On Observing and Constraining Active Systems

Gianluca Moro, Mirko Viroli

DEIS - University of Bologna
Via Rasi e Spinelli, 176
I-47023 Cesena (FC)
{gmoro,mviroli}@deis.unibo.it

Abstract. While agents have emphasised the notion of active software components, they are not likely to be the only active components in agent-based systems. In this paper, we first discuss the general notion of active system, and show how it relates with the issue of the consistent observation of distributed and heterogeneous multi-component systems. Then, we introduce the concept of boundary interface as a methodological abstraction for the engineering of active systems, which allow observer agents to be provided with three different kind of consistent system views, featuring observable, controlled, and constrained consistency, respectively.

1 Introduction

Agent-based systems have emphasised the notion of active components, which do not simply react to invocations or queries, but exhibit instead some observable behaviour as emerging from their inner computational activity. Agents [24] are indeed such sorts of systems, but what is emerging from current experience with the engineering of complex software systems is that they are not likely to be the only kind of active entity populating the agent space.
This is easy to see when taking into account systems modelling highly dynamic domains, where some knowledge sources represent portions of the world whose state is changing in an unpredictable way. Agents in charge of monitoring or controlling such components are unlikely to be designed as polling observers, continuously checking knowledge sources to verify possible state changes. Instead, they are more easily to be thought of as sorts of registered observers, which have to be notified when something happens in the part of the world they are interested in. As a result, the general notion of an active system comes in, which includes not only agents, but also any component or set of components of an agent-based system which is able to manifest any change to its state as an observable event.
However, the typical structure of the space where agents live and work, as a heterogeneous, distributed, unpredictable environment with no centralised control, introduces the issue of an agent's consistent observation. Generally speaking, agent systems are often built by putting together new components with legacy ones, by modelling semantic relationships between different components which

are often not easy to be implemented. This makes it difficult or even impossible to provide agent observers with a consistent view of the resulting system, in that the observation may require a higher abstraction level with respect to the one provided by single components.

Let us consider an example derived from a real application experience of a big hospital in Bologna [15]. A patient is subjected to several medical exams in different wards A, B, C, respectively managed by three different subsystems built over the time as independent parts without cooperative behaviours. An observer agent, representing for instance the patient's family doctor, is not interested in the single exam, carried out in the single ward, but only in the three exams as a whole, which represents for him a consistent view of his patient. But the three subsystems do not cooperate, so how can the problem be resolved without rebuilding all over again?

The first obvious solution is to charge the observer agent with the burden of consistency. However, this doesn't seem to be an effective and satisfactory approach: when heterogeneous subsystems are semantically correlated, it is likely to have many different observers asking for the same kind of consistent view, connecting sub-components at a higher level of abstraction. If observer agents directly accept the notifications sent by the systems, the computational overhead needed to obtain the consistent view would be replicated into each agent, causing problems of redundancy, maintainability and correctness. So, semantic correlations between structurally unrelated components are more likely to be faced outside the observers, by factorising consistency in some infrastructural abstraction.

To this end, the first goal of this paper is to introduce an abstraction, called boundary interface, which can be used as a methodological tool to build up observably consistent active systems from possible heterogeneous, independent, distributed, and decentralised subsystems, freeing observer agents from the charge of implementing consistent views. We go on to discuss how such an abstraction can be differently instrumented, to provide for three different notions of consistency: observable, controlled, and constrained, at a growing level of expressing power. As a result, the notion of boundary interface subsumes the twofold role of providing observer agents with a consistent perception of complex observed systems, and possibly controlling of their evolution over time according to some super-imposed consistency constraints.

The reminder of the paper is organised as follows. Section 2 describes the conceptual framework of the boundary interface, illustrating its features and its motivations. Section 3, 4 and 5 describe the internal architecture for the boundary interface when it materialises respectively an observable consistency, controlled consistency and a constrained consistency. Section 6 shows an example of monitoring, controlling and constraining of an active system consisting in mobile robots. To conclude, Section 7 reports the related works.

2 Conceptual Framework

When a system effectively represents portions of the world, the status evolution of the world coincides with the status evolution of the system. For instance, the status evolution of a database represents the status evolution of the portion of the world the database models. Systems are designed for serving the concurrent requests of proactive agents preserving their internal consistency (i.e. constraints and rules of the modeled information). In order for an observer to take decisions correctly, the observed system should be able to reveal only consistent views. Known models and technologies [18, 1] allow a system to preserve and supply consistent views of its status only when dealing with client-server like query requests.
However, in order to discover possible changes, an observer should not have to work on a polling basis, continuously querying the system, because of the following reasons:

- *lack of effectiveness* - this approach does not guarantee the perception of all status changes over the time, in fact the system could change more rapidly with respect to the query frequency;
- *lack of efficiency* - the system is forced to serve all the queries even when nothing is changed, thus wasting computational resources.

To overcome these problems, observers should be notified by the system itself only when something they are interested in happens. To this end, system components should be designed as active systems, explicitly manifesting their status changes. Each observer interested in some portion of the system should register itself to one or more sub-components, and then be notified when any of these sub-components change. This is a very common interaction pattern typically known as the publish/subscribe protocol. Its importance in the structuring of active systems has been highlighted in many different areas: active databases [1, 8], object frameworks [7, 18] and coordination models [20, 4].
In this way however, each observer agent should also carry out the task of rebuilding consistent views of the system's states of interest directly from sub-component events. Indeed, this task often requires some low-level knowledge on the system and on the update operations involving those components. So, it is likely to be implemented and executed in a redundant way by each observer, with risk of errors and waste of computational resources. Moreover, the resulting system would be unmanageable, since extensions or modifications would be replicated in each observer agent.
Our proposal is to factorise the process of building and notifying consistent views of a multi-component system in an abstraction we call *boundary interface*. This software layer is a sort of output interface complementary to the traditional usage interface (or rather entry interface). Its main goal is to publish a set of higher level events grouping system events in macro-events, each representing an atomically notifiable, consistent view, of the multi-component system.
In general, the observation can involve two or more systems or subsystems possibly independent and without cooperative behaviours. In this case the boundary

interface could realise a higher level system, resulting from the combination of independent sub-systems, by super-imposing higher level consistency rules. Let us consider the hospital example mentioned above. The higher level consistent view of the patient, required by the family doctor, does not belong to any of the three subsystems, but it could be built by the boundary interface. This will be mainly realised by collecting and processing the events notified by all the single subsystems.

When the boundary interface builds consistent views of a multi-component system, we say that it realises the notion of *observable consistency*. The boundary interface, in this case, allows registered observers to perceive consistent manifestation of status changes of the system $\mathcal{S}$, intending $\mathcal{S}$ as a multi-component system. The only requirement for $\mathcal{S}$ is the capability of issuing the status changes of its parts (i.e., its components or subsystems).

With this framework the boundary interface can be easily though as a software layer which not only monitor $\mathcal{S}$ and notify registered observers, but which can also control in some way the evolution of $\mathcal{S}$. In this case, the system $\mathcal{S}$ should be able to issue the proposals of the status changes of its parts before they are actually performed. The boundary interface may then decide to refuse or accept these proposals by applying/using new super-imposed notions of consistency. If and only if these proposals are accepted then the corresponding status change becomes effective.

For instance, let us consider two independent industrial robots consisting of two mobile arms. Each one proposes its moving intention to the boundary interface, which decides to refuse or accept them according to super-imposed rules. These rules may be designed, for example, to avoid collisions or dangerous movements. Notice that this notion of consistency cannot belong to any of the two robots, but is more likely to be viewed as belonging to the overall system, made up of the two robots as a whole. When dealing with this new semantics, we say that the boundary interface implements a notion of *controlled consistency*.

Another task that we may need the boundary interface to implement is the constraining of the evolution of the system. In this case the requirement for $\mathcal{S}$ is to both issue status changes of its parts and to accept status change requests from the outside. In this scenario, the boundary interface may act as a constrainer, since it can monitor the evolution of the system and decide to change its evolution by sending some change request. Namely, the boundary interface constrains the evolution of $\mathcal{S}$ by pushing it towards a specific direction (intended as status) according to a super-imposed consistency. Considering the previous example of the two robots, a super-imposed consistency could establish that when the robot *R1* is late on some job with respect to *R2*, the boundary interface may constrain *R2* to slow down in order to be aligned with *R1*. In this case, we say that the boundary interface realises a notion of *constrained consistency*.

3 Observable Consistency

Consider again the system $\mathcal{S}$, which is composed of different, possibly distributed and heterogeneous sub-components, each able to notify the events of changes on its status or on a part of it.
When we talk about "events" we explicitly refer to the publish/subscribe protocol. In particular we consider the case in which a publisher, the active system, fires events to one or more subscribers, the observers. Each publisher exports one or more types of event, and a subscriber may register for one of them by sending a specific request message. So the publisher, when some condition is satisfied or an action occurs, may decide to fire one event to all the subscribers which have registered for it. This event consists in a message which typically carries information about the publisher (source of the event), the cause of the event and some data value. In the remainder of the paper we will focus on those active systems which are able to manifest their activity using the publish/subscribe protocol, i.e., that are able to behave as publishers.
We call *system events* the change events that an active system $\mathcal{S}$ may fire. *Observers* are entities of the domain which can interact with $\mathcal{S}$ and are interested in its changes. They should behave as subscribers in the publish/subscribe protocol.
In general, the events the observers are interested in, which we call *output events*, belong to a different abstraction level than that of system events, which just represent changes on subparts of $\mathcal{S}$. So, the boundary interface is in charge to allow observer agents to perceive system changes by filtering and processing system events and producing output events. From the point of view of the publish/subscribe protocol, the boundary interface can be seen as a subscriber for system events and as a publisher for output events, while observers will behave as subscribers for output events. In this specific case the boundary interface will be called *observation interface*, and we say that it materialises an *observable consistency*.
Notice that in our framework we are not interested much in having the boundary interface dynamically registering for events, but we suppose that at startup-time it is instantiated once for all and connected in a static way to the corresponding active systems. The tasks the observation interface implements are: (i) to correlate system events using some super-imposed policy, (ii) to group them and perform some computation over the data they carry, (iii) to pack output events, and (iv) to fire them to the interested observers. The architecture of the observation interface is shown in Figure 1.
When computing the data values carried by multiple events, a preliminary and fundamental operation, which we call *correlation*, is always requested. By the term *correlation* we mean the identification of a relation or a property over the events, whose satisfaction causes the data values they carry to be grouped and computed as a whole.
In our architecture the correlation rules will be typically designed in order to identify the common cause for the generation of some system events, leading to the firing of one or more output events.

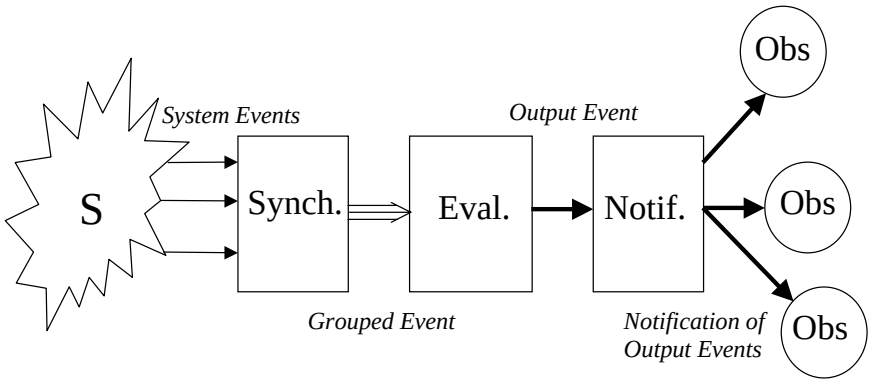

Fig. 1. Sub-Components for the Observation Interface

The part of the observation interface which realises this job is called Synchroniser. It takes all the system events which have to be processed, it classifies them, and it decides when to group some of them in order to fire a *grouped event* to the next sub-components. A grouped event is an event which carries the data values of a set of (correlated) system events; we use it in order to notify multiple data values into a whole. The strategies of synchronisation (also called correlation strategies) may be different, depending on the kind of system we are observing and on the kind of elaboration that has to be performed. Some of them are:

- *Temporal correlation*, events coming at close times are grouped together, and the firing of a grouped event is performed when an internal clock sends its tic.
- *Size correlation*, events are put into a buffer and sent when the buffer becomes full.
- *Data correlation*, events carry some information that is used both to classify them and to decide when to send a Grouped Event.

When a grouped event is fired it reaches the Evaluator component. Its task is to produce the actual output event that will be notified to the observers. This is typically realised by performing some functional transformation over the data values the grouped event carries, and producing as a result an output event.
The final component of the observation interface is the Notifier. Its task is to keep track of the observers' registration, to accept output events from the Evaluator, and to decide which observer has to be notified. It exports a statically-fixed set of output events, representing consistent changes on logical sub-parts of $\mathcal{S}$, and observers should specify those in which they are interested. So the Notifier has to classify the events the Evaluator sends and decide which observer to notify.
When the output events are logically disjoined the implementation of the Notifier is quite simple. More likely there can be some causality relation between the output events, leading to implementations that provide notification of all the observers which have registered for an output event that has some relation with

the one that is occurring.
Consider in fact the case in which the observation interface monitors two subsystems $\mathcal{A}$ and $\mathcal{B}$ and exports three events which respectively represent: changes on $\mathcal{A}$, changes on $\mathcal{B}$, changes on either $\mathcal{A}$ or $\mathcal{B}$. When an event of the third kind occurs the observers for the first one and the observers for the second one might be notified too.
Another interesting issue is the way a Notifier should deal with multiple registrations, i.e., those cases in which a single observer registers for more than one output event. Suppose an observer is interested in receiving both the changes on $\mathcal{A}$ and $\mathcal{B}$, and registers for both the corresponding output events. Suppose then that due to a message coming from the Evaluator the Notifier is willing to notify both the output event $\mathcal{A}$ and $\mathcal{B}$. If the observer receives both the notifications it would have still the problem of consistency, with the need of a further step of synchronisation. Given a single message coming from the Evaluator, a feasible notification policy should provide each observer for the notification of just one message, possibly containing the information about more than one output event occurred.

4 Controlled Consistency

When the only task of the boundary interface is to allow observers to perceive the system changes, then the unique requirement for the system is to be able to fire events representing internal changes.
Another interesting task for the boundary interface is to control the evolution of the system by observing its events and replying to each of them with an acceptance or a refusal feedback message. In this case, the system requirement is not just to be able to notify changes, but to propose changes waiting for their acceptance before the actual committing. When the boundary interface can handle this event's semantics we call it a *control interface*, and we say it materialises a *controlled consistency*.
The control interface super-imposes some (consistency) rules which are used to control the evolution. The corresponding semantics is somewhat similar to that of the interaction between a constrained property and the corresponding vetoer in the JB (Java Beans [10]) architecture. It can be consider a direct extension of the publish/subscribe protocol, in which the publisher waits for a feedback from the subscribers each time an events is fired. Examples of systems which can handle this behaviour are the transactional ones, which can be used so that if an event handler routine fails (refuse message from the observer) the transaction which cause the event to be fired fails too, and is rolled back. This is in fact the typical usage of a vetoer object in an EJB (Enterprise Java Beans [7]) framework. Other examples may include non-transactional systems composed by agents which send the notification and simply waits for the response, possibly carrying on a parallel activity.
Notice that in order to avoid any problem of autonomy of control, a system should export "controllable" events only if it is actually able to deal with refusal

feedbacks. In fact an events will be considered actually occurred by the control interface only if it has been accepted.
An architecture for this kind of boundary interface may be the one shown in Figure 2.
The first component of the interface, as for the observation interface, is the Syn-

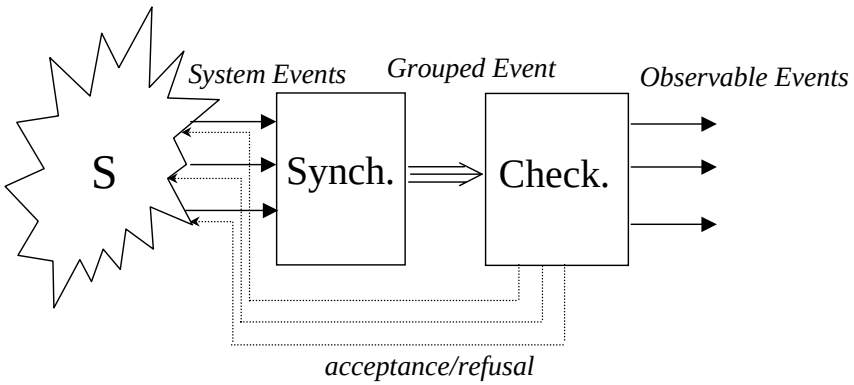

Fig. 2. Sub-Components for the Control Interface

chroniser. In fact, we already argued that the operation of correlation is always necessary when computing data coming from different events. The Synchroniser will be realised as previously discussed for the observation interface: it collects system events using some local correlation policy, and then fires grouped events to the next component.
In this case the grouped event reaches the Checker, which stores the rules used to decide if each single system event should be rejected or accepted. When this decision has been taken, it is communicated to the system through a feedback message.
In this architecture, one may think of allowing the decision of the Checker to be observed. The system events which have been accepted may in fact be notified to interested observers for further processing. We call the events a control interface may fire *observable events*. As we will see in section 6, these events may be handled as system events by a subsequent interface, allowing for instance to compose a control interface with an observation interface.

5 Constrained Consistency

A more powerful way of interacting with the evolution of a system than by just controlling it through the acceptance or the refusal of its proposal of changes, is to constrain its processing by querying new updates. So, another interesting pattern of interaction between a system and its boundary interface is the one in which the latter acts as a constrainer for the former.

To this end the system should be able to ask for completion proposals. It notifies internal changes and then accepts and executes corresponding external requests for new changes. Suppose the system is attached to the interface through n events, and that at a given time it notifies changes only through a subset of them. The boundary interface therefore may try to complete the information the system has sent by calculating acceptable/required values for the remaining events, and notifying them to the system through feedback messages, that we call *completion events*. The system may then try to perform the changes requested, possibly replying with a refusal message in case of fail.
When dealing with this semantics, we call the boundary interface a *constraint interface*, and we say that it materialises a *constrained consistency*. In Figure 3, the architecture for this kind of boundary interface is shown. Its structure

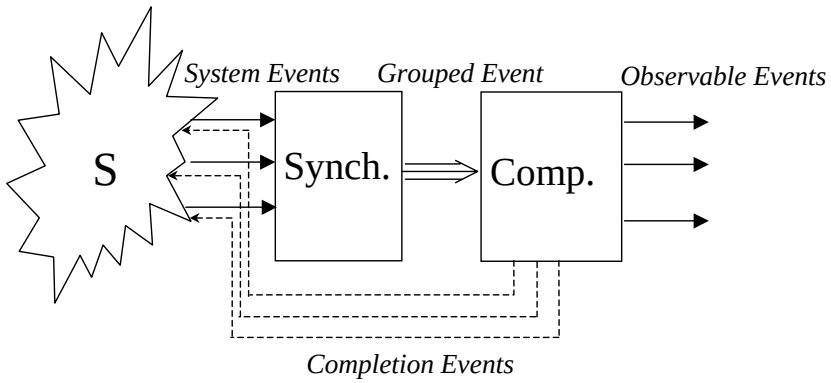

Fig. 3. Sub-Components for the Constraint Interface

is very similar to that of the control interface, in that both interact with the system in order to monitor and then adjust its evolution. The main difference is that in the first case the system proposes changes, while in the second it asks for a completion on the information it has.
As usual such completion is implemented in two steps: one in which events are correlated and grouped together (synchronisation task), and another in which some elaboration is done over the values carried by system events. When the completion has been determined, the Completer component notifies the system with the new data. Here again, in order to preserve the autonomy of control, the System should issue events to a constrained interface only if it is able to accept (and possibly refuse) request of changes corresponding to each of them.
In case of a transactional system $\mathcal{S}$ the processing caused by the completion events should be executed within the transaction which fired the corresponding system events. This can be achieved if the system works using the two-phase commit protocol [1] and the constraint interface is a participant of the transaction.

In an agent-based system an appropriate protocol of interaction should be designed in order to support the semantic of requested completion.
Analogously to the control interface furthermore, the constraint interface may let some events, which we call again observable events, to be notified to interested observers. Considering a completion request, observable events are fired if and only if $\mathcal{S}$ accepts all the completion events. In this case both system events and completion events are fired, since they represent the right view of the constrained system. Obviously $\mathcal{S}$ should treat the completion events' requests, so that they won't cause new proposals for completion again. This in order to avoid recursive behaviours.

6 An Example of Application

In previous sections we saw three different patterns of interaction between a system S and its boundary interface. In each case we analysed the requirements for the system and an architecture for the boundary interface, which we renamed respectively observation interface, control interface, and constraint interface. In this section we show a brief example in which these three kinds of boundary interface can be exploited.
Suppose our system $\mathcal{S}$ is composed by three mobile robots, namely $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$, which moves into a given area. Some observer agents are interested in monitoring some information like the position of each robot, the mutual distance, which robots are going to collide, and so on. We suppose then that all observers need to be notified at a given frequency, say each time per second, and that each robot can be viewed as an active system which notifies changes on its position with an unpredictable dynamic.
An infrastructure to solve this problem may exploit an observation interface which takes the three system events from $\mathcal{S}$, namely `pA`, `pB` and `pC` (the position (x,y) of each robot) and exports 9 output events: the three positions `oA`, `oB` and `oC`, the three mutual distances `dAB`, `dBC` and `dCA`, and three boolean values stating when a given robot is near enough to another robot or to an obstacle, say `bA`, `bB` and `bC`.
The Synchroniser includes an internal clock which fires a tick each time a second. This tick causes a grouped event to be sent, carrying the oldest position received from each robot since the last tick occurred. The Evaluator computes the 9 values required for each output event, possibly using cached values for those positions which has not changed since the last time a grouped event arrived. The Notifier notifies each observer with those values which it has registered to, and which has been updated since the last time. For example, if since the last Synchroniser's tick occurred, only robot $\mathcal{A}$ has changed its position, then only output events `oA`, `dAB`, `dCA`, `bA`, `bB` and `bC` will be sent.
Now suppose each robot does not notify changes on its position, but proposes new ones, which should be considered carried out only in case of acceptance. In this case we may think to use a control interface in order to avoid the distance between any two robots to be smaller than a give parameter. Having not

to directly deal with observers, the Synchroniser of this interface may behave differently to that of the previous case, for example it can fire a grouped event each time a system event occurs, that is considering all the update proposals not to be correlated in any way. The Checker computes the distance between the three robots using the new value for the one which is willing to move and the oldest ones for the others two, sending a refusal if any of the three distances is too small.
The system composed by the three robots ($\mathcal{S}$) and this control interface can be considered a controlled system, which is able to manifest its evolution through three system events (which are actually observable events of the control interface, according to the architecture explained in section 4). Hence we may apply the observation interface depicted in this section in order to allow observer agents to monitor this "collision-safe" system. See figure 4.
Finally suppose each robot both notifies proposals for moves and accepts new

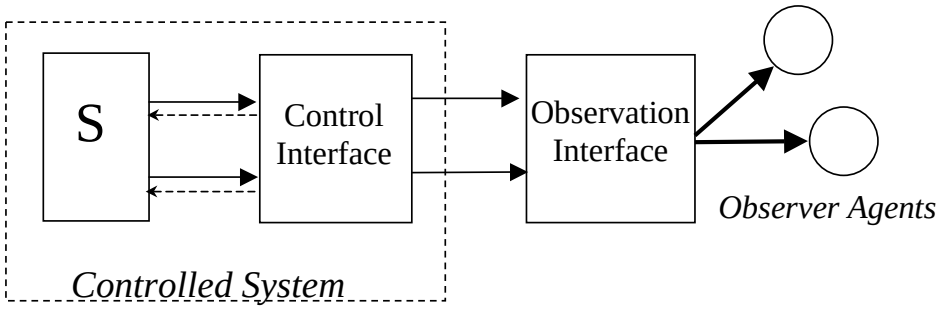

Fig. 4. Observing a controlled system

positions to move to. In this case a constraint interface can be applied to the system in order to force the robots to stay sufficiently near to each other. So when a robot, say $\mathcal{A}$, decides to explore a new area, the other two can be constrained to follow it. This can be realised for example with the same Synchroniser of the previous control interface, and with a Completer which constrains the two robots which are not moving (in this case $\mathcal{B}$ and $\mathcal{C}$) to follow $\mathcal{A}$, i.e., to move to a position near $\mathcal{A}$. Again an observation interface can be applied to let observer agents to monitor these three "friendly" robots, as shown in figure 5.
To conclude the abstraction we called boundary interface allows an active system to interact with some observer agents materializing some super-imposed consistency rule. It becomes natural to see this as a pattern of interaction between a multi-agent system and its environment. With this respect the observation interface, the control interface and constraint interface are specialization of the boundary interface, materializing different kinds of consistency (observable, controlled, constrained). When composing a control interface and an observation interface, we identify the boundary interface with the sum of them, being a component which allows both the control and the observation of a system.

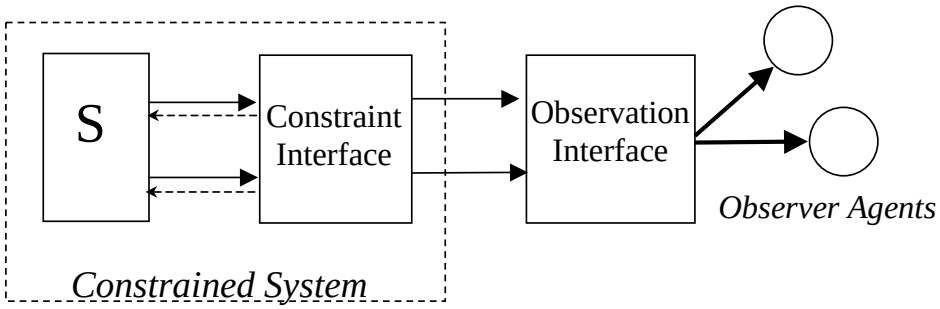

Fig. 5. Observing a constrained system

Analogous argument can be issued in the case of observation of a constrained system.

7 Related Works and Conclusions

Typically, a system is thought and designed only by facing the point of view of its use, that is, by providing for a set of functionality that some client may use to manipulate it and possibly to change its status. For this class of clients effective models and methodologies which guarantee that the system transits always across consistent status have been proposed, like Transaction Processing System [1, 13].
In this paper, we focus on observers, which are mainly interested to be notified by the system when its status (or part of it) changes. Of course, in this case also, it is necessary that the observers are notified of consistent status changes of the system. However, the problem of determining models and frameworks for allowing consistent observations has not received the same attention. Currently, relevant research areas like object oriented systems, DBMS and agent technology seem to offer only implementation mechanisms for the observation. In fact, system's observers typically just accept events from reactive elements of the system. For instance, in object oriented systems, active entities are built over frameworks (like Enterprise Java Beans [7] and Corba [18]) exploiting design patterns such as Gamma's observer [9] and Schmidt's reactor and active object [21], which don't face the problem of consistency. In previous works [16, 17] we proposed a solution applyable to object applications, featuring an architectural pattern based on the observation interface.
In commercial DBMSs also (Oracle [19], Sybase [22],etc.) the problem of making the database active is solved at a mechanism level. In order to allow the notification of changes to observers, the database architecture has been modified by adding triggers [12, 14]. Triggers are pieces of code attached to database tables that can notify to a given port events of insertion, deletion and update of tuples (see for example Oracle Triggers [19]).
Some research prototypes of active DBMS have been developed on the basis

of Event-Condition-Action model (ECA) and active rules model [1, 8, 6]. Some examples are HiPAC [5], Starburst [23] and Chimera [2]. Most of them support composite events [3] that allow internal ECA rules to be activated when two or more events occur. Our Synchroniser component performs a task that could be analogous to a composite event detection, but which is implemented outside the DBMS. This allows existing databases with minimal active capabilities to be turned into full-featured observable systems thanks to the observation interface. Some coordination models [4, 20] make use of publish/subscribe capabilities as well. JavaSpaces [11] for example deals both with event notification and transactional consistency. Its tuple spaces can be programmed so to notify registered observers when a tuple matching a given template is inserted. A request for notification can specify whether the events should be fired directly when the tuple is inserted or waiting for the corresponding transaction to commit. Supposing a system's change is represented with a tuple inserted in a JavaSpace, and that a committed change can be considered consistent, this management allows for consistent changes to be notified to registered observers. With our framework a JavaSpace can be seen as an active system which notifies its changes, hence we can apply an observation interface in order to add a super-imposed observable consistency, hence monitoring a co-ordinated system evolution. It is not clear in which way JavaSpace could be modified or extended in order to satisfy the requirement needed to be controlled and constrained.
In the agent based systems, to our knowledge, the problem of consistent observation has not yet been addressed in a systematic way. However, given that agent are likely to act in software system more and more on behalf of human beings, to sense the world and act on it consistently and timely by their own initiative, this problem is going to become more relevant for multi-agent systems than for any other kind of system. That is why we argue that the problem tackled by this paper, even though of general interest, seems to be even more important for agent-based systems.
Independently from our proposal, we believe that the design of multi-agent systems for typical real-life applications should also consider and study the ideas of observation, control and constraint according to notions of consistency. From the implementation point of view we are interested in deepening the interaction of transactional systems with all the three kinds of boundary interface.

References

1. Atzeni P., Ceri S., Paraboschi S., and Torlone R., Database Systems Concepts, Languages and Architectures. McGrawHill, pag. 441-461, ISBN 007 709500 6, 1999
2. Ceri, S., Fratemali, P., Paraboschi, S., and Branca, L. (1996). Active Rule Management in Chimera. *In Active Database Systems - Triggers and Rules For Advanced Database Processing*, pages 151–176. Morgan Kaufmann.
3. Chakravarthy, S., Krishnaprasad, V., Anwar, E., and Kim, S. K. (1994). Composite Events for Active Databases: Semantics Contexts and Detection. *In Proceedings of the 20th International Conference on Very Large Data Bases*, pages 606–617.

4. G.Cabri, L.Leonardi, and F.Zambonelli. Reactive Tuple Spaces for Mobile Agent Coordination. *In Proc. 2nd Int. Workshop on Mobile Agents*, volume 1222 of LNCS, pages 213-228. Springer-Verlag, Berlin Germany, 1998.
5. Dayal, U, Buchmann, A, and Chakravarthy, S, The HiPAC Project, *In Active Database Systems (Eds. J. Widom and S. Ceri)*, pp. 177-206, MK, 1996.
6. U. Dayal, E. N. Hanson, J. Widom, Active Database Systems, Addison Wesley, September 1994
7. Enterprise Java Beans Specification, 1999, Sun Mycrosystems, http://www.javasoft.com/products/ejb/docs.html
8. P. Fraternali, L. Tanca, A Structured Approach for the Definition of the Semantics of Active Databases, ACM Transaction on Database Systems, Vol. 20, No. 4, pp. 414-471, December 1995.
9. Gamma, E., Helm, R., Johnson, R., and Vlissides, J. Design Patterns : Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, Mass., 1995.
10. Java Beans Tutorial, Sun Mycrosystems, Sept. 1997, http://java.sun.com/beans/docs/Tutorial-Sep97.ps
11. JavaSpaces Specification, Sun Mycrosystems, Jan. 1999, http://www.sun.com/jini/specs/js.pdf
12. S. Kim, S. Chakravarthy, A Pratical Approach to Static Analysis and Execution of Rules in Active Databases, ACM, Las Vegas Nevada USA, pag. 161-168, 1997.
13. Sherri Kennamer, Microsoftcorn: A High-Scale Data Management and Transaction Processing Solution, SIGMOD'98 Seattle. WA, USA, 1998 ACM pag. 539-540
14. G. Kappel, W. Retschitzegger, The TriGS Active Object-Oriented Database System - An Overview, Technical Report of Department of Information System, University of Linz, 1998.
15. G. Moro, A. Natali, M. Viroli, An Interactive Computational Model for Monitoring Systems, Tech-report. DEIS-LIA-006-99 n. 40, DEIS - University of Bologna
16. G. Moro, A. Natali, M. Viroli, An Architectural Pattern for Consistent Observation of Active Systems. *In Workshop Readers of the European Conference on Object Oriented Programming, 2000, Lecture Notes in Computer Science*, Cannes 2000.
17. G. Moro, A. Natali, M. Viroli,On the Consistent Observation of Active Systems. *Proceedings of the AI*IA/TABOO Joint Workshop 'Dagli oggetti agli agenti: tendenze evolutive dei sistemi software' (WOA 2000)*, Parma, May 2000, http://lia.deis.unibo.it/books/attiWOA2000/.
18. Object Management Group, CORBAservices: Common Object Services Specification. Revised 09/12/98.
19. Oracle Corporation, PL/SQL User's Guide and Reference Release 2.3, Part No. A32542-1, chapter 8, February 1996
20. A.Omicini and F.Zambonelli. Coordination of mobile information agents in Tucson. *Journal of Internet Research*, 8(5), 1998.
21. Schmidt, D. Using design patterns to develop reusable object-oriented communications software. Commun. ACM 38, 10 (Oct. 1995),65-75.
22. Sybase. Sybase SQL Reference Manual: Volume 1. Sybase, Inc., 1996.
23. Widom, J. (1996). The Starburst Active Database Rule System. *IEEE Transactions on Knowledge and Data Science*, 8(4): 583–595.
24. Wooldridge, M.J., Jennings, N. R., Intelligent Agents: Theory and Practice, The Knowledge Engineering Review 10:2, pp. 115-152. Cambridge University Press, 1995.