Consistency and Distributed Sensor Data Processing

Laurent-Frédéric Ducreux, Suzanne Lesecq, François Pacull, Stéphanie Riché
CEA-LETI MINATEC Campus
17 rue des Martyrs, 38000 Grenoble, France
Email: surname.name@cea.fr

***Abstract*—This paper proposes a framework dedicated to the management of data processing within a geographically distributed system made of heterogeneous software and hardware components. The integration and coordination of these different components is easily performed thanks to a uniform abstraction level proposed here. The resource-oriented approach combined with a rule-based system allowing transactional manipulation of these resources provides a unified view of the distributed system. An example will show the powerfulness of the proposed middleware.**

***Keywords*-Middleware; rule-based language.**

I. Introduction

When dealing with distributed sensors data processing, from co-located sensors to sensors spread over different continents, the problem to solve at the application level always remains the same:

- to share data;
- to synchronize ”entities”;
- to manage concurrent accesses;
- to define conditions, and
- to ensure consistency on the system ”global state”.

Until now, each level was defining its own way to solve (or sometimes not!) its own issues, with different specific points of view and mechanisms.

The proposition in this paper is to provide a uniform abstraction layer that eases the integration and coordination of the different components (software and hardware) that compose the network of sensors and actuators. It is based on a resource-oriented approach combined with a rule-based system allowing transactional manipulation of these resources. This results in a unified view of the system, whatever its size and its geographical distribution.

Chaski is rooted in previous projects conducted at XRCE (Xerox Research Centre Europe), in particular the Coordination Language Facilities (CLF) project [1], [2] and STITCH [3] developed from 1995 to 2003. Chaski may be seen as a natural evolution, based on the lessons learned and the will to reduce the middleware footprint in order to better fit with a large number of small devices and appliances.

Chaski, is the combination of three paradigms, namely *associative memory*, *distributed transaction* and *production rules* presented in Section 2. Put together they offer a powerful framework to address distributed (sensor) data processing with properties that are up to our knowledge novel in this field. Section 3 presents the Chaski rule-based language, an intermediate coordination layer that can be used by application programmers or automatically generated from *Domain Specific Languages* or graphical user interfaces. This rule-based language offers in addition the possibility to dynamically modify the application to adapt not only to changes induced by the user, but also changes due to the self adaptation of the system to the context (both for planned modifications and failures). Section 4 demonstrates the Chaski expressiveness through an application.

II. Paradigms

A. Associative Memory

An abstraction layer is highly desirable to unify the view of the various components usually involved in distributed sensor processing: the sensors, the sensed data, the actuators, the software components responsible for the data fusion, the software components used to interact with the system either for application or monitoring purpose, etc.

The option chosen is to consider these entities as ”resources” managed in an *associative memory*. The associative memory is implemented with a tuple-space, following the concept introduced by Linda in the 80s [4] and resurrected several times since in the middleware field [5], [6]. An associative memory provides a repository of tuples that can be concurrently accessed by a set of software components that may either insert resources (data) as tuples in the space or retrieve or consume resources from the space according to a matching given pattern. This tuple-space may be seen as a distributed shared memory.

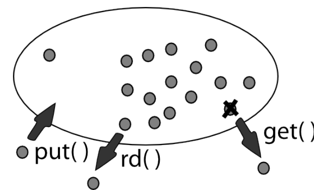

Figure 1. Associative memory and basic operations

From original Linda, only 3 operations are retained (see Figure 1) to manipulate tuples: `rd()` which verifies the presence of a tuple matching a given pattern, `get()` which consumes a tuple, and `put()` which inserts a tuple (the

Copyright (c) IARIA, 2011.

original Linda operations `in()` and `out()` have been renamed to make their semantic less ambiguous).

In Chaski, the associative memory is split into distinct units called *bags*. Unlike in the original Linda, bags may be distributed over different hosts, offering a fully distributed associative memory. Moreover, the bags are typed. Distinct types introduce specific differentiations in the semantic and operation of the `rd()`, `get()` and `put()` methods. For instance, the bag may be a *set*, or a *multiset* with an impact on the `put()` operation. When a resource is `put()` in a bag already containing an identical resource, then it is inserted when the bag is a *multiset* but inserted if it is a *set*.

The abstraction of bags completely hide the data distribution thus it matches the first problem that is how *to share data*. It is also well suited to encapsulate sensors and actuators. Actually, a sensor is a "virtual" bag in which the resources are the different sensed information that can be accessed one by one through the `rd()` and `get()` operations. An actuator is a "virtual" bag where each inserted resource through the `put()` may trigger actions.

B. Distributed transactions

A distributed transaction is a set of operations in which at least 2 geographically distributed clients are involved. Transactions usually ensure the so-called Atomicity, Consistency, Isolation, Durability (ACID) properties. In Chaski, only atomicity is managed, to ensure all-or-nothing outcomes for each unit of work (operations set). The atomicity is implemented using a two-phase commit protocol (2PC) [7] to ensure that each participant in the transaction agrees on whether the transaction should be committed or not. In the first phase, the participants involved in the transaction are queried. In the second phase, when all participants replied they are ready, the coordinator formalizes the transaction.

In order to allow transactional manipulation at the resources level, the `rd()`, `get()` and `put()` basic operations are divided into 2 phases. The first one prepares the operation and locks the resources. The second phase performs the operation if the transaction has to be committed or it unlocks the resources otherwise. Distributed transactions tackle two problems raised in the introduction, i.e. *"entities" synchronization* and *concurrent accesses management*.

C. Production rule system

A production rule consists in 2 parts: a precondition part and a performance/action part. If a precondition matches the current system state, then the action is performed. A production system can be seen as a mechanism that executes productions in order to achieve some goals. An example of production rule system is, e.g. OPS5 [8].

The production rules in Chaski use only the 3 basic operations defined on top of the associative memory. The system state is kept and managed through the associative memory (tuple-space). So, `rd()` operations are naturally used to access and evaluate this state. The precondition of a rule is a set of `rd()` operations. The distinct `rd()` operations required to evaluate the precondition are performed sequentially, with a right propagation following logic programming approach [9]. This will be developed in details in Section III. The performance phase of a rule is a set of `rd()`, `get()` and `put()` operations embedded in one or more transactions. Broadly speaking:

- `rd()` operations are used to ensure that the conditions which triggered the rule are still valid when the rule is actually executed. If the conditions are no longer valid, then the transaction is aborted. This ensures that a rule cannot be performed when conditions were fulfilled at some point in the past, but this is no longer the case;
- `get()` operations are used to validate the resources existence (e.g. resources involved in the definition of the precondition) but, in addition, to ensure the resources will be consumed when the transaction is committed. Therefore, if different transactions depend on the same resource availability, only one is committed;
- `put()` operations are used to insert new resources in the associative memory, as combinations of resources returned by `rd()` operations of the precondition phase.

When a performance phase contains multiple transactions, they are executed in sequential order. Then, if the distinct transactions perform a `get()` over the same "token" resource, it is very easy to formalize alternative treatments for a given precondition with the insurance that only the first possible treatment will be performed. This provides a natural and very elegant way to solve various difficult problems (e.g. graceful degradation, redundancy or dispatching) that arise in a distributed environment. The full picture of this mechanism is described in Section III.

The Chaski production rule system answers the two last problems, that is how *to verify distributed conditions* and *to ensure consistency on the "global state" of the "system"*.

III. Rule-based languages

One of the key features of Chaski relies on its coordination language which is an evolution of the STITCH language. In this section, the basis of the Chaski language are introduced.

A. Writing of Chaski rules

In a Chaski rule, the precondition and a performance parts are separated by the symbol "::".

The associative memory is split into bags. These bags may be grouped into objects according to the application design. For instance, bags may be grouped for location reasons (hosted by the same machine) or for semantic reasons (all the bags used to manage a given sensor network are grouped in the same object).

A bag is uniquely defined among the overall system with syntax `[<objectname>,<bagname>]` where `<objectname>` and `<bagname>` respectively define the

names by which the object and the bag are known within the system. As an example, `"Zigbee"` and `"Sensors"` could uniquely define the bag containing tuples built as `(<sensorid>, <value>)` and corresponding to all the measurements done by the sensors currently connected to the system through the zigbee protocol.

Then, applying an operation on this bag (e.g. `rd()`) is noted `["Zigbee","Sensors"].rd(id,value)`. A `["Zigbee","Sensors"].rd(id,value)` operation in the precondition returns one by one the tuples matching the given pattern `(id,value)`, that is, in the present example, all the sensed informations collected by the system through the zigbee protocol. When the last corresponding tuple is returned, the `rd()` is blocked until a new matching tuple becomes available.

This mechanism is similar to the one a software program that is reading a file on which one or several other software programs are appending data. The read operation blocks when the pointer reaches the current end of the file and is automatically unblocked when new data become available, as the result of append operations performed by the other software programs. If one or several fields (e.g. `id`) are set with the identity of a specific sensor (e.g. `"Zig001"`) then `["Zigbee","Sensors"].rd("Zig001",value)` only returns the current value associated to this particular sensor. As soon as the sensor reads a new measurement, the corresponding new tuple is added to the bag and then will be returned by the pending `rd("Zig001",value)`.

In the precondition part, several *read*s can be sequentially invoked and the related instantiated variables are instantly right propagated, as it would be done in a classical logic programming approach. For instance, the following precondition will be true each time a new temperature is read by any of the temperature sensors in the system.

```
["Directory","SensorNetworks"].rd(network) &
[network,"Type"].rd(id, "temperature") &
[network,"Sensors"].rd(id, value)
```

The first `rd()` returns all the wireless sensor networks known by the system. Obviously, if a new wireless network is integrated in the system, and if the resource defining its availability is inserted in the bag `"SensorNetworks"`, then it will be automatically taken into account in the evaluation of the precondition. Each returned value will be instantly propagated in order to define the variable `network` used to designate the object name in which the `"Sensors"` and `"Type"` bags have to be queried. The second `rd()` gets the `id` of all the temperature sensors while the last `rd()` gets, for each of them, their mesured `value`. As a result, we obtain the search tree of Figure 2. Here, the computation item is seen as a ”virtual bag” that, each time a `rd()` is done, takes the first field of the tuple as input parameter, computes the `consign = compute_consign(value)` function, and then returns a tuple containing the result in the updated second field.

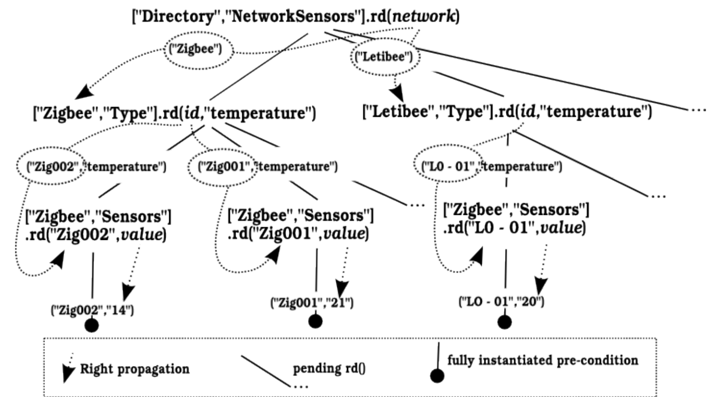

Figure 2. Evaluation tree

From the system side, it is seen exactly as any other bag. In the same way, it is possible to map *assertions*:

```
ASSERT: lib.lowerthan(value,threshold)
```

Here too, a virtual bag generates the expected resource if the condition is true. If the condition is not met, no resource is generated and the `rd()` is blocked.

The last facility allows to define of *temporization* in seconds:

```
TIMEOUT: 30
```

Here, the virtual bag waits the given delay before returning a tuple allowing the evaluation of the rest of the precondition.

For instance, if supplemented with an assert condition, the rule explained above will only consider the sensors with readings lower than 16°:

```
["Directory","SensorNetworks"].rd(network) &
[network,"Type"].rd(id, "temperature") &
[network,"Sensors"].rd(id, value) &
ASSERT lib.lowerthan(value,"16") &
COMPUTE lib.compute_consign(value,consign) &
[network,"Location"].rd(id, location) &
```

In addition, the reference that has to be transmitted to the heater according to the measured temperature is computed. Finally, the location of the temperature sensor related to the heater that has to receive this reference is retrieved. For the tree in figure 2 that contains 3 sensors, only the first one will actually trigger the performance phase.

The last point is to control the frequency of the precondition evaluation. Currently, each time a new measure is performed by a temperature sensor, the precondition is evaluated. This is obviously not suitable. Thus, at the beginning of the precondition, a `rd()` is added to a bag called `Ticket` that regularly generates resources used as tickets for triggering time-dependent actions. For instance, suppose that the `Ticket` bag generates a `("heating","ticket")` resource each 10 mn. Then, the precondition presented above is evaluated each 10 mn. If such a sensor exists, its location and the reference to apply to the related heater is computed. The full rule with its performance part can now be written:

```
["Clock","Ticket"].rd("heating","ticket") &
["Directory","SensorNetworks"].rd(network) &
[network,"Type"].rd(id, "temperature") &
[network,"Sensors"].rd(id, value) &
ASSERT lib.lowerthan(value,"16") &
COMPUTE lib.compute_consign(value,consign) &
[network,"Location"].rd(id, location) &
::
{
["Clock","Ticket"].get("heating","ticket") ;
[network,"Sensors"].rd(id, value) ;
["Heating","Actuators"].(location,consign)
}{
["Clock","Ticket"].get("heating","ticket") ;
[network,"Sensors"].rd(id, value) ;
["Alarm","SMS"].("06778899","heating problem")
}.
```

The performance part is made of 2 distinct transactions, basically corresponding to 3 different cases:

1) in the first phase of the first transaction, we check if the ticket `("heating","ticket")` is still available, if the temperature is the same as the one measured in the precondition, and if the heating system is manageable. If it is true, the second phase is performed: the ticket is consumed and the resource defining the consign for the heater is inserted in the bag controlling the heating system. Since, the resource corresponding to the temperature value might be shared by some other rules we do not consume it: a `rd()` is used instead of a `get()`; The second transaction has no chance to be performed as the ticket resource disappeared.
2) the temperature measurement has not been modified but, for some reason, in the first phase, the system has not been able to communicate with the heating system. In this case, the first transaction aborts, the ticket remains and the second transaction is tried. This second transaction sends an alarm via SMS to the user to warn her of the heating system problem;
3) if the temperature measurement has changed, then none of the 2 transactions are performed because the resource corresponding to the temperature of the precondition disappeared. In this case, the resource ticket is preserved in the system, enabling the precondition to be re-evaluated with the new temperature resource.

B. Other extensions

Two other mechanisms are provided to refine the evaluation mechanism of the precondition. Their goal is mainly to reduce the size of the search tree and to cut off earlier branches. These mechanisms are exemplified hereafter:

```
{*,!}["Clock","Ticket"].rd("heating","ticket") &
{*,!}["Directory","SensorNetworks"].rd(network) &
{*,600}[network,"Type"].rd(id, "temperature") &
{*,!}[network,"Sensors"].rd(id, value) &
ASSERT lib.lowerthan(value,"16") &
COMPUTE lib.compute_consign(value,consign) &
{1,!}[network,"Location"].rd(id, location) &
```

The first field into the curly brackets defines the number of awaited replies. It has already seen that `rd()` opens a stream of replies. With this new field, the system knows that after receiving the required number of values, the stream can be closed. For instance, a sensor is known to stay in a single location (a single location resource exists in the bag for the given sensor). The second field indicates how long (in s) to wait for resources while the `rd()` is pending. Here, for instance, the waiting for a temperature measurement will be no longer than 10 mn (i.e. 600 s), as a new ticket is issued every 10 minutes. Then, even if the temperature would finally be read after 10 mn, it would not anyway be validated in the precondition re-evaluation. Then, it is not useful to keep the stream waiting for this `rd()` open.

To be complete, it is worth to say that opened streams are automatically closed by the system when the resource on which depends the `rd()` stream gets consumed.

IV. APPLICATION

In this section, we illustrate the high level of expressiveness of the Chaski coordination language through the following application. Several seniors are living autonomously in their own habitation. Nevertheless, they need a daily checking. A pool of health professional has to check that the elderly people are fine.

Two organizations of these daily meetings can be proposed. The first one consists in fixing in advance the appointment between one elderly people and one health professional. Obviously, this way will not take into account the different unforeseen events:

- an elderly person is late at the appointment because involved in daily activities;
- a health professional is late because previous check-up lasted more than expected;
- emergency has to be taken into account;
- if a check-up is shorter than planned, the remaining time is just lost even if another check-up would be possible;
- the load is not balanced among the pool of health professionals, some are idle, some are overwhelmed.

The second way is to let to the application organizing the scheduling in real time by relying on the Chaski framework and its rules to capture information from different type of sensors and to synchronize distributed actions.

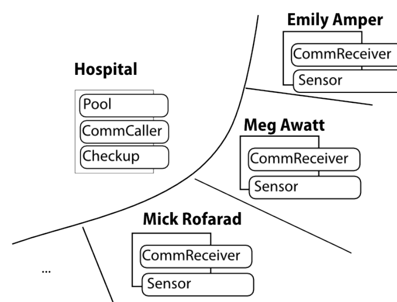

Figure 3. Overview of the distributed application

Figure 3 gives an overview of the distributed application with the decomposition in Objects/Bags:

`["Hospital","Checkup"]` is a bag containing information about the people to be contacted during a day;

`["Hospital"."Pool"]` manages the available doctors who can make the daily checking;

name
Emily Amper
Eloi de Moure
Mani Kor
Meg Awatt
Mick Rofarad
Kyle Hobbit

doctorname	status
"Dr. Home"	"idle"
"Dr. Whatelse"	"idle"

Table I
EXAMPLE OF RESOURCES CONTAINED IN CHECKUP AND POOL BAGS

`[ ?? ,"Sensor"]` encapsulates a presence sensor at the location of each elderly person;

`["Hospital","CommCaller"]` is a bag encapsulating the video conference equipment of the hospital;

`[ ?? ,"CommReceiver"]` is a bag encapsulating the video conference equipment at elderly person location;

The main rule is:

```
["Hospital","Checkup"].rd(senior) &
["Hospital","Pool"].rd(doctor,"idle") &
[senior,"Sensor"].rd("present")
::
{
["Hospital","Checkup"].get(senior) ;
["Hospital","Pool"].get(doctor,"idle") ;
[senior,"Sensor"].rd("present") ;
["Hospital","CommCaller"].put(doctor,"call",senior) ;
[senior,"CommReceiver"].put("accept",doctor)
}.
```

The bag `[senior,"Sensor"]` encapsulates a presence sensor that insert in the bag the information about the presence or not of the person. For instance, it can be a simple presence sensor put on top of a computer/laptop/pad screen. The bags `["Hospital","CommCaller"]` and `[senior,"CommReceiver"]` just encapsulate the video conference software module used by the system. It can use for instance the command line interface of skype.

The behavior of the rule is as follows. The first token returns all the people that are waiting for a checkup. The second token returns all the available Health professional. The third token corresponds to a presence sensor on the location of each of the elderly people. When one of them is detected as "present" then the precondition of the rule is verified and the performance part may be triggered.

Assume that the precondition becomes true with `"Mani Kor"` as `senior` and `doctor` is `"Dr Home"`. Then the performance part is:

```
{
["Hospital","Pool"].get("Dr Home","idle") ;
["Mani Kor","Sensor"].rd("present") ;
["Hospital","Checkup"].get("Mani Kor") ;
["Hospital","CommCaller"].put("Dr Home","call","Mani Kor") ;
["Mani Kor","CommReceiver"].put("accept","Dr Home")
}.
```

When the performance part is triggered, the initial situation responsible for the precondition could have changed and then it is required to verify that the condition is still true. This is done either using the `rd()` or the `get()` operations. Both verify that the required resource is still available but the second in addition will consume the resource in the second phase of the transaction. The different cases concerning the performance are discussed hereafter.

If the 3 resources `("Mani Kor")`, `("present")` and `("Dr Home","idle")` are still available then they are locked and the 2 last tokens verify that the video conference is technically possible (e.g. equipment "on" at both sides). If everything is fine then resources `("Dr Home","idle")` and `("Mani Kor")` are consumed and the video conference is launched. The consumption of resources `("Dr Home","idle")` and `("Mani Kor")` prevents the two people (doctor and senior) to be involved in another video conference because all the rules that require one of these resources will fail in the performance phase. When the check-up is finished, the doctor has just to notify the system that he is available again. As a consequence the resource `("Dr Home","idle")` is inserted in the bag `["Hospital","Pool"]`. This will reactivate all the rules waiting for a resource in this bag.

The transaction may cancel because:

- the doctor could have received an emergency. In this case, the corresponding resource is removed from `["Hospital","Pool"]`. This could be done automatically by removing the resource when the doctor id card used for authentication and tracing is removed from the reader associated to the computer;
- the doctor could be involved in another check-up if 2 check-ups were possible at the same time. Only one triggers the video conference and the other fails due to the absence of the resource `("Dr Home","idle")`;
- `"Mani Kor"` could already be with another doctor;
- `"Mani Kor"` could have left the room equipped with video conference system for some reason and then the resource `("present")` is no longer available;
- video conference equipment failed at one of the sides.

If the transaction fails then no resources are consumed. Thus, the precondition of the rule may become again true if the doctor or the senior becomes again available.

Concerning the faulty equipment, another transaction can be added after the first one and then triggered in sequence.

```
{
["Hospital","Pool"].get("Dr Home","idle") ;
["Mani Kor","Sensor"].rd("present") ;
["Hospital","checkup"].get("Mani Kor") ;
["Hospital","Support"].put("Pb","Dr Home","Mani Kor");
}
```

If the resources awaited by the 3 first token are available, this means that the previous transaction failed. Otherwise `("Mani Kor")` would have disappeared. Then this transaction is executed only if the first one fails. In this case

a resource `("Problem","Dr Home","Mani Kor")` is inserted allowing another rule waiting for it to manage the problem and to warn the support team for instance.

V. Discussion

The simple example given in the previous section shows the expressiveness of the Chaski language may simplify the design of distributed applications which need to put together data coming from different distributed locations.

ECA rules

This language goes beyond what is possible to do with Event-Condition-Action (ECA) rules [10] on different points. First, it unifies the event and the condition because they are seen in the same way by the system: an event and value condition are resources contained in the bags. This allows to have much more refined way to trigger the rules. Second, the event and the condition evaluation concern distributed data with no restriction about the distribution. Third, the verification of the condition is embedded in a transaction including also the actions to be performed. This guaranties the same logical time.

STITCH coordination language

As Chaski rules are an improvement of the STITCH coordination language, we discuss here the differences.

First, STITCH does not allow to consider alternative transactions in the same rule. This means that fall-back treatment is less natural and requires additional bags and explicit extra rules that makes the design of the application more complex. In addition, in STITCH, the insertion (put) is not embedded in the transaction but handled by a separate mechanism ensuring that the resource is eventually inserted. It obviously ensures the insertion but does not allow any treatment if the insertion cannot be done at the transaction time. This makes the system less flexible. For instance, this could not ensure the correct setting of the video conference in our example. Another difference is that, in STITCH, the exact same resource of the precondition is used in the transaction. Then, if the sensor reads many times the presence of a person and inserts each time a new identical `("present")` resource, this would cause the transaction to abort and to be retried with another resource with the same value. This generates much more work. The last difference that is not highlighted in the present example concerns the extension introduced in Section III-B that allows respectively to define the number of replies awaited and the maximum time to be blocked when pending.

VI. Conclusion

In this paper, a new abstraction layer is proposed. It provides a uniform view of the components - sensors, actuators and services - that are encountered in (wireless) sensors and actuators networks. Thanks to this abstraction layer, integration of geographically distributed software and hardware components is easier. Morever, to enforce the whole system consistency, sensing and actuation are embedded in transactions. Prototyping is accelerated thanks to the rule-based coordination language. The powerfulness of this middleware is exemplified on a realistic application.

Acknowledgment

This research has been sponsored in part by the European Project OUTSMART (PN 285038) under the 7th Framework Programme FP7-2011-ICT-FI.

References

[1] J.-M. Andreoli, F. Pacull, D. Pagani, and R. Pareschi, “Multiparty negotiation of dynamic distributed object services,” *Journal of Science of Computer Programming*, vol. 31, pp. 179–203, 1998.

[2] D. Arregui, F. Pacull, and M. Riviere, “Heterogeneous component coordination: the clf approach,” in *In Proc. of EDOC’2000, Makuhari*, 2000, pp. 194 – 203.

[3] J.-M. Andreoli, D. Arregui, F. Pacull, and J. Willamowski, “Resource-based scripting to stitch distributed components,” in *in Proc. of EDICS’02*, 2002, pp. 429–443.

[4] N. Carriero and D. Gelernter, “Linda in context,” *Commun. ACM*, vol. 32, pp. 444–458, April 1989.

[5] E. Freeman, K. Arnold, and S. Hupfer, *JavaSpaces Principles, Patterns, and Practice*, 1st ed. Essex, UK, UK: Addison-Wesley Longman Ltd., 1999.

[6] S. W. McLaughry and P. Wycko, “T spaces: The next wave,” in *Proceedings of the Thirty-second Annual Hawaii International Conference on System Sciences-Volume 8 - Volume 8*, ser. HICSS ’99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 8037–.

[7] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency control and recovery in database systems*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1987.

[8] “Book review: Rule based programming with ops5 by thomas a. cooper and nancy wogrin (morgan kaufmann pubtishers),” *SIGART Bull.*, pp. 14–15, July 1989, reviewer-Trowbridge, Timothy L.

[9] R. A. Kowalski, “The early years of logic programming,” *Commun. ACM*, vol. 31, pp. 38–43, January 1988.

[10] U. Dayal, A. P. Buchmann, and D. R. McCarthy, “Rules are objects too: A knowledge model for an active, object-oriented databasesystem,” in *Lecture notes in computer science on Advances in object-oriented database systems*. New York, NY, USA: Springer-Verlag New York, Inc., 1988, pp. 129–143.

Copyright (c) IARIA, 2011. ISBN: 978-1-61208-145-8