ..., cn are distinct names, T1, ..., Tn are UML types, m ≥ 0, op1, ..., opm are distinct names and all different from match.

Recall that all operations on tuples are queries[1], because in UML all operations of a DataType must be queries.

Notice that constraints, as invariants, may be freely attached to a tuple classifier.

The semantics of the tuple stereotype is simply given by translating it into standard basic UML. An instance of a tuple classifier, as the one in Fig. 1, corresponds to the following UML class diagram fragment.



2.2 Tuple Spaces in UML

We model a (type of) tuple space in UML as a passive class having an attribute of type set of some given tuple type, and the read, take, and write operations; the latter does not return a result, while the first twos return either a tuple or a special value to denote failure.

We introduce a stereotype of Class for the tuple space types <<space>>, and a stereotype of Association <<of>> for the associations connecting tuple spaces with the corresponding tuple types. The notation for such two stereotypes is presented in Fig. 2.

[1]In UML queries are operations without side effects, i.e., they are pure functions.

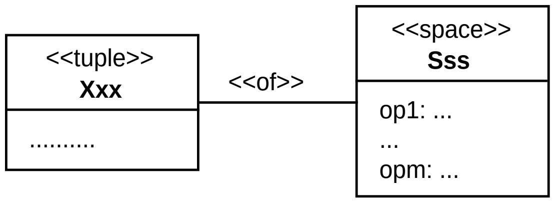


Figure 2: The stereotypes <<space>> and <<of>>

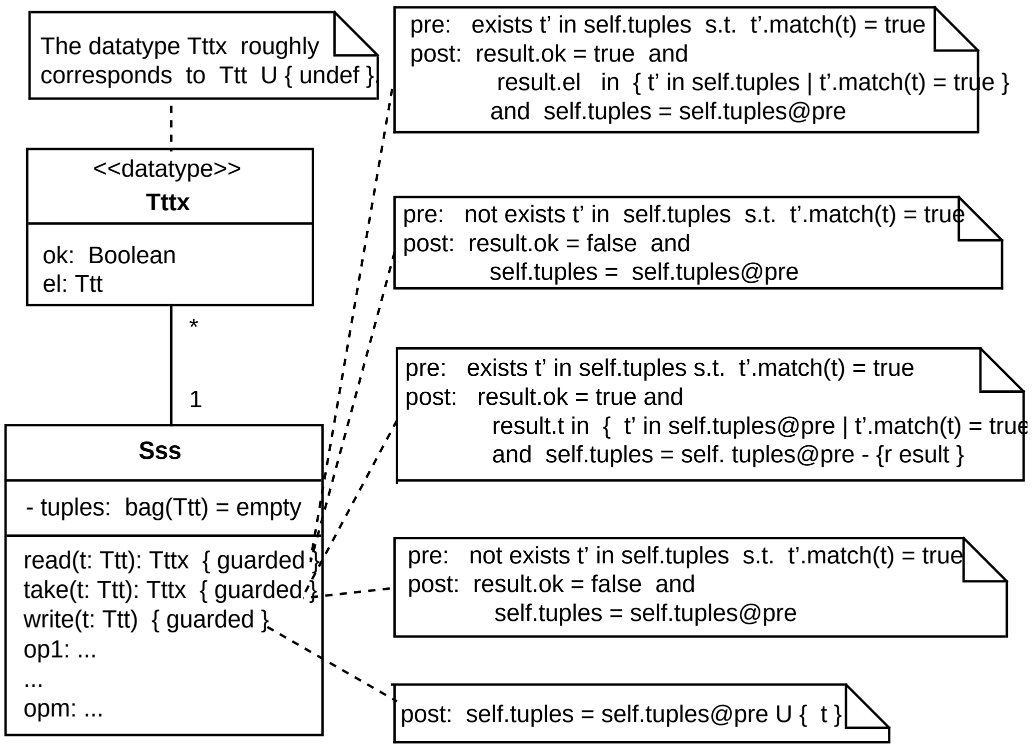
A space classifier must be always connected to one and only one tuple classifier by means of an instance of <<of>>, moreover it cannot have any attribute.

The "of" association cannot

- have a name,
- be navigable in any versus,
- have named rolenames,
- and have any multiplicity constraint.

Notice that constraints, as invariants, may be freely attached to a space classifier

A space classifier, with the connected tuple classifier, as the one in Fig. 2, corresponds to the following UML class diagram fragment.



The datatype Tttx roughly corresponds to Ttt U { undef }
<<datatype>>
Tttx
ok: Boolean
el: Ttt
*
1
Sss
- tuples: bag(Ttt) = empty
read(t: Ttt): Tttx { guarded }
take(t: Ttt): Tttx { guarded }
write(t: Ttt) { guarded }
op1: ...
...
opm: ...
pre: exists t' in self.tuples s.t. t'.match(t) = true
post: result.ok = true and
result.el in { t' in self.tuples | t'.match(t) = true }
and self.tuples = self.tuples@pre
pre: not exists t' in self.tuples s.t. t'.match(t) = true
post: result.ok = false and
self.tuples = self.tuples@pre
pre: exists t' in self.tuples s.t. t'.match(t) = true
post: result.ok = true and
result.t in { t' in self.tuples@pre | t'.match(t) = true
and self.tuples = self. tuples@pre - {r esult }
pre: not exists t' in self.tuples s.t. t'.match(t) = true
post: result.ok = false and
self.tuples = self.tuples@pre
post: self.tuples = self.tuples@pre U { t }

The meaning of the UML guarded annotation of an operation of a class, here used on read, take and write, is ambiguous. Indeed, [13] says that it means:

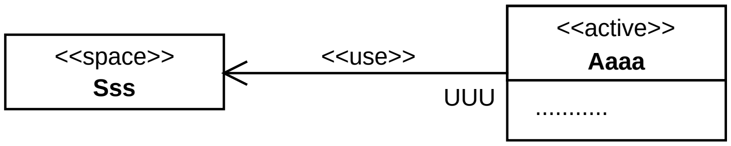


Figure 3: The stereotype <<`use`>>

> *Multiple calls from concurrent threads may occur simultaneously to one Instance (on any guarded Operation), but only one is allowed to commence. The others are blocked until the performance of the first Operation is complete....*

It is ambiguous because "Multiple calls" may be interpreted either as

- of the same guarded operation,
- or of all the guarded operations of the class.

Our realization of the tuple spaces works only with the second interpretation, so our stereotype enforces such semantic interpretation.

The operations for accessing a tuple space (`read`/`take`/`write`) can be called only in a synchronous way.

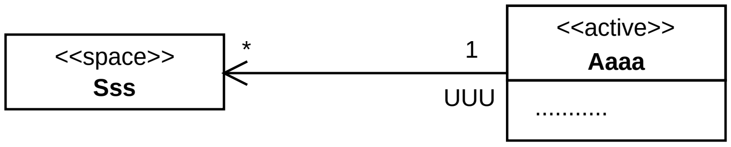
Whenever the type of `sp` is a tuple space, we abbreviate `sp.tuples` into `sp`.

To describe the structure (architecture) of a distributed system it is useful to say which processes use which tuple spaces. Thus, we introduce a proper stereotype of Association: <<`use`>>. The corresponding notation is shown in Fig. 3.

A use association is binary, must connect a space classifier and an active class (and the navigation is allowed only from the active class to the space), cannot have a name, the role corresponding to the active class must be named, whereas the other cannot be named.

No multiplicity can be attached to any end of such associations, and implicitly we assume that is `1` on the active class end and `*` on the other end. It is possible to have different "use" associations between an active class and a tuple space classifier (obviously the role names should be different). Thus, an active object may use different instance of the same space classifier, but each one will be denoted by a different role name.

A use association, as the one in Fig. 3, corresponds to the following association.



Example As example we show in Fig. 4 the class diagram for the invoice system, which consists of active objects of three different types (handler of the messages from the clients, order invoicer and warehouse handler) and of two tuple spaces (one storing the orders and the other the information about the products in the warehouse). In this diagram we have constraints attached to the tuple spaces, e.g., requiring that all the tuples describing the orders have different dates (the date includes also the hour), and a global constraint requiring that there is exactly one instance of some classes.

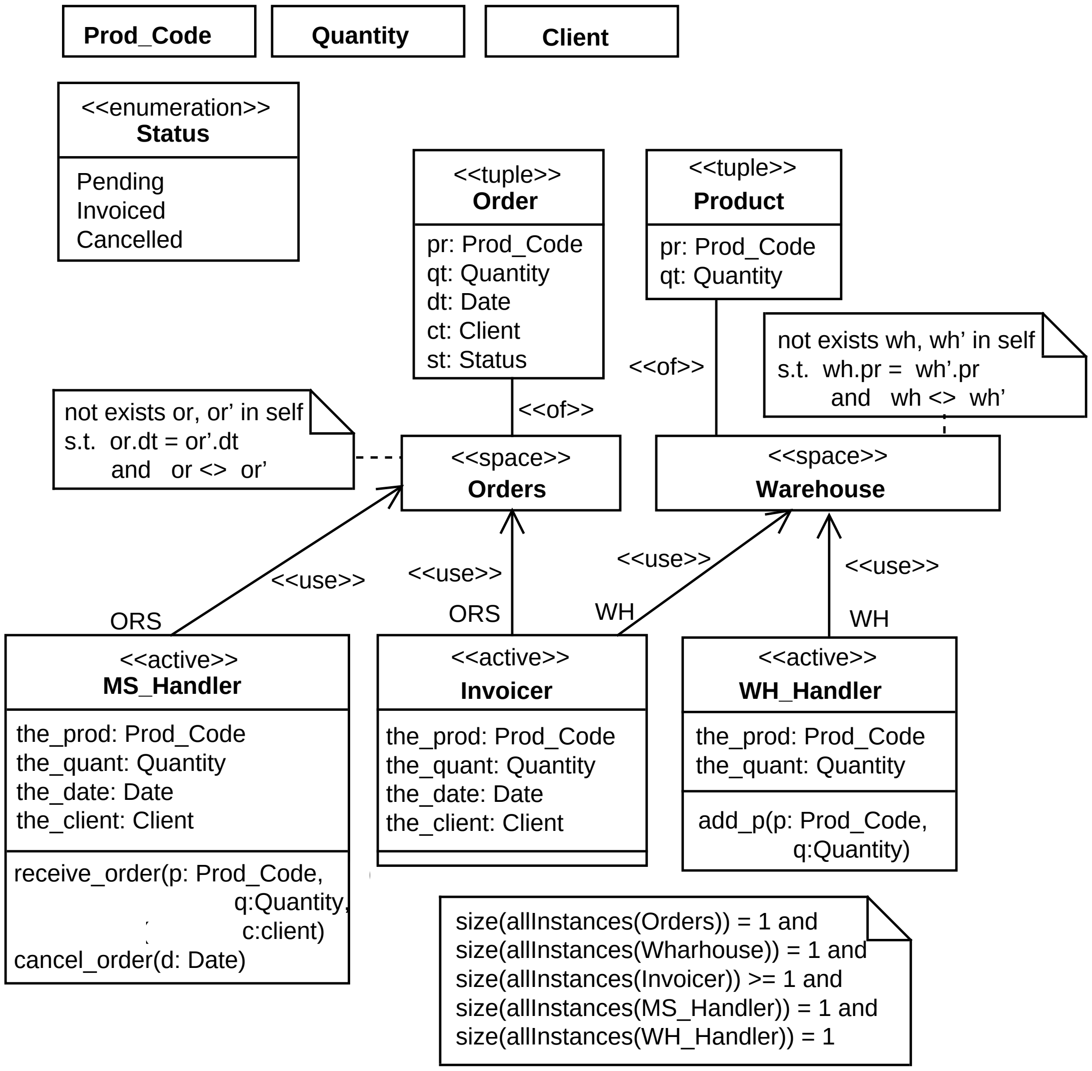


Figure 4: The class diagram of the invoice system

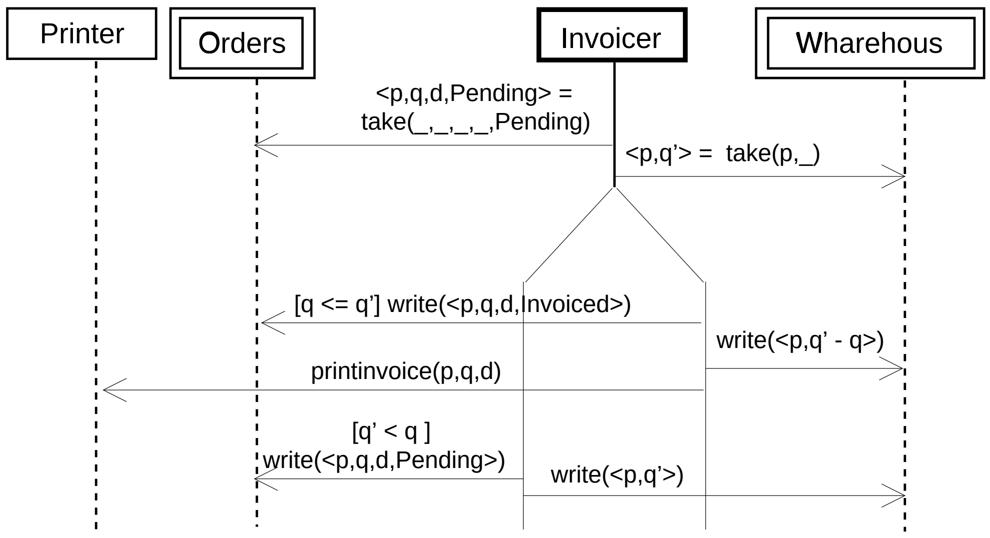

Printer
Orders
Invoicer
Wharehous
<p,q,d,Pending> = take(_,_,_,_,Pending)
<p,q'> = take(p,_)
[q <= q'] write(<p,q,d,Invoiced>)
write(<p,q' - q>)
printinvoice(p,q,d)
[q' < q] write(<p,q,d,Pending>)
write(<p,q'>)

Figure 5: A sequence diagram for the invoice system

3 Using Tuple Spaces within UML

In Sect. 2 we have shown how to conveniently model tuple spaces using UML. To be sure that such extension allows us to properly model distributed systems coordinated via tuple spaces using UML, we still need to investigate the impact of the introduced stereotypes on the use of the other kinds of UML diagrams. Here, we consider only those diagrams that we think are the most relevant for distributed systems, precisely class diagrams, sequence/collaboration diagrams and state machines.

Class diagrams, obviously, do not pose any problems; see, e.g., the example in Fig. 4. Also sequence and collaboration diagrams work finely with tuple spaces; indeed we can use the new classifiers (tuples and tuple spaces), the new associations (of and use) and the operations for accessing the tuple spaces (read/write/take) in a sequence/collaboration diagram without problems, because such diagrams roughly depict the execution of operation calls among objects. In Fig. 5 we report an example of a sequence diagram for the invoice system. It shows how the invoicer handles the orders: it takes a pending order (if any) from the space Orders, then the tuple of the ordered product from the space Warehouse; if the quantity is sufficient, then the order is invoiced and the reduced quantity is written back in Warehouse, otherwise the two taken tuples are written back in the corresponding spaces.

3.1 State Machines

We have found much more problematic the use of state machines to describe the behaviour of active objects using tuple spaces (clearly it is not sensible to associate state machines with tuple spaces). The reason is that the operations for accessing a tuple space must be called by the active objects and so they may appear only in the action part of the machine transitions, and cannot generate events to trigger such transitions.

Consider for instance the Invoicer active class in the running example. Such class has no operations and so for modelling its behaviour with a state machine we can use only the timed

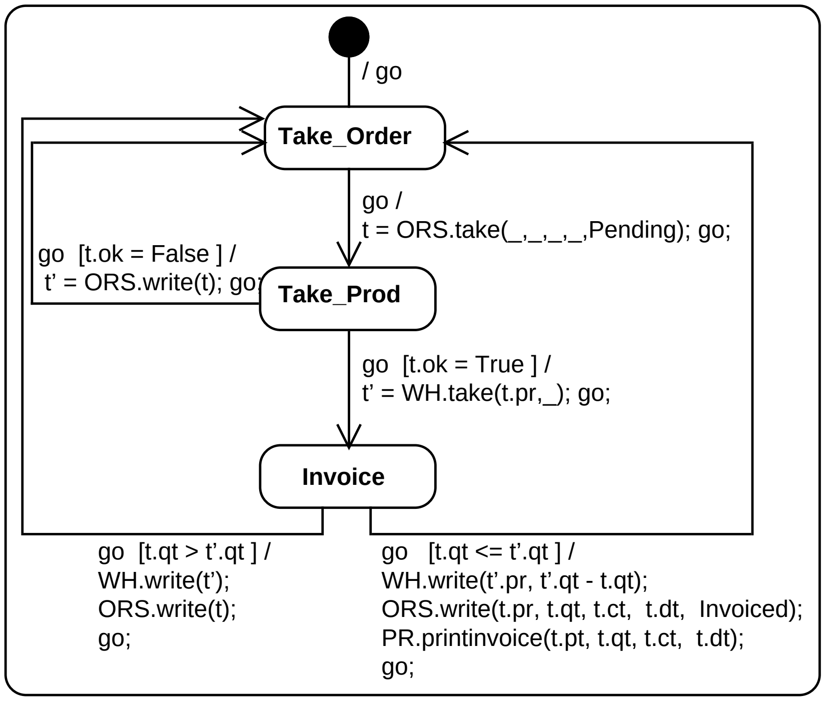

/ go
Take_Order
go /
t = ORS.take(_,_,_,_,Pending); go;
go [t.ok = False] /
t' = ORS.write(t); go;
Take_Prod
go [t.ok = True] /
t' = WH.take(t.pr,_); go;
Invoice
go [t.qt > t'.qt] /
WH.write(t');
ORS.write(t);
go;
go [t.qt <= t'.qt] /
WH.write(t'.pr, t'.qt - t.qt);
ORS.write(t.pr, t.qt, t.ct, t.dt, Invoiced);
PR.printinvoice(t.pt, t.qt, t.ct, t.dt);
go;

Figure 6: A unacceptable modelization of the INVOICER's behaviour

and change events. However, interesting change events, i.e., detecting changes in the used tuple spaces, should use attributes of other objects, and so they are of the kind that should not be used as stated by the official UML documentation. Furthermore, even if allowed, their use in a distributed setting may make the system quite centralized.

The Invoicer should behave in this way:

- take a pending order, if any, from the ORS space,
- take the tuple corresponding to the ordered product form the WH space,
- if the quantity in the warehouse is enough, then send the invoice and put back the two tuples properly updated (the order will be invoiced and the product quantity will be reduced), otherwise put back the two tuples unchanged.

It seems to us that the only way to model it using standard UML state machines is as reported in Fig. 6. Notice that we have added to the class Invoicer an auxiliary operation go without arguments and returning anything, add two auxiliary attributes t and t' of type tuple.

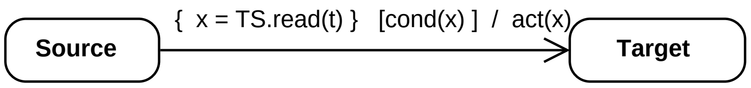
Clearly this solution is not acceptable. The fundamental reason for this problem is the fact that state machines are intended for describing the behaviour of reactive systems, as stated by [11] page 30 and page 186, while processes interacting via shared tuple spaces are not reactive, but autonomously interacting by reading and writing shared memory. Thus, we need an extension of UML for modelling in general the behaviour of active objects, and not focused on reactivity, at least in this particular case.

In the following section we propose a "lightweight" extension, where the only addition is a new form of trigger for the state machine transitions. In [2] instead we will propose a "heavyweight" extension adding a new kind of diagrams, "behaviour" diagrams, to describe in the general case the behaviour of active objects, which can also very conveniently be used for distributed processes interacting via shared tuple spaces.

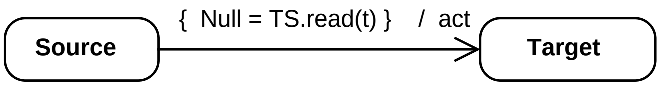
4 Lightweight Extension

To conveniently describe the behaviour of processes using shared tuple spaces with the UML state machine, we need transitions that are triggered by the failed or successful execution of either a read or a take operation on a tuple space. Moreover, in the successful case the returned tuple should be assigned to some variable that must be available in the condition and action part of the transition itself. We propose to extend UML state machines to include the following four new forms of transitions (corresponding to the new triggers):

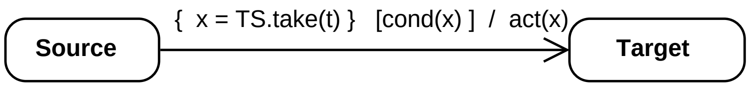
successful reading



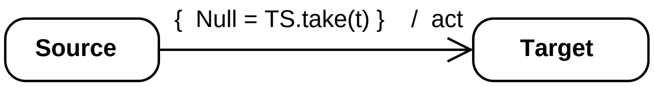
failed reading



successful taking



failed taking



where TS is an expression of type Sss (a space of tuples of type Ttt), x is a variable of type Ttt and t an expression of the same type. cond(x) (act(x)) means that x may freely appear in such condition (action).

The intuitive semantics of such transitions is that they can be fired whenever the execution of the corresponding operation on the tuple space TS is

- successful and returns a tuple that will be assigned to x;
- not successful;
- successful and returns a tuple that will be assigned to x;
- not successful.

Now we have to integrate such intuition with the semantics of the other transitions of the state machines, that is based on the run-to-completion step started by dispatching events.

Unfortunately, we cannot consider the new triggers as standard events. Indeed, events are either (a) generated by other objects (call and signal) or (b) correspond to the happening of some facts (change and time) checked by the object itself. In case (a) the tuple space should go on generating events by calling the objects that use it, whereas, in case (b), the object should go on looking up the tuple space, and that destroys the distributed nature of the modelled systems.

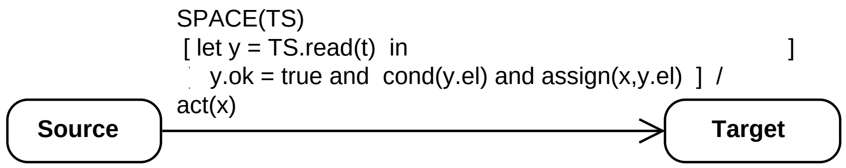
To keep it, the read and take operations have to be performed only when it is required by a user of the tuple space.

But the semantics of the UML state machine is described in terms of events, event queues, run-to-completion-steps started by dispatching an event, and so if we want that our extension is really “light” we cannot describe the semantic of the new transitions in a completely different way.

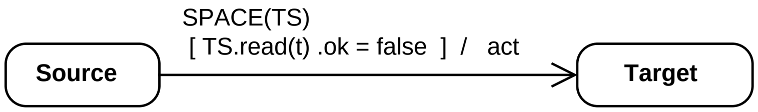
Our solution is to introduce for each pair: active class ACL and used space tuple TS, one *special* event, SPACE(TS), that it is always available to be dispatched from the event queue of the objects of ACL. We name such events *ghost event*, since they float around without the need to be explicitly generated.

Then, using the ghost events, the above new transitions are simply defined as follows:

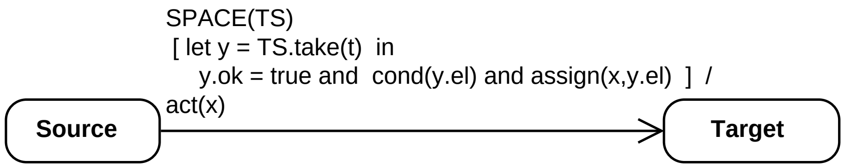
successful reading


SPACE(TS)
[let y = TS.read(t) in]
y.ok = true and cond(y.el) and assign(x,y.el)] /
act(x)
Source
Target

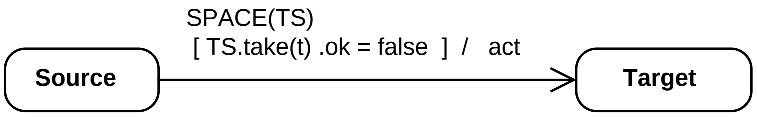
failed reading



successful taking



failed taking



where x is an auxiliary attribute, and assign a boolean auxiliary method assigning a value to an attribute returning always the true value (they are implicitly declared).

Then the usual UML semantics of the state machines completely describes also the semantics of the new transitions.

When a special ghost event, say SPACE(TS), is dispatched (this is always possible, since it is always in the queue ready to be dispatched), it may trigger one of the above transitions.

In the first case, then the condition is evaluated requiring to perform the calls of TS.read(t) and of assign(x,y). If the returned tuple is ok and cond holds on the returned tuple, then the transition is fired, its action part is executed using x (the evaluation of assign has assigned to it the returned tuple), and the machine moves to the final state of the transition. Otherwise, the transition is not fired.

Similarly, in the other cases.

The semantics and the static correctness conditions concerning the ghost events are modified in the following way.

- The condition part of a transition triggered by the ghost events may have the side effects (due to the calls of take and of assign).

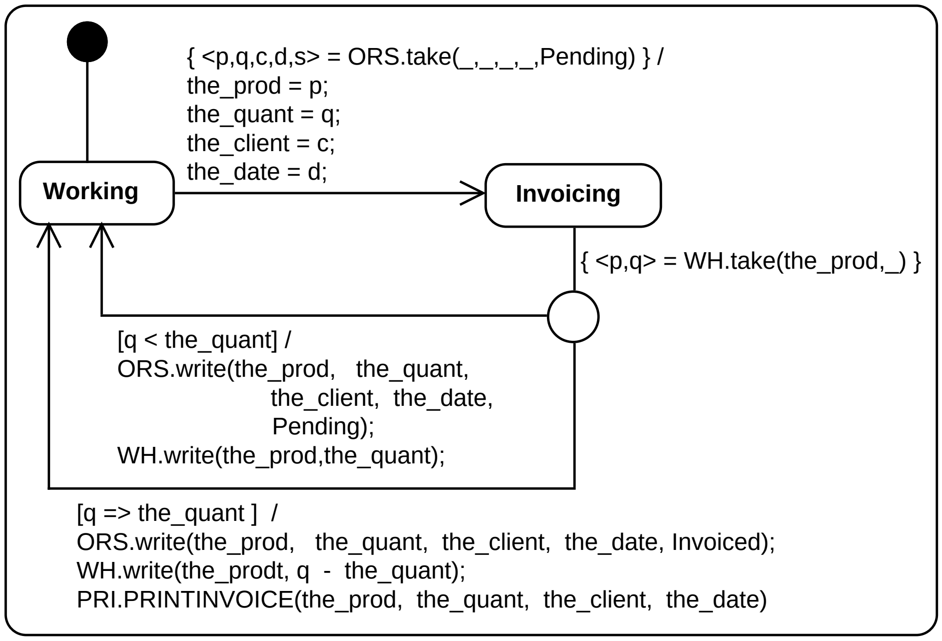

{ <p,q,c,d,s> = ORS.take(_,_,_,_,Pending) } /
the_prod = p;
the_quant = q;
the_client = c;
the_date = d;
Working
Invoicing
{ <p,q> = WH.take(the_prod,_) }
[q < the_quant] /
ORS.write(the_prod, the_quant,
the_client, the_date,
Pending);
WH.write(the_prod,the_quant);
[q => the_quant] /
ORS.write(the_prod, the_quant, the_client, the_date, Invoiced);
WH.write(the_prodt, q - the_quant);
PRI.PRINTINVOICE(the_prod, the_quant, the_client, the_date)

Figure 7: Extended state machine modelling the INVOICER'behaviour

- The ghost events can be used only in the above four cases.
- The ghost events are always ready to be dispatched in the event queue.
- The queue handling policy must be fair w.r.t. the picking of the ghost events, i.e., it cannot happen neither that ghost events are dispatched forever nor that non-ghost events are picked forever.

Examples

We present in Fig. 7, 8 and 9 the extended state machines modelling the behaviour of the objects of the three active classes present in the class diagram of the invoice system of Fig. 4. Here we use also another notational shortcut. If x1, ..., xn are the fields of the considered tuple type,

{ < x1, ..., xn > = UUU.read(t) } [cond(x1, ...,xn)] / act(x1, ..., xn)

stands for

{ t = UUU.read(t) } [cond(t.x1, ...,t.xn)] / act(t.x1, ..., t.xn) .

5 Conclusions

In this paper we have presented UML-SPACES, a UML profile [8], for the specification of distributed systems made by autonomous processes coordinated via tuple spaces. We think that this may be a significant addition to the existing "tools" for the development of reliable software systems, since UML does not offer special provisions for modelling distributed systems, while it is becoming one of the most used notation for software systems.

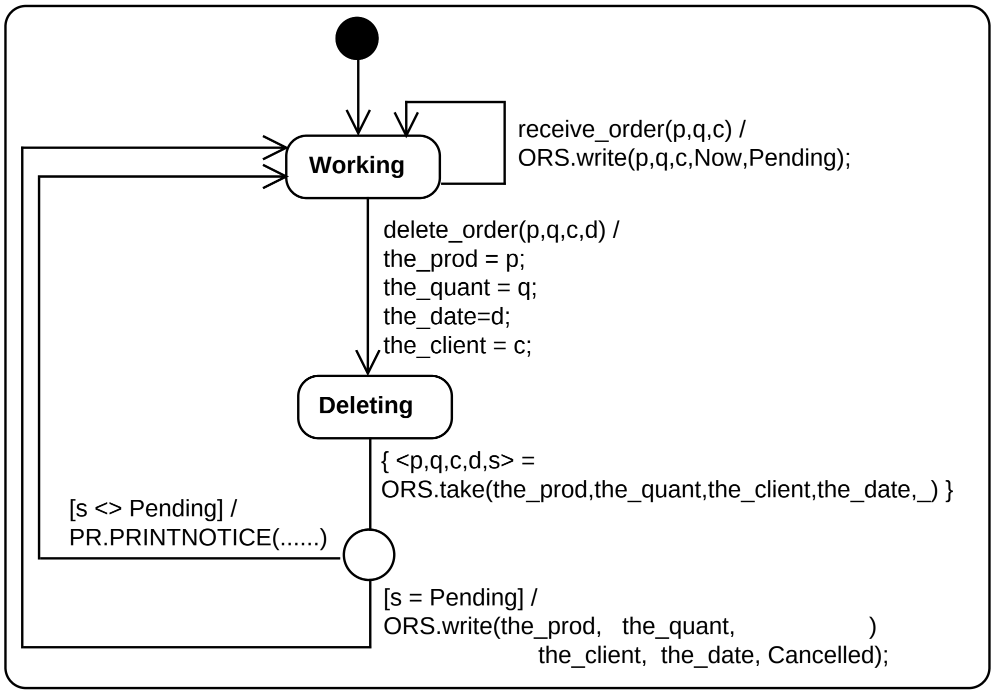


Figure 8: Extended state machine modelling the MS_HANDLER'behaviour

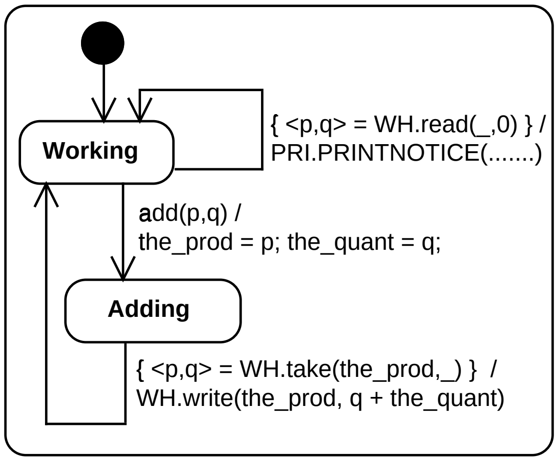

Working
{ <p,q> = WH.read(_,0) } /
PRI.PRINTNOTICE(.......)
add(p,q) /
the_prod = p; the_quant = q;
Adding
{ <p,q> = WH.take(the_prod,_) } /
WH.write(the_prod, q + the_quant)

Figure 9: Extended state machine modelling the WH_HANDLER'behaviour

Indeed, other attempts to extend UML for distribution are present in the literature; for example [6] incorporates in UML a particular technique for developing distributed systems (meta object protocol).

Instead, UML-SPACES is a quite light extension of UML; we have defined it just by adding a few new classifiers for tuple types, tuple spaces and related associations (stereotypes of DataType, Class and Association), and a new form of transition for the state machines (which can be triggered by the failed or successful execution of a lookup operation on a tuple space). We have shown that to introduce such transition it is sufficient to extend the Event metaclass with a new kind of event (ghost events).

We have designed UML-SPACES trying to modify UML as less as possible; thus it is easy to extend to the case of distributed systems methods, tools, techniques based on UML: we just need to add some static constraint and modify the function corresponding to dispatching events from the queue. Moreover, since tuple spaces are supported by Java technology [7], it is possible to develop Java code generation tools for UML-SPACES.

We can also give a formal precise definition of UML-SPACES by very slightly modifying the formal semantics of state machine in [9].

Here we have presented only the most basic constructs, but we can add also more refined operation for accessing tuple spaces, as

- read (take) all, returning all tuples matching a pattern,
- write all (writing a set of tuples),
- delete, deleting all the tuples matching a pattern.

JavaSpaces offers also additional features that could be added also to UML-SPACES. For example, the annotation "transaction" for the methods of a tuple space, characterized by the fact that when they start to execute no other process can access the space till the end of such execution; and also new change events raised in the processes using a tuple space whenever a tuple matching a pattern is added/removed from the space. These events do not require the continuous polling of the space by the processes, it is sufficient that the space calls the users when a tuple is added/removed.

Developing UML-SPACES we have noted how the state machines are mainly intended for modelling reactive processes, and that can be less convenient in the case of general active objects. Thus, we have also designed a heavyweight extension, see [2], introducing a new kind of diagram for modelling the behaviour of active objects: *behaviour diagrams*, inspired by labelled transition systems, and already used in some visual formal notations ([10] and [4]).

We know that in the case of distributed systems it is also relevant to model the structure (architecture) of the system at the level of the instances; we plan to investigate what does UML offer in this regard, trying to understand the role of object diagrams (not very well clarified in UML) and of the deployment diagrams.

References

[1] M. Allemand, C. Attiogbe, and H. Habrias, editors. *Proc. of Int. Workshop "Comparing Specification Techniques: What Questions Are Prompted by Ones Particular Method of Specification". March 1998, Nantes (France).* IRIN - Universite de Nantes, 1998.

[2] E. Astesiano and G. Reggio. UML Behaviour Diagrams for Modelling Active Objects. Technical report, DISI, Università di Genova, Italy, 2000. In preparation.

[3] N. Carriero and D. Gelernter. How to Write Parallel Programs: A Guide to the Perplexed. *ACM Computing Surveys*, 1989.

[4] E. Coscia and G. Reggio. JTN: A Java-targeted graphic formal notation for reactive and concurrent systems. In Finance J.-P., editor, *Proc. FASE 99 - Fundamental Approaches to Software Engineering*, number 1577 in Lecture Notes in Computer Science. Springer Verlag, Berlin, 1999.

[5] R. France and B. Rumpe. <<UML>>'99 preface. In R. France and B. Rumpe, editors, *Proc. UML'99*, number 1723 in Lecture Notes in Computer Science. Springer Verlag, Berlin, 1999.

[6] J.S. Lee, T.H. kim, G.S. Yoon, J.E. Hong, S.D. Cha, and D.H. Bae. Developing Distributed Software Systems by Incorporating Meta-Object Protocol (*di*MOP) with Unified modelling Language (UML). In *Proc. 4th International Symposium on Autonomous Decemtralizzed Systems*, pages 65–72, 1999.

[7] Sun Microsystems. JavaSpaces Specification. Technical report, Sun, 1999.

[8] OMG. White paper on the Profile Mechanism – Version 1.0. `http://uml.shl.com/u2wg/default.htm`, 1999.

[9] G. Reggio, E. Astesiano, C. Choppy, and H. Hussmann. Analysing UML Active Classes and Associated State Machines – A Lightweight Formal Approach. In T. Maibaum, editor, *Proc. FASE 2000 - Fundamental Approaches to Software Engineering*, number 1783 in Lecture Notes in Computer Science. Springer Verlag, Berlin, 2000.

[10] G. Reggio and M. Larosa. A Graphic Notation for Formal Specifications of Dynamic Systems. In J. Fitzgerald and C.B. Jones, editors, *Proc. FME 97 - Industrial Applications and Strengthened Foundations of Formal Methods*, number 1313 in Lecture Notes in Computer Science. Springer Verlag, Berlin, 1997.

[11] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual.* Object Technology Series. Addison-Wesley, 1999.

[12] B. Selic and J. Rumbaugh. Using UML for Modeling Complex Real-Time Systems. Technical report, ObjecTime Limited, 1998.

[13] UML Revision Task Force. *OMG UML Specification*, 1999. Available at `http://uml.shl.com`.